Cracking Salted Hashes

Web Application Security: - The Do's and Don'ts of "Salt Cryptography"

Overview:

Data Base security has become more critical as Databases have become more open. And Encryption which is one among the five basic factors of data base security.

It's an insecure practice to keep your sensitive data like Password, Credit Card no etc unencrypted in you database. And this paper will cover the various Cryptography options available and do and don'ts of them.

Even if you have encrypted your data that doesn't mean that your data's are fully secured, and this paper will be covered in an Attacker perspective.

Slat Cryptography.

http://en.wikipedia.org/wiki/Salt_(cryptography)

Assume a user's hashed password is stolen and he is known to use one of 200,000 English words as his password. The system uses a 32-bit salt. The salted key is now the original password appended to this random 32-bit salt. Because of this salt, the attacker's pre-calculated hashes are of no value (Rainbow table fails). He must calculate the hash of each word with each of 232 (4,294,967,296) possible salts appended until a match is found. The total number of possible inputs can be obtained by multiplying the number of words in the dictionary with the number of possible salts:

2^{32} \times 200 000 = 8.58993459 \times 10^{14}

To complete a brute-force attack, the attacker must now compute almost 900 trillion hashes, instead of only 200,000. Even though the password itself is known to be simple, *the secret salt makes* breaking the password increasingly difficult.

Well and salt is supposed to be secret, to be simple if the attacker knows what salt is used then we would be back again to step one. So below listed are few possible ways you could use to crack salted hashes.

Application that doesn't use cryptography hashes:

While auditing a web application I came across this piece of program. It used java script to encrypt the user password before sending. And a salt too was used, and the salt used was the current Session ID.

```
onclick="javascript:document.frm.id.value='user';

document.frm.passwd.value='value';

this.form.passwd.value=(hex_md5('CC6AB28BA9FAD121184B09E00F1DD6E7'+this.form.passwd.value));

this.form.submit();
```

Where "CC6AB28BA9FAD121184B09E00F1DD6E7".

So In the Back End program:

There would be no way for the back end program to verify the password value, as the salt is used and is the random session id. And as MD5 function are non reversible hash function, the password cannot be verified unless and until the passwords are saved as clear text in the Data Base.

The salt used to encrypt the password before sending were the random generated session ids. That means that the back end databases are no way encrypted.

Some time these kinds of coding gives away a lot of information.

Point NO 1: Always encrypt and save your passwords database.

Slated Hashes.

I recently came across a huge Data Base of a well known !@#$$il[Encrypted] ☺, the Database contained Email-Ids and there alternate passwords, but unfortunately the passwords were encrypted and was in hashed format, "Good Practice". So the following paper would be in respect to how I cracked those hashes.

Cracking The Hashes:

The few possible way to crack hashed passwords are:

1) The algorithm used for hashing should have some flaws and hashes should be reversible
2) Or that you will have to Brute force the hashes with a wordlist of Dictionary or Rainbow tables.
3) Or simply if you have UPDATE Privileges on that Data Base Update it with a know password's hash value.

For all of these attacks to work you need to know what algorithm the hashes are computed on.

So what is that you could do to figure out the Hashing Algorithm used??

Answer: All algorithms generate a fixed length hash value, so based on the Output you could estimate what algorithm was used☺. "Well all these things are pretty know facts ", but Still am putting it here.

For this am putting here a small Cheat sheet for figuring out the Hash functions based on the output.

Language: / Algorithm:	PHP	ASP	JAVA
MD5	Function: md5("input"); Hash("input"); **Output: 32 Char** Ex: "5f4dcc3b5aa765d61d8327deb882cf99"	Function: System.Security.Cryptography **Output: 32 Char** Ex: "5f4dcc3b5aa765d61d8327deb882cf99"	Function:java.security.MessageDigest **Output: 32 Char** Ex: "5f4dcc3b5aa765d61d8327deb882cf99"
Salt+Crypto	Function: Crypt() By default its DES Output: 13 Char Ex: "sih2hDu1acVcA"	" "	" "

