2012

Breaking the Crypt

Sudeep Singh

5/21/2012

Table of Contents

Preface

Password hashing algorithm strength is generally one of the most overlooked aspects during a Security Assessment. Emphasis is often laid upon encouraging the user to choose a stronger password, however that is only one side of the coin.

As the computational power is increasing with time, one must keep the Moore's Law in mind while choosing the Cryptographic Hashing Algorithm in their implementations to store confidential data.

Hash Cracking Processes are being tweaked regularly due to which there is a need to choose stronger passwords and better hashing algorithms.

Advanced Hash Cracking

The purpose of this paper is to make the reader aware of various Hash Cracking Techniques ranging from Basic to Advanced. The intended audience for this paper is those who have a basic understanding of hash cracking and password hashing algorithms.

It also focuses on the need for implementation of stronger hashing algorithms than what are mostly used in real life scenarios. The various properties of a Cryptographic Hash Function which play an important role in determining its resistance to Hash Cracking are discussed in depth.

From a Hash Cracking perspective, this paper is generic and the techniques introduced can be extended to most of the Password Hashing Algorithms which exist.

With the help of Real Time Scenarios, the risks associated with weak passwords and weak hashing algorithms are explained.

Some Advanced Techniques of using known Password Auditing Tools such as JTR and Oclhashcat are discussed.

This paper focuses on implementations of Hashing Algorithms on CPU and GPU. There are several other platforms which are being used for Hash Cracking such as FPGA, Playstation 3, Amazon EC2 Clusters.

Cryptographic Hash Properties

Any cryptographic hashing algorithm should possess the following properties to be considered for a real life implementation:

Collision Resistance – In simple terms, it means it should not be possible to find two messages which give the same hash when a hashing algorithm is applied to them.

If H is the hashing algorithm,
m1 is the first message and m2 indicates the second message,

Then the following condition should not hold true for the hashing algorithm,

H(m1) = H(m2)

These terms have a significance which will become more evident as we delve deeper into hashing algorithms.

Preimage Resistance – The definition of this property states that it should be computationally infeasible to find an input which hashes to a specific output when the corresponding input to that output is not known.

It shall become clearer with the help of an example,

Given a hash output, y
Hashing algorithm is, H

It should not be computationally feasible to find an input, x such that, H(x) = y

Second Preimage Resistance – This relates to Collision Resistance property in a way. In this case, we have a message, m1 and its corresponding Hash, H. It should be computationally infeasible to find another message, m2 such that it hashes to the same output, H.

Hash to the Stash

To understand the consequences of choosing a weak password, I will take an example.

For a user who frequently uses Internet, there is a high probability that they are registered on at least 3 or more of the following site categories:

- Public Email services such as Gmail, AOL, Hotmail
- Social Networking sites such as Facebook, Myspace
- Professional Networking sites such as LinkedIn
- Public Forums on Internet
- Online Banking Sites
- Streaming Media Sites

Even a naive and a not so tech-savvy user these days would have registered on at least 3 or more of the above categories of sites.

There is also a high probability that the user has chosen the same password across all sites or a slightly modified version of a base password on each site.

From a security standpoint, the user is as vulnerable to a compromise as the weakest site among all of the above on which he has registered or the weakest password he has chosen.

We can consider Public Email Services, Social Networking and Professional Networking sites to be free of any security loopholes for the purpose of this demonstration.

Let's focus on Public Forums or Membership Sites. Usually they are the targets for attackers who stay up-to-date with Zero Day Exploits.

These sites are often scanned for Web Application Vulnerabilities.

Consider the following scenario.

1. You registered on a Public Forum based on SMF Open Source Software.

2. The site runs with a PHP Installation on an Apache Web Server.

3. The Admin Panel software used is Cpanel.

4. Cpanel provides phpMyAdmin application to manage the backend, MySQL Database.

5. Vulnerability in the Cpanel Software is disclosed in public.

6. Attackers start scanning the Internet looking for Forums running the vulnerable version of Cpanel using either Google Dorks or Custom Scanners.

7. Your Forum Admin or the Hosting Provider did not update the Cpanel to latest Version and the Forum was discovered during a vulnerability scan.

8. The Attacker breaks into the backend database and extracts all the hashes of members.

We are going to focus on Step 8 since cracking the Hashes is what this paper is about.

To crack a hash, first we need to identify the type of the Hash. If it is a well known Public Forum Software, then the hash types are already known unless the Web Admin has taken that extra step to modify the password hashing method.

For instance, SMF Forums have a hash type as: sha1(strtolower($user),$pass))

It makes use of the SHA1 algorithm where,
$user is the username
$pass is the password

Here the username in lowercase is used as the Salt while hashing. We will see further that how it was concluded.

At first, let us access the Login Page of the Forum just as any other registered user would do.

http://example.com/index.php?action=login

We enter a custom username and password and try to Login. If you look at the URL now, it changes to,

http://example.com/index.php?action=login2

Let us inspect the source code:

```
<form action="http://example.com/index.php?action=login2" method="post"
accept-charset="UTF-8"
onsubmit="hashLoginPassword(this, '1e74ea6f273729d0a552bba3d376b504');">

<input type="text" name="user" size="10" />
<input type="password" name="passwrd" size="10" />

<select name="cookielength">

<option value="1440">1 Day</option>
<option value="10080">1 Week</option>
<option value="43200">1 Month</option>
<option value="-1" selected="selected">Forever</option>
</select>

<input type="submit" value="Login" /><br />
<input type="hidden" name="hash_passwrd" value="" />

</form>
```

Once you enter the requested details in the form and hit the Login Button, hashLoginPassword JavaScript function is triggered and it is passed a reference to this.form and a session ID.

Let us check the hashLoginPassword Function:

```
function hashLoginPassword(doForm, cur_session_id)
{
    // Compatibility.
    if (cur_session_id == null)
        cur_session_id = smf_session_id;

    if (typeof(hex_sha1) == "undefined")
        return;
    // Are they using an email address?
    if (doForm.user.value.indexOf("@") != -1)
        return;

    // Unless the browser is Opera, the password will not save properly.
    if (typeof(window.opera) == "undefined")
        doForm.passwrd.autocomplete = "off";

    doForm.hash_passwrd.value =
hex_sha1(hex_sha1(doForm.user.value.php_to8bit().php_strtolower() +
doForm.passwrd.value.php_to8bit()) + cur_session_id);
```

```
    ........}
```

Once it validates the session ID, username and Browser, it sets the value for the hidden form field, hash_passwrd as follows:

```
doForm.hash_passwrd.value =
hex_sha1(hex_sha1(doForm.user.value.php_to8bit().php_strtolower() +
doForm.passwrd.value.php_to8bit()) + cur_session_id);
```

hex_sha1 appears to be a Custom JavaScript Function. By inspecting the main source code, we locate the SHA1.js script.