And lot others:

Original Source: http://www.insidepro.com/eng/passwordspro.shtml

Hash Type	Hash Example	Additional Information
DES(Unix)	IvS7aeT4NzQPM	Used in Linux and other similar OS. Length: 13 characters. Description: The first two characters are the salt (random characters; in our example the salt the string "Iv"), then there follows the actual hash. Notes: [1] [2]
Domain Cached Credentials	Admin:b474d48cdfc4974d86ef4d24904cdd91	Used for caching passwords of Windows domain. Length: 16 bytes. Algorithm: MD4(MD4(Unicode($pass)).Unicode(strtolower($username))) Note: [1]
MD5(Unix)	$1$12345678$XM4P3PrKBgKNnTaqG9P0T/	Used in Linux and other similar OS. Length: 34 characters. Description: The hash begins with the 1 signature, then there goes the salt (up to 8 random characters; in our example the salt is the string "12345678"), then there goes one more $ character, followed by the actual hash. Algorithm: Actually that is a loop calling the MD5 algorithm 2000 times. Notes: [1] [2]
MD5(APR)	$apr1$12345678$auQSX8Mvzt.tdBi4y6Xgj.	Used in Linux and other similar OS. Length: 37 characters. Description: The hash begins with the $apr1$ signature, then there goes the salt (up to 8 random characters; in our example the salt is the string "12345678"), then there goes one mo $ character, followed by the actual hash. Algorithm: Actually that is a loop calling the MD5 algorithm 2000 times. Notes: [1] [2]
MD5(phpBB3)	H9123456785DAERgALpsri.D9z3ht120	Used in phpBB 3.x.x. Length: 34 characters. Description: The hash begins with the H signature, then there goes one character (most oft the number '9'), then there goes the salt (8 random characters; in our example the salt is the string "12345678"), followed by the actual hash.

MD5(Wordpress)	PB123456780BhGFYSlUqGyE6ErKErL01	Used in Wordpress. Length: 34 characters. Description: The hash begins with the P signature, then there goes one character (most often the number 'B'), then there goes the salt (8 random characters; in our example the salt is the string "12345678"), followed by the actual hash. Algorithm: Actually that is a loop calling the MD5 algorithm 8192 times. Notes: [1] [2]
MySQL	606717496665bcba	Used in the old versions of MySQL. Length: 8 bytes. Description: The hash consists of two DWORDs, each not exceeding the value of 0x7fffffff.
MySQL5	*E6CC90B878B948C35E92B003C792C46C58C4AF40	Used in the new versions of MySQL. Length: 20 bytes. Algorithm: SHA-1(SHA-1($pass)) Note: The hashes are to be loaded to the program without the asterisk that stands in the beginning of each hash.
RAdmin v2.x	5e32cceaafed5cc80866737dfb212d7f	Used in the application Remote Administrator v2.x. Length: 16 bytes. Algorithm: The password is padded with zeros to the length of 100 bytes, then that entire string is hashed with the MD5 algorithm.
MD5	c4ca4238a0b923820dcc509a6f75849b	Used in phpBB v2.x, Joomla version below 1.0.13 and many other forums and CMS. Length: 16 bytes. Algorithm: Same as the md5() function in PHP.
md5($pass.$salt)	6f04f0d75f6870858bae14ac0b6d9f73:1234	Used in WB News, Joomla version 1.0.13 and higher. Length: 16 bytes. Note: [1]
md5($salt.$pass)	f190ce9ac8445d249747cab7be43f7d5:12	Used in osCommerce, AEF, Gallery and other CMS. Length: 16 bytes. Note: [1]
md5(md5($pass))	28c8edde3d61a0411511d3b1866f0636	Used in e107, DLE, AVE, Diferior, Koobi and other CMS. Length: 16 bytes.
md5(md5($pass).$salt)	6011527690eddca23580955c216b1fd2:wQ6	Used in vBulletin, IceBB. Length: 16 bytes. Notes: [1] [3] [4]
md5(md5($salt).md5($pass))	81f87275dd805aa018df8befe09fe9f8:wH6_S	Used in IPB. Length: 16 bytes. Notes: [1] [3]
md5(md5($salt).$pass)	816a14db44578f516cbaef25bd8d8296:1234	Used in MyBB. Length: 16 bytes. Note: [1]

md5($salt.$pass.$salt)	a3bc9e11fddf4fef4deea11e33668eab:1234	Used in TBDev. Length: 16 bytes. Note: [1]
md5($salt.md5($salt.$pass))	1d715e52285e5a6b546e442792652c8a:1234	Used in DLP. Length: 16 bytes. Note: [1]
SHA-1	356a192b7913b04c54574d18c28d46e6395428ab	Used in many forums and CMS. Length: 20 bytes. Algorithm: Same as the sha1() function in PHP.
sha1(strtolower($username).$pass)	Admin:6c7ca345f63f835cb353ff15bd6c5e052ec08e7a	Used in SMF. Length: 20 bytes. Note: [1]
sha1($salt.sha1($salt.sha1($pass)))	cd37bfbf68d198d11d39a67158c0c9cddf34573b:1234	Used in Woltlab BB. Length: 20 bytes. Note: [1]
SHA-256(Unix)	$5$12345678$jBWLgeYZbSvREnuBr5s3gp13vqi...	Used in Linux and other similar OS. Length: 55 characters. Description: The hash begins with the 5 signature, then there goes the salt (up to 8 random characters; in our example the salt is the string "12345678"), then there goes one more $ character, followed by the actual hash. Algorithm: Actually that is a loop calling the SHA-256 algorithm 5000 times. Notes: [1] [2]
SHA-512(Unix)	$6$12345678$U6Yv5E1lWn6mEESzKen42o6rbEm...	Used in Linux and other similar OS. Length: 98 characters. Description: The hash begins with the 6 signature, then there goes the salt (up to 8 random characters; in our example the salt is the string "12345678"), then there goes one more $ character, followed by the actual hash. Algorithm: Actually that is a loop calling the SHA-512 algorithm 5000 times. Notes: [1] [2]

Well hope this reference table might be of help for you some time.

And out these the hashes I had to crack where “13 Chars” hashes. So it was obvious form my table that It was based on Php Crypt function.

A simple walk through of of the Php crypt function:

1) It’s is a hash algorithm which takes in a “String” and a “salt” and encrypts the hashes.

2) And by default it uses “DES” to encrypt hashes.

Consider the Ex:

```
<?php
$password = crypt('password');

?>

Hashes: laAsfestWEiq1

Here password hashes generated would be on basis of a random 2 digit salt.

Or we could provide our on salt.

<?php
$password = crypt('password','salt');

?>

Hashes: sih2hDu1acVcA
```

And the comparison password verification code would be as follows:

```
if (crypt($user_password, $password) == $password) {
   echo "Correct Password";
}
?>
```

In either of the cases the salt is appended with the Hashes, property of DES. Well as I mentioned above the security of salt cryptography is on the fact that the salt is unknown to the cracker. But here it's not. Well with this basic piece of Information, it was easy to crack hashes that I had in my hands☺.

And all the hashes were cracked easily, all I have to do was load a common passwords dictionary and add it with the constant salt, and get my work done.

Consider the given Hash/salt programs with the following cases.

Salt/Hash algorithm with Constant Salt:

```
$password = $password_input;      //user input
$salt = "salted";
$password = md5($salt.$password); //saved in db md5(saltedpassword)

Hashes: 1423de37c0c1b63c3687f8f1651ce1bf

Salt: salted
```

In this program a constant salt is used therefore the salt is not saved in the database. So our dumped hashes won't be having the salt value.

For verifying such algorithms we need to try the following things.

1) Try to create a new user using the target application.
2) Dump the data again and verify what algorithm is used using the above mentioned methods.
3) Consider the new password added was "password" md5('password')== "5f4dcc3b5aa765d61d8327deb882cf99", instead if the updated value was "1423de37c0c1b63c3687f8f1651ce1bf" that says a salt is used and is a constant one as it don't seem to be added with the final hashes.