First few lines of the code of SHA1.js

```
/*
 * A JavaScript implementation of the Secure Hash Algorithm, SHA-1, as
defined
 * in FIPS PUB 180-1
 * Version 2.1 Copyright Paul Johnston 2000 - 2002.
 * Other contributors: Greg Holt, Andrew Kepert, Ydnar, Lostinet
 * Distributed under the BSD License
 * See http://pajhome.org.uk/crypt/md5 for details.
 */

function hex_sha1(s){return binb2hex(core_sha1(str2binb(s),s.length *
chrsz));}

function sha1_vm_test()
{
    return hex_sha1("abc") == "a9993e364706816aba3e25717850c26c9cd0d89d";
}
```

So, hex_sha1 is the JavaScript Implementation of the SHA1 Function.

With this review, we have confirmed how the hash is set on the client side.

Since the attacker has access to the Cpanel, they also have access to the Server Side Scripts.

At first, we will review the code of index.php:

We locate the smf_main() function,

```
function smf_main()
{
    global $modSettings, $settings, $user_info, $board, $topic, $maintenance,
$sourcedir;

...................

// Here's the monstrous $_REQUEST['action'] array - $_REQUEST['action'] =>
array($file, $function).
$actionArray = array(
        'activate' => array('Register.php', 'Activate'),
        'admin' => array('Admin.php', 'Admin'),
        'login' => array('LogInOut.php', 'Login'),
        'login2' => array('LogInOut.php', 'Login2'),
);

....

}
```

So, when the action performed is login or login2 as we observed above, the Login or Login2 function from LogInOut.php is called respectively.

We review the source code of LogInOut.php, we are interested in Login2 function which is called when the action performed is login2.

```
function Login2(){

// Load the data up!

$request = db_query("
        SELECT passwd, ID_MEMBER, ID_GROUP, lngfile, is_activated,
emailAddress, additionalGroups, memberName, passwordSalt
        FROM {$db_prefix}members
        WHERE memberName = '$_REQUEST[user]'
        LIMIT 1", __FILE__, __LINE__);

$user_settings = mysql_fetch_assoc($request);

    if (isset($_REQUEST['hash_passwrd']) && strlen($_REQUEST['hash_passwrd'])
== 40)
    {
        // Needs upgrading?
        if (strlen($user_settings['passwd']) != 40)
        {
            $context['login_error'] = $txt['login_hash_error'];
            $context['disable_login_hashing'] = true;
            return;
        }
        // Challenge passed.
        elseif ($_REQUEST['hash_passwrd'] == sha1($user_settings['passwd'] .
$sc))
            $sha_passwd = $user_settings['passwd'];

        $sha_passwd = sha1(strtolower($user_settings['memberName']) .
un_htmlspecialchars(stripslashes($_REQUEST['passwrd'])));

....

}
```

After reviewing the source code we can see that the Hash of the Password is as following:

**$sha_passwd = sha1(strtolower($user_settings['memberName']).
un_htmlspecialchars(stripslashes($_REQUEST['passwrd'])));**

Or

sha1(strtolower($user),un_htmlspecialchars(stripslashes($pass)));

Now, we know the hashing algorithm used to hash the password.

This is the first step of our attack.

The next step is to locate the hashes. This is fairly easy and straightforward. Usually you can locate a table with the name containing, *members*, *user* and so on.

In the case of SMF, it is called smf_members

You can view this table using phpMyAdmin by clicking on the table name.

There are many columns in here, which we do not need for the purpose of demonstration.

So, by running the following SQL Query, we will extract the hash and username.

```
SELECT `memberName` , `passwd` , `emailAddress` FROM `smf_members` WHERE 1
```

After exporting the extracted hashes, we arrange them in the following format,

hash:username

Please note that username is the salt here.

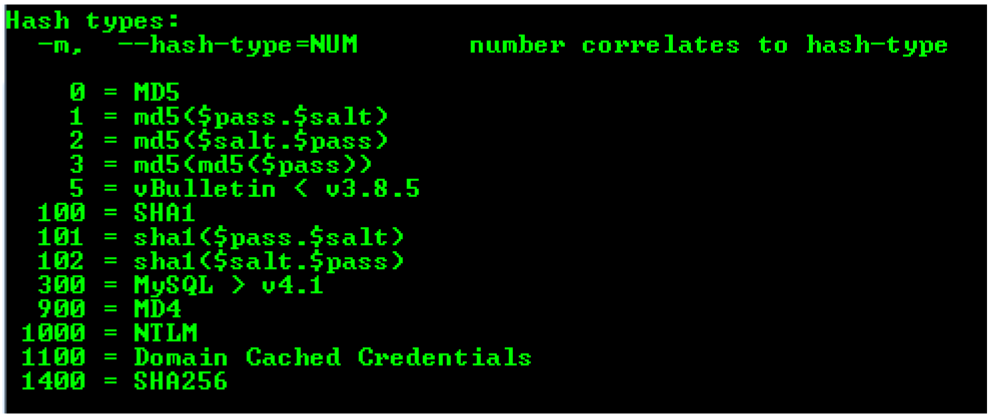
The next step is to identify the software to be used for cracking the hashes.

Since SMF is a well known Forum Software, this hashing algorithm format has already been implemented in JTR and Oclhashcat.

For instance, to crack them using Oclhashcat, this is how we go about it:

Type in, oclhashcat32.exe -h

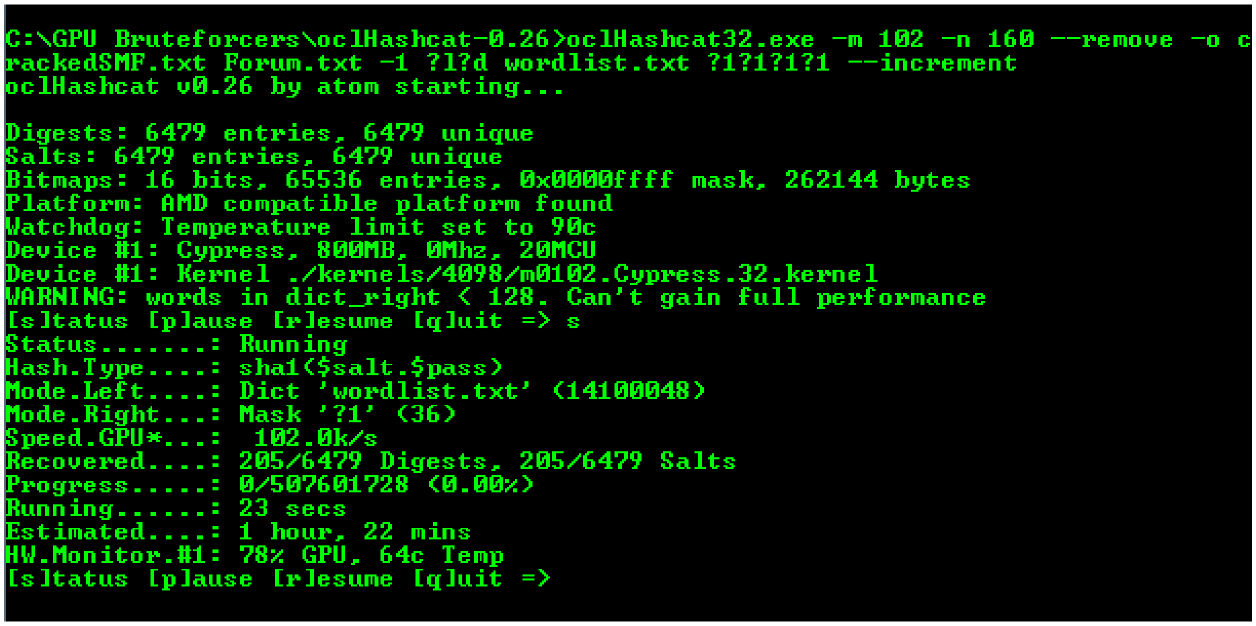
This will list out all the options as shown below:



```
Hash types:
  -m,  --hash-type=NUM       number correlates to hash-type

    0 = MD5
    1 = md5($pass.$salt)
    2 = md5($salt.$pass)
    3 = md5(md5($pass))
    5 = vBulletin < v3.8.5
  100 = SHA1
  101 = sha1($pass.$salt)
  102 = sha1($salt.$pass)
  300 = MySQL > v4.1
  900 = MD4
 1000 = NTLM
 1100 = Domain Cached Credentials
 1400 = SHA256
```

102 - sha1($salt,$pass) -> This matches our SMF Forum hash format.

This is how the cracking process looks like in action.



```
C:\GPU Bruteforcers\oclHashcat-0.26>oclHashcat32.exe -m 102 -n 160 --remove -o c
rackedSMF.txt Forum.txt -1 ?l?d wordlist.txt ?1?1?1?1 --increment
oclHashcat v0.26 by atom starting...

Digests: 6479 entries, 6479 unique
Salts: 6479 entries, 6479 unique
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes
Platform: AMD compatible platform found
Watchdog: Temperature limit set to 90c
Device #1: Cypress, 800MB, 0Mhz, 20MCU
Device #1: Kernel ./kernels/4098/m0102.Cypress.32.kernel
WARNING: words in dict_right < 128. Can't gain full performance
[s]tatus [p]ause [r]esume [q]uit => s
Status.......: Running
Hash.Type....: sha1($salt.$pass)
Mode.Left....: Dict 'wordlist.txt' (14100048)
Mode.Right...: Mask '?1' (36)
Speed.GPU*...:  102.0k/s
Recovered....: 205/6479 Digests, 205/6479 Salts
Progress.....: 0/507601728 (0.00%)
Running......: 23 secs
Estimated....: 1 hour, 22 mins
HW.Monitor.#1: 78% GPU, 64c Temp
[s]tatus [p]ause [r]esume [q]uit =>
```

With a fairly decent wordlist and a custom mask attack, within 23 seconds approximately 205 passwords from 6479 hashes are recovered.

It should be easy to comprehend the result of this from a security perspective.

In the next few sections, we will look more in depth about the options available to us in these tools.

Oclhashcat – An insight

This section is not going to cover the basics of hash cracking and how you go about it.

Please note that most of the passwords I use as an example in this article are modified versions of the word, **defcon**. However, this concept can be extended to just about any dictionary word since I have covered most of the mangling rules.

Hash cracking tools like JTR have existed since a long time. As the hardware speeds have scaled up over the years, so have the hash cracking speeds. It provides us better opportunities to crack more hashes than we could have on the old Pentium Processors.

Cracking Hashes on GPU: Nvidia's CUDA and ATI's OpenCL gave developers a chance to port the hash cracking algorithms to GPUs. GPU's have a high parallel processing power and this is a big advantage over CPU's which do serial processing instead.

CPU's are good at performing complex instructions quickly and GPU on the other hand can perform many easy instructions quickly.

To discuss more in depth, I'll start with using oclhashcat as an example. Oclhashcat is developed by atom (hashcat team) and it's compatible with both Nvidia's CUDA based GPU's and ATI's Open CL GPUs.

You can get more details here:

http://hashcat.net/oclhashcat-plus/

Here, I present a few ways in which we can use this tool more effectively to get better success rates at cracking hashes.

Starting with the basic forms of attack:

```
./oclhashcat64.exe -m 0 -n 160 --remove -o foundMD5.txt md5.txt dict1.txt
dict2.txt
```

m - Specifies the type of hash we are attacking. Just type in, oclhashcat --help and scroll down to see all the hash types.

-m 0 is for Raw MD5 hashes.

-n 160 - This is for workload tuning of GPU and you might have to tweak it depending on your GPU. The higher the load on your GPU, the higher will be the operating temperature as well, something that you will have to keep in mind.

--remove - This option is quite useful. Oclhashcat will remove the hashes it has cracked from the hashlist so that it doesn't attempt to crack them again during another session.

-o by default, oclhashcat will display the cracked hashes on the console. But, since we want to save our results, it's a good idea to use this option and redirect the output to a text file. In this case, foundMD5.txt

md5.txt - the hash file. At present, oclhashcat expects the hash list to have only hashes in it and no usernames. Unlike the format which JTR accepts, here we need filter out any usernames from the list.

dict1.txt and dict2.txt - oclhashcat works on the concept of the left mask and right mask. It breaks every word into a left and right part and allows us to define how these parts are controlled.

This option is really useful, as we will see further in this article, how efficiently we can use this option.

It's compulsory to provide both a left mask and a right mask. You cannot give one and omit the other. In this case, dict1.txt is the left mask and dict2.txt is the right mask.

Ok, so that was just the basic. Let's get to more effective methods.

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt md5.txt dict1.txt ?d?d?d?d
```

Here the left mask is a dictionary however the right mask is a predefined charset.

These charsets are predefined by oclhashcat for digits, lowercase, uppercase and special chars.

Later we will see there are many more charsets defined as well for the more obscure passwords using characters from Unicode Charset.

This form of attack works well against passwords where a user has padded a sequence of chars at the end.

The above command will try all 4 digits padded to the password. But to have a higher success rate and to save our time, we have the option –increment

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt md5.txt dict1.txt ?d?d?d?d --
increment
```

This is a time saving option. It tries the following 4 right masks.

```
?d
?d?d
?d?d?d
?d?d?d?d
```

Makes life much easier.

We have been padding only digits at the end of our passwords. How do we attack patterns like, **defcon@123**. The good thing about masks is they give you complete control over every character position.

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt md5.txt dict1.txt ?s?d?d?d
```

The above command will pad patterns like: @123, #276, $125 to the end of every word in the dictionary, dict1.txt.

But, there could be many more combinations of special chars and digits as well. For instance, **defcon@1!2#**

We can make our attacks more dynamic by binding the predefined charsets to masks.

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt -1 ?d?s md5.txt dict1.txt
?1?1?1?1
```

The above command will bind the two predefined charsets, ?d and ?s to mask custom charset, ?1.

So, our right mask now becomes, ?1?1?1?1 providing us a better flexibility.