Cracking the salt:

Now for breaking this, the only thing you could do is a bruteforce the hashes for figuring out what the salt is, for ex:

And once we know the salt append it with every password we check and crack it.

We know :

Md5('password')== "5f4dcc3b5aa765d61d8327deb882cf99"

Now question is

Md5('password' + "????WHAT????") ===
"1423de37c0c1b63c3687f8f1651ce1bf"

Note: Never use a constant salt for all hashes:

"If same constant salt is used for all hashes then it would be easy to crack all hashes"

So Point NO 2: If your PHP application is storing Sensitive values and you want to encrypt and store its salted hashes then Crypt() function is not the right option nor depending on any constant salt functions is the right choice.

Salt/Hash algorithm with Random Salt:

If random salt is used for each hash, which is necessary for application whose source is publicly available, then it would be necessary to store the salt along with the hashes. That gives it a –ve point because it's possible to extract the salt for the hashes. But + point is, that cracker need to build hash tables with each salt for cracking each hash. This makes it hard to crack multiple hashes at a time. But still possible to crack the selected hashes, consider the admin one.

Consider the example:

```
$password = $rand(5);       //user input
$salt = "salted";
$password = md5($salt.$password); //saved in db md5(saltedpassword)

Hashes: 6f04f0d75f6870858bae14ac0b6d9f73:14357 (Hash:Salt)

Salt: 14357
```

We could extract the salt, but as different hash will be having a different salt, it's impossible to crack all hashes at a stretch.

But it would be back again dependent on how good the passwords are.

At similar situations a Dictionary attack on the hashes would be the only possibility. Or else we need a better Cracking program, which provides distributed cracking process.

Rainbow tables rocks not because it has got all possible values hashes, but because "Searching" algorithm is faster.

Consider.

Rainbow tables check → searching [Fast]

Brute Force → Read a value → Append salt → Compute hashes → Compare [slow]

This property makes the attack slow even if we know the salt.

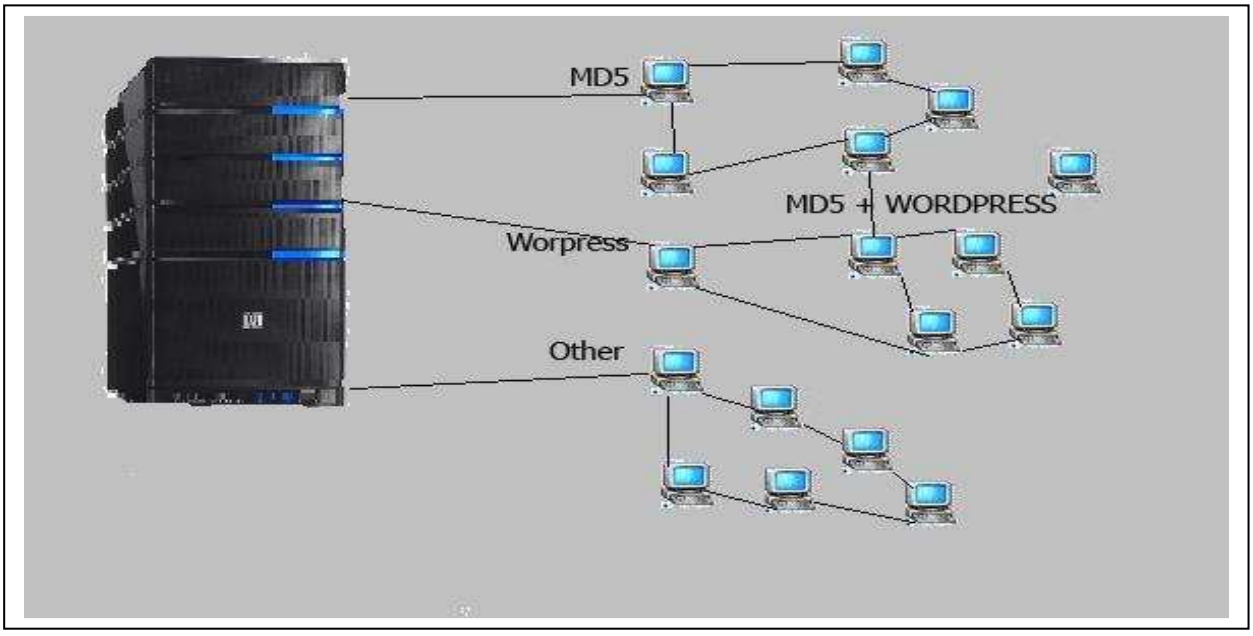
So such situation a better and free Cracking [Distributed Cracking System would be necessary]