But what about the following patterns?

defcon@34

defcon#$!
defcon()
return();

We can use the --increment option combining it with the custom charset mask

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt -1 ?d?s md5.txt dict1.txt
?1?1?1?1 --increment
```

So, far I have covered only passwords which have padded a sequence of characters to them. But, we have not touched our left mask, which is the dictionary at all.

Time to apply some modification rules to it as well.

in oclhashcat, the left mask and right mask can be controlled using -j and -k options.

-j allows the control over left mask and -k over right mask.

The correspondence between -j and -k to left and right mask respectively can be remembered easily by looking at the position of characters j and k on the keyboard. The left one corresponds to left mask and right one to right mask. It's not a big deal to remember, but just in case, it might help someone.

This time, we will attack following pattern:

Defcon@$a5

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt -1 ?l?d?s -j U md5.txt
dict1.txt ?1?1?1?1 --increment
```

The new option here is, -j U

U will convert the first char of left mask to uppercase for every word in the dictionary. And since in the padding we have, @$a5, I have binded the charsets, ?l?d?s to -1

More complex password patterns could remove a few chars from the dictionary and do padding instead.

For instance, **defc@3$d**

Here the letters, o and n are stripped off from the end of "defcon" and padded with @3$d instead. To attack such a pattern, we will do the same with our dictionary. This time, we will strip off the last 2 chars from every word of dictionary in the left mask and upper case the first one followed by the password padding.

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt -1 ?l?d?s -j ]] dict1.txt
?1?1?1?1 --increment
```

-j]] is the new option. The left square bracket, ']', will strip off one char from the end of the left mask and then apply the other masks as mentioned above.

So, far we have been using dictionary as one of our masks. If we are targeting a fast hashing algorithm like, raw md5 or raw sha1, we can try a pure bruteforce attack as well.

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt -1 ?l?d?s ?1?1?1?1 ?1?1?1?1 -
-increment
```

In this case, the left mask remains constant at ?1?1?1?1 and right mask is incremented from ?1 to ?1?1?1?1.

It's important to understand that, the load on GPU depends on the amount of different combinations in the mask.

For instance, if you are running an attack with the right mask set to ?1?1 and getting a lower hashing speed. That doesn't mean the performance of your hardware is down.

It just means, you are not utilizing the full power of your GPU. The essence of this is, you need to keep the GPU busy with as many combinations as possible to get the best out of it.

I have covered different ways of using the left mask and right mask. Most of the techniques apply the same to both left and right mask with a few exceptions as listed below:

-k instead of -j. As mentioned above, to control the words in the right mask, we need to use -k instead of -j.

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt md5.txt -1 ?l?d?s -k ]]]]
?1?1?1?1 dict1.txt
```

This will strip the last 4 characters from the second dictionary and pad them to the left mask.

--increment option - While we can use the --increment option with the right mask to save time. The same option cannot be used with the left mask.

So, the following command won't work.

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt md5.txt -1 ?d?u?l?s ?1?1?1?1
--increment dict1.txt
```

Date Pattern:

An interesting example. Some people use their passwords in the form of dates and believe me, these are hard to bruteforce. Since, they have special characters in between digits and it's not possible to find a word like this in dictionary.

Patterns such as:

```
08/10/83
10-04-12
03.07.09
```

Masks to the rescue!

Once again, masks can make this task really easy.

```
oclhashcat -m 0 -n 160 --remove -o foundMD5.txt md5.txt ?d?d?s?d ?d?s?d?d
```

This covers a few advanced techniques of using oclhashcat.

Please Note: The latest version of oclhashcat-plus now targets both slow and fast hash types. Almost all the techniques I have described above can be applied to oclhashcat-plus too with the exception of a few options.

Some changes:

1. The options –j and –k are not present in the recent version.
2. The option, --increment is also not present.
3. Now each attack mode (dictionary/hybrid mask, combination, bruteforce) needs to be specified at the time of attack.
4. Oclhashcat-plus also allows the use of word mangling rules.
5. There is no longer a concept of left and right mask. You need to use one mask.

The techniques described above are not exhaustive and you are not limited to only the above combinations.

From an advanced hash cracking standpoint, we have only scratched the surface. There's a lot more to it

The need for Stronger Hashes

Once the site is compromised and your hashes are leaked to the public, the single line of defense for the user is either of the following:

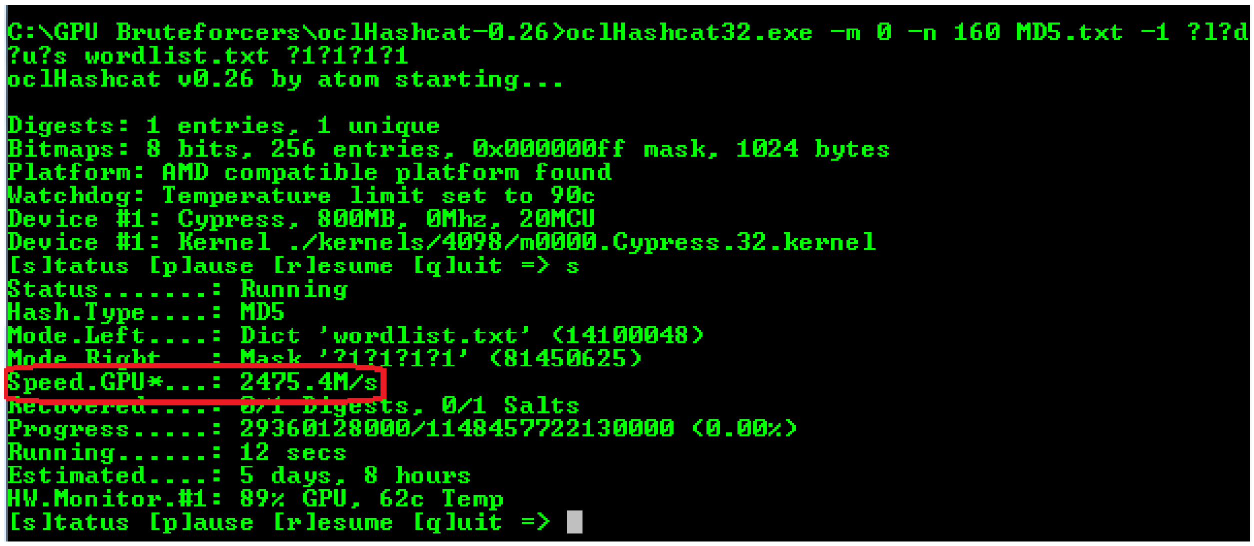
1. `How strong is your password?`
2. `How strong is the Cryptographic Hashing Algorithm used?`

Point 1 is your responsibility and it is something that you can control. Point 2 however is not in your hands and it is the responsibility of the Site Administrator.

When you sign up on a site, you are often presented with a message asking you to choose a strong password and it should conform to certain rules.

It is important to understand that both the above points are of equal priority. If your password is considerably strong, it can still be broken if the password hashing algorithm used is a Saltless MD5 hash.

With a fairly recent GPU one can generate candidate hashes during a dictionary attack against Saltless MD5 hashes at close to a couple of thousand million hashes per second as shown below:



```
C:\GPU Bruteforcers\oclHashcat-0.26>oclHashcat32.exe -m 0 -n 160 MD5.txt -1 ?l?d
?u?s wordlist.txt ?1?1?1?1
oclHashcat v0.26 by atom starting...

Digests: 1 entries, 1 unique
Bitmaps: 8 bits, 256 entries, 0x000000ff mask, 1024 bytes
Platform: AMD compatible platform found
Watchdog: Temperature limit set to 90c
Device #1: Cypress, 800MB, 0Mhz, 20MCU
Device #1: Kernel ./kernels/4098/m0000.Cypress.32.kernel
[s]tatus [p]ause [r]esume [q]uit => s
Status.......: Running
Hash.Type....: MD5
Mode.Left....: Dict 'wordlist.txt' (14100048)
Mode.Right...: Mask '?1?1?1?1' (81450625)
Speed.GPU*...: 2475.4M/s
Recovered....: 0/1 Digests, 0/1 Salts
Progress.....: 29360128000/1148457722130000 (0.00%)
Running......: 12 secs
Estimated....: 5 days, 8 hours
HW.Monitor.#1: 89% GPU, 62c Temp
[s]tatus [p]ause [r]esume [q]uit =>
```

As can be seen above, it is possible to exhaust a Large Wordlist containing above 14 Million Unique Words combined with almost the Entire ASCII Key Space (all lowercase, uppercase, digits, special characters) in 5 days and 8 hours.

Someone with a considerable computational power can crack these hashes with a Probability close to 90% or above.

Fast vs Slow Hashes

The need for a stronger Cryptographic Hashing Algorithm was realized several years back. Hashing algorithms such as MD5, SHA-1 were no longer suitable for storing password hashes due to the amount of computational power available for cracking hashes.

Fast hashing Algorithms are those which involve minimal computational complexity and time to calculate the hash of a given password. Algorithms such as MD5, SHA-1 are considered fast and they are easier to bruteforce since we can generate several million candidates in a second.

Slow Hashing Algorithms are based on the above Cryptographic Primitives. They are stronger because they involve several thousands of iterations of the basic cryptographic algorithms.

A suitable example to quote at this point would be the MD5 (Unix) variant of the Saltless MD5 Hash.

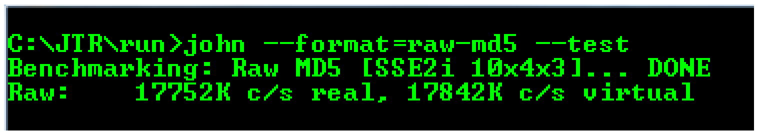
It uses two methods which increase the computational complexity:

It uses several thousand iterations of MD5 Hashing with Salt which increases the time taken to calculate the hash.
It adds a salt which makes it resistant to pre computational attacks such as Rainbow Tables.

On Unix Based Operating Systems, these hashes are better known as **crypt(3)**, the C function used to implement the algorithm.

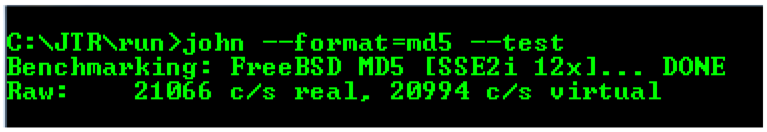
A comparison of the benchmark performed by John The Ripper on Raw MD5 and MD5 Crypt:

Fast Hashing Algorithm:



```
C:\JTR\run>john --format=raw-md5 --test
Benchmarking: Raw MD5 [SSE2i 10x4x3]... DONE
Raw:    17752K c/s real, 17842K c/s virtual
```

Slow Hashing Algorithm:



```
C:\JTR\run>john --format=md5 --test
Benchmarking: FreeBSD MD5 [SSE2i 12x]... DONE
Raw:    21066 c/s real, 20994 c/s virtual
```

The Saltless Hashing algorithm is faster by orders of magnitude than its highly iterated and salted counterpart.

How much Salt?

We know that the addition of a salt to a password during the hashing phase will prevent precomputational attacks such as Rainbow Table attacks to succeed. However, there is still something worth considering while using a Salt.

Salts should not be short enough that it is possible to precompute hashes of all possible combinations of a password and salt.

For instance, the traditional UNIX hashes used a 12 bit salt, which results in a total of 4096 combinations. Attackers determined to break the hashes can trade a lot of disk space to store Rainbow Tables corresponding to all the possible salt combinations.

Another point worth mentioning is that in the password hashing scheme where the salt is not random but as common as a username can result in issues. For instance, if the same user is registered on multiple sites with same username and each of these sites make use of the same hashing algorithm. A compromise of password on one is equivalent to comprise of password on all of them and it reduces the amount of computational power required to crack that hash.

Often people have an incorrect notion and consider that hashing algorithms such as MD5 (UNIX), DES which store the salt in the hash transparently are insecure.

DES Hash - JqVtCFFe0OM3E – Salt is the first 2 characters, Jq
MD5 (UNIX) Hash - $1$O5tDDRb7$I2pzJoM88vkgyDR3Owp35/ - Salt is the characters stored between second and third $ symbol. In this case, the salt is O5tDDRb7

However, considering the scenario where an attacker has got access to the Backend already, even if the salt is stored separately in the same table, they are still accessible.

From a Computational Complexity point of view, let us take VBulletin Forum Hashes as an example.

Versions of Vbulletin prior to 3.8.5 used a 3 Character Salt, for instance:

47dbb53a4dcbd13039e7a8dcfd31a782:yZG

yZG are the salt characters. If you are interested to know how this salt is calculated, you can view the source code of /includes/class_dm_user.php

```
/**
    * Generates a new user salt string
    *
    * @param    integer    (Optional) the length of the salt string to
generate
    *
    * @return    string
    */
    function fetch_user_salt($length = SALT_LENGTH){
        $salt = '';

        for ($i = 0; $i < $length; $i++)
        {
            $salt .= chr(rand(32, 126));
        }

        return $salt;
    }
```

The value of **SALT_LENGTH** parameter is set to 3 for versions of VBulletin prior to 3.8.5

For versions of VBulletin newer than 4.0, the salt length was increased to 30. The parameter, SALT_LENGTH was set to 30 by default.