Idea for One such Distributed Cracking System would be as follows

Tool: One such tools documentation would be.

The whole Idea of such a system comes from the concept of torrents, where if you want something you have to share something. Here if you want to crack something you will have to share your processing speed.

Architecture Of the tool should be:

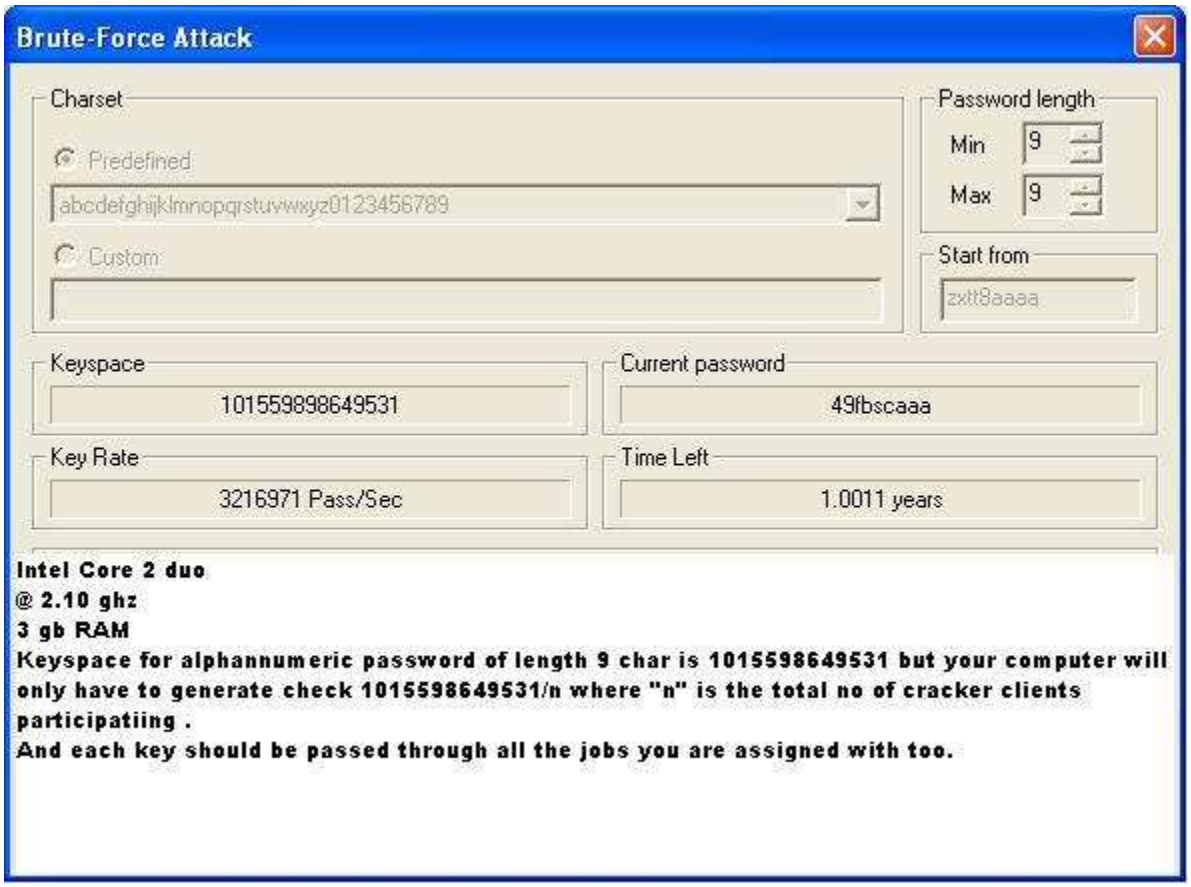


Note: Sorry for the poor Image

1) You download the Cracker tool Client
2) You have an admin hash to crack that of wordpress, you add the hash along with salt to cracker Client.
3) Cracker client sends the hash to Crack server.
4) Crack server accepts you as part of the distributed cracking Network.
5) Crack server updates you with the new set of hashes, algorithm, and permutations you have to carry out.
6) Logic is when someone is doing work for you, will have to work for them too.
7) There by your work will be carried out by many different computers.

How this speeds your cracking.

1) Your computer when in the network is assigned to generate wordlist , consider the key space for a 9 char alphanumeric password is 101559787037052 and your computer will have to generate 101559787037052/N , where "N" is the total no of cracker clients in the NW.



2) You computer will have to pass each word generated through multiple algorithms your assigned with on your multithreaded Cracker Client.

3) Once a client cracks a password it updates it to the Cracker server, and cracker server passes it to the user who requested the information.

4) So if you have 350 cracker clients working together then every body's work will be done in a day or two.

Finding an unknown Hash Algorithm:

Consider the case with such an algorithm

- Consider a situation where the hashes are multiple encrypted with different hash algorithms, for example:

```
<?php

$password = sha1('password');  //  de4he6la fe4oe6late4he6lade4he6lade4he6la

$final_password= md5($password)

Final Password Hashes: 1423de37c0c1b63c3687f8f1651ce1bf
```

  - In such kind of situations, Hashes may looks like Md5 but it's actually the md5 of sh1 hashes.

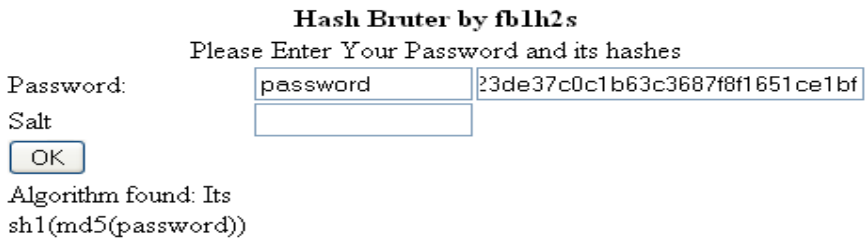
- So in such kind of situation were multiple hashing algorithm is used and algorithm is unknown, and it would be really hard to find what the hashes are.

Now you need an algorithm brute force for predicting the back end algorithm.

Algorithm_Bruter

- So I came up with this script, which takes in a known “password” and it’s “hashes” and then moves it through many different commonly used hash algorithms and tries to find a match, predicting what algorithm it used in the back end.
- For script need to be provided with a Plain Text Value and its alternate Hashes and as output you will get the algorithm used.
- You could check out the script here.
- http://www.fb1h2s.com/algorithmbruter.php
- This could be used in above mentioned situations.

Algorithm_Bruter.php



I am going through different Programming forums and taking out different, forms of multiple hashing; programmers are using and, will update it on this script. So you could find what algorithm was used.

Hope this paper was of some help for you in dealing with salted hashes.

And all greets to Garage Hackers Members.

http://www.garage4hackers.com

And shouts to all ICW, Andhra Hackers members

http://www.andhrahackers.com/

and my Brothers:-
B0Nd,Eberly,Wipu,beenu,w4ri0r,empty,neo,Rohith,Sids786,SmartKD,Tia,hg-h@xor,r5scal,Yash,Secure_IT, Atul, Vinnu and all others.

This paper was written for Null meet 21/08/

By FB1H2S

www.fb1h2s.com