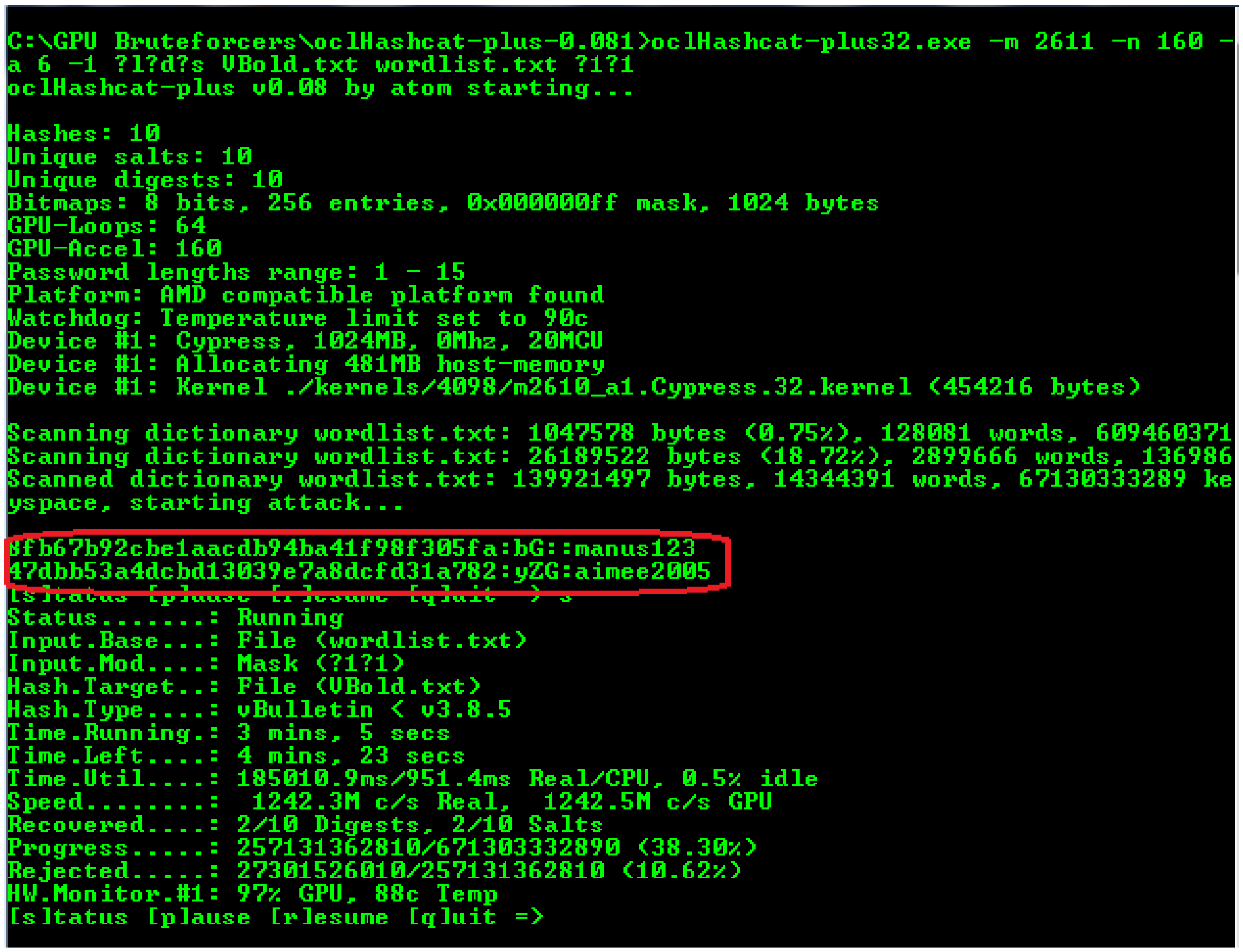
cb45353c18eb5ff2c0277fce6cd490c3:GnLx7ZcA/$$8IBXxzr^kY`#h2u{{3R

Let us compare the Hashing Speed for both the Hash Types.

I have taken 10 hashes of each type, older versions of VBulletin with a 3 character salt and newer versions of VBulletin with a 30 character salt.

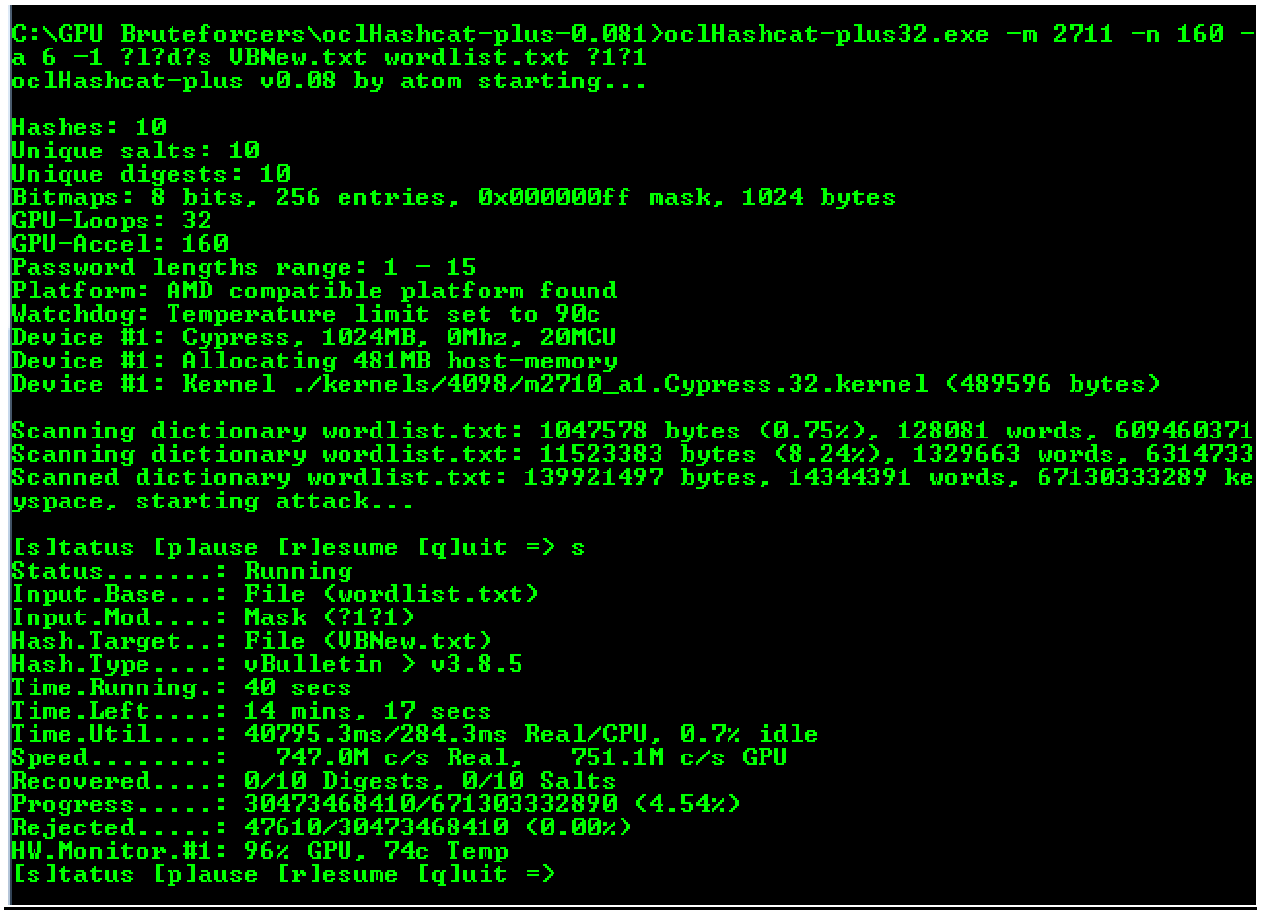
Let us compare the computational complexity of these:

VBulletin Hashes with 3 Char Salt



```
C:\GPU Bruteforcers\oclHashcat-plus-0.081>oclHashcat-plus32.exe -m 2611 -n 160 -
a 6 -1 ?l?d?s VBold.txt wordlist.txt ?1?1
oclHashcat-plus v0.08 by atom starting...

Hashes: 10
Unique salts: 10
Unique digests: 10
Bitmaps: 8 bits, 256 entries, 0x000000ff mask, 1024 bytes
GPU-Loops: 64
GPU-Accel: 160
Password lengths range: 1 - 15
Platform: AMD compatible platform found
Watchdog: Temperature limit set to 90c
Device #1: Cypress, 1024MB, 0Mhz, 20MCU
Device #1: Allocating 481MB host-memory
Device #1: Kernel ./kernels/4098/m2610_a1.Cypress.32.kernel (454216 bytes)

Scanning dictionary wordlist.txt: 1047578 bytes (0.75%), 128081 words, 609460371
Scanning dictionary wordlist.txt: 26189522 bytes (18.72%), 2899666 words, 136986
Scanned dictionary wordlist.txt: 139921497 bytes, 14344391 words, 67130333289 ke
yspace, starting attack...

8fb67b92cbe1aacdb94ba41f98f305fa:bG::manus123
47dbb53a4dcbd13039e7a8dcfd31a782:yZG:aimee2005
[s]tatus [p]ause [r]esume [q]uit => s
Status.......: Running
Input.Base...: File (wordlist.txt)
Input.Mod....: Mask (?1?1)
Hash.Target..: File (VBold.txt)
Hash.Type....: vBulletin < v3.8.5
Time.Running.: 3 mins, 5 secs
Time.Left....: 4 mins, 23 secs
Time.Util....: 185010.9ms/951.4ms Real/CPU, 0.5% idle
Speed........:  1242.3M c/s Real,  1242.5M c/s GPU
Recovered....: 2/10 Digests, 2/10 Salts
Progress.....: 257131362810/671303332890 (38.30%)
Rejected.....: 27301526010/257131362810 (10.62%)
HW.Monitor.#1: 97% GPU, 88c Temp
[s]tatus [p]ause [r]esume [q]uit =>
```

VBulletin Hashes with a 30 Char Salt



```
C:\GPU Bruteforcers\oclHashcat-plus-0.081>oclHashcat-plus32.exe -m 2711 -n 160 -
a 6 -1 ?l?d?s VBNew.txt wordlist.txt ?1?1
oclHashcat-plus v0.08 by atom starting...

Hashes: 10
Unique salts: 10
Unique digests: 10
Bitmaps: 8 bits, 256 entries, 0x000000ff mask, 1024 bytes
GPU-Loops: 32
GPU-Accel: 160
Password lengths range: 1 - 15
Platform: AMD compatible platform found
Watchdog: Temperature limit set to 90c
Device #1: Cypress, 1024MB, 0Mhz, 20MCU
Device #1: Allocating 481MB host-memory
Device #1: Kernel ./kernels/4098/m2710_a1.Cypress.32.kernel (489596 bytes)

Scanning dictionary wordlist.txt: 1047578 bytes (0.75%), 128081 words, 609460371
Scanning dictionary wordlist.txt: 11523383 bytes (8.24%), 1329663 words, 6314733
Scanned dictionary wordlist.txt: 139921497 bytes, 14344391 words, 67130333289 ke
yspace, starting attack...

[s]tatus [p]ause [r]esume [q]uit => s
Status.......: Running
Input.Base...: File (wordlist.txt)
Input.Mod....: Mask (?1?1)
Hash.Target..: File (VBNew.txt)
Hash.Type....: vBulletin > v3.8.5
Time.Running.: 40 secs
Time.Left....: 14 mins, 17 secs
Time.Util....: 40795.3ms/284.3ms Real/CPU, 0.7% idle
Speed........:   747.0M c/s Real,   751.1M c/s GPU
Recovered....: 0/10 Digests, 0/10 Salts
Progress.....: 30473468410/671303332890 (4.54%)
Rejected.....: 47610/30473468410 (0.00%)
HW.Monitor.#1: 96% GPU, 74c Temp
[s]tatus [p]ause [r]esume [q]uit =>
```

Hashing Speed for VBulletin Hashes with 3 Character Salt = 1242.3M c/s
Hashing Speed for VBulletin Hashes with 30 Character Salt = 747.0M c/s

In both the cases the same attack was run on the VBulletin hashes and it should be evident that the Hashing Speed in the case of a shorter salt is faster than the hashing speed in the case of a longer salt.

How Many Iterations?

As discussed above, the slower hashing algorithms like MD5 (Unix) are based on the faster cryptographic primitives. They make use of multiple iterations, often thousands of them which makes the process of generating candidate hashes during a bruteforce process slow down.

However, the number of iterations should not be so high that the computational feasibility takes a hit. Reason being, if these hashing algorithms are being applied to a web application, every time a user logs in, the hash is computed for the entered password.

Let us compare Two Slow Cryptographic Hashing Algorithms and understand the way multiple iterations are implemented in them.

MD5 (UNIX) makes use of a fixed number of iterations as a result of which the computational cost is reduced. You can view the code for crypt-md5 on freebsd.org here:

http://www.freebsd.org/cgi/cvsweb.cgi/src/lib/libcrypt/crypt-md5.c?rev=HEAD

Specifically the following section of code is used to slow down the process of hashing by using iterations:

```
MD5Final(final, &ctx);

/*
 * and now, just to make sure things don't run too fast
 * On a 60 Mhz Pentium this takes 34 msec, so you would
 * need 30 seconds to build a 1000 entry dictionary...
 */
for(i = 0; i < 1000; i++) {
    MD5Init(&ctx1);
    if(i & 1)
        MD5Update(&ctx1, (const u_char *)pw, strlen(pw));
    else
        MD5Update(&ctx1, (const u_char *)final, MD5_SIZE);

    if(i % 3)
        MD5Update(&ctx1, (const u_char *)sp, (u_int)sl);

    if(i % 7)
        MD5Update(&ctx1, (const u_char *)pw, strlen(pw));

    if(i & 1)
        MD5Update(&ctx1, (const u_char *)final, MD5_SIZE);
    else
        MD5Update(&ctx1, (const u_char *)pw, strlen(pw));
    MD5Final(final, &ctx1);
}

p = passwd + strlen(passwd);
```

```
    return (passwd);
```

Even though the inputs to MD5Update() function vary based on the iteration counter, the number of iterations are fixed.

Now, let us look at another Slow Cryptographic Hashing Algorithm, bcrypt (Blowfish).

As is the case with other Slow Hashing Algorithms, it also makes use of several iterations of basic cryptographic primitives along with a high entropy salt.

However, what sets the Blowfish Algorithm apart from others is the use of variable iterations which can be configured by an Administrator.

$2a$05$kbASUNNrr9/tOIaQPTM67eLU8kHdyWM3GJIvWe3IrYFwBatcxwBJO

If we look at the specifications of crypt function (CRYPT_BLOWFISH = 1) of PHP here:

http://www.php.net/crypt

It states that the salt must begin with, "$2a$" followed by the base 2 logarithm of the iteration count.

In our case, it is 5 which means the total number of iterations are 2^5 = 32.

This algorithm was designed keeping in mind the Moore's Law. As the computational power increases with time, the feasibility to crack these hashes will increase as well. Hence, the variable iterations can be altered easily to increase the computational complexity.

Now that we have taken an in depth look at various Cryptographic Hashes Properties, it is time to look at another Hash Cracking Tool and see how effectively we can use it.

John The Ripper (JTR) – Tweak That Attack!

You are all familiar with JTR if you've been cracking hashes for quite some time. I wanted to draw attention to certain features of JTR which will help you gain a better grasp at how it works and how it can be used more efficiently.

Important Files:

There are several important files which must be kept secure:

john.pot -> The well known pot file of jtr. Every time, jtr cracks a hash, it records the result in hash:pass format in a file called john.pot. It helps JTR to skip these hashes in future cracking processes should they reappear in the hash list.

We all know this much about john.pot. There are ways in which john.pot can be used to customize our cracking process. It can be parsed to do character frequency analysis of already cracked hashes and prepare custom charsets. More on that later.

john.rec -> This is the default session file of jtr. When you run jtr without explicitly specifying a session name, it creates a file called john.rec for the current session. This file has the extension ".rec". So, any file in your $john/run directory with extension ".rec" is the session file. They hold the command line parameters which we passed to JTR in that session. It helps JTR in resuming the cracking process the next time.

The name of this file is the same as the name given to the session at command line.

```
./john -w:wordlist.txt --format=raw-MD5 --session=cracking1 hashes.txt
```

Once we run this command, 2 new files called cracking1.rec and cracking1.log are created in JTR's home directory.

It's also a good reference for us. For instance, if we have 10 different sessions created called "cracking1" to "cracking10". Now if after 3 days, you resume any of these cracking sessions, you will most likely not remember what these sessions do. So, quickly open the .rec file corresponding to your cracking session and read the CLI parameters.

john.log -> This file holds the cracking process details, the sequence of chars JTR has already tried, are stored in this file. Its size depends on the duration for which the cracking process was run. Again, this is another file which needs to be kept secure.

You can have multiple .rec and .log files, since they are specific to the session. Their names are used by JTR to correlate them.

charset files -> What makes JTR so efficient is its ability to perform character frequency analysis on a list of words based on the frequency of occurrence of characters in a specific position. Based on this, JTR can generate words and increase the probability of cracking hashes. The list of words used by JTR to perform character frequency analysis has to be provided by us and is usually the hashes cracked so far during a session.

Charset files have ".chr" extension. By default, JTR provides us several charset files like all.chr, digits.chr, alnum.chr, lanman.chr. You cannot read the contents of these files by opening them with a text editor. They are used by JTR. However, we are given the option to create our own custom charset files.

How does it create this custom charset file?

By default, JTR parses the john.pot file and looks up all the passwords. It then performs character frequency calculation on these passwords and generates the custom charset file.

However, we would like to use our own file of cracked hashes. Here's a quick way of doing this.

Let's say we are cracking the hashfile, hashes.txt which contains a list of MD5 hashes. After cracking around 100 hashes, I decide to speed up the cracking process by fingerprinting the passwords already cracked so far.

So, I list down the cracked hashes in the console.

```
./john -show --format=raw-MD5 hashes.txt
```

It gives a list of all the cracked hashes in user:pass format.

```
./john -show --format=raw-MD5 hashes.txt > parseMe.txt
```

This will redirect the above output to parseMe.txt

Now, we need to filter out all the usernames and keep only the passwords in the format, :pass as opposed to user:pass.

It can be done quickly using Perl,

A sample script:

```
#!/usr/bin/perl

use strict;
use warnings;

$input=$ARGV[0];
$output=$ARGV[1];

chomp $output;

my ($user,$pass);

open(INPUT,"<$input") || die (\"Couldn't open the file with error $!n");
open(OUTPUT,">$output");

while(<INPUT>)
{
   chomp $_;
   ($user,$pass)=split /:/,$_;
   print OUTPUT ":".$pass."\n";
}

close(INPUT);
close(OUTPUT);
```

It will read each entry from the passfile in user:pass format and convert it to :pass format storing the result in an output file.

First thing, take a backup of the john.pot file somewhere outside the JTR's directory and delete it from inside run folder. Reason being, we are going to have a new john.pot file after this perl script is run and this will be used to prepare our custom charset.

Let's say, this script is called, Charset.pl. You need to call this script now as:

perl charset.pl parseMe.txt john.pot

Now, we have our specific patterns in john.pot file.

```
./john --make-charset=specific.chr
```

JTR will quickly parse the john.pot file to prepare our specific.chr file. We need to add an entry for this custom charset file inside john.conf which will enable JTR to use it.

Create a new section in JTR's john.conf as follows:

```
[Incremental:Specific]
FILE=$JOHN/specific.chr
MinLen= 1
MaxLen= 8
CharCount=
```

Min and Max lengths are of course going to depend on your requirements.

Now, to use our custom charset file, specific.chr in our next cracking session:

```
./john -i:Specific --format=raw-MD5 --session=cracking2 hashes.txt
```

Notice, the name "Specific" which I passed on the command line. This needs to be the same as what was defined in the john.conf file's incremental section.

john.conf -> Among all the files, this will be the most often visited file during the cracking process. All the tweaks need to be made in this file to increase the efficiency of our cracking process. A good understanding of the john.conf file helps in having a better command on JTR.

Sections

The JTR configuration file organizes information into separate sections. Each section has a unique function. Section Names are enclosed within Square Brackets.

There are different types of sections.

Rules sections: Any rule section name must have "List.Rules:" prefixed to it. The default rules section available in JTR are:

[List.Rules:Single] - This consists of all the rules which are applied to words during a single mode crack.

[List.Rules:Wordlist] - This consists of all the rules applied to a wordlist or dictionary when we pass the --rules parameter on command line.

[Incremental:Attack_Name] - Incremental sections are specific to the Incremental or Blind Bruteforce attacks performed in JTR. These are usually the last resort to hit those hard-to-find combinations of characters. Once we have exhausted other forms of attack, we go for this one. They run, until JTR has cracked all the hashes in the list or has finished trying all combinations generated from the charset files.

Here Attack_Name represents the name of the type of Incremental attack we are performing. It could be anything we want. What is important is the values under each Incremental Section. By default, JTR comes with 4 different sections of Incremental attack, All, Digits, Alpha, LanMan.

Each section requires following parameters:

FILE= Path to the charset file used by JTR to generate the words. This path is relative to JTR's home directory, $JOHN.
MinLen = Min length of the word generated by JTR using the charset
MaxLen = Maximum Length of the words generated by JTR with above charset
Charcount = The length of keyspace JTR uses to generate the characters. If we are targetting all the digits, lowercase and upper case characters of ASCII then, it will be 9+26+26 = 61.

There are other interesting sections as well like for external mode which we will cover in other series.

What combinations is JTR generating?

Once you have tweaked your cracking session by creating custom charset files, or by creating new rules inside john.conf file, it would be a good if could check what words JTR is generating using them. This will help us in confirming whether we got the desired results from our tweaking or not.

Let's say, I create a new rule in john.conf which will append 3 digits to the end of every word. Now, I want to run JTR such that it generates all these words by applying my custom rules and display them on the console.

This can be done easily using the --stdout option.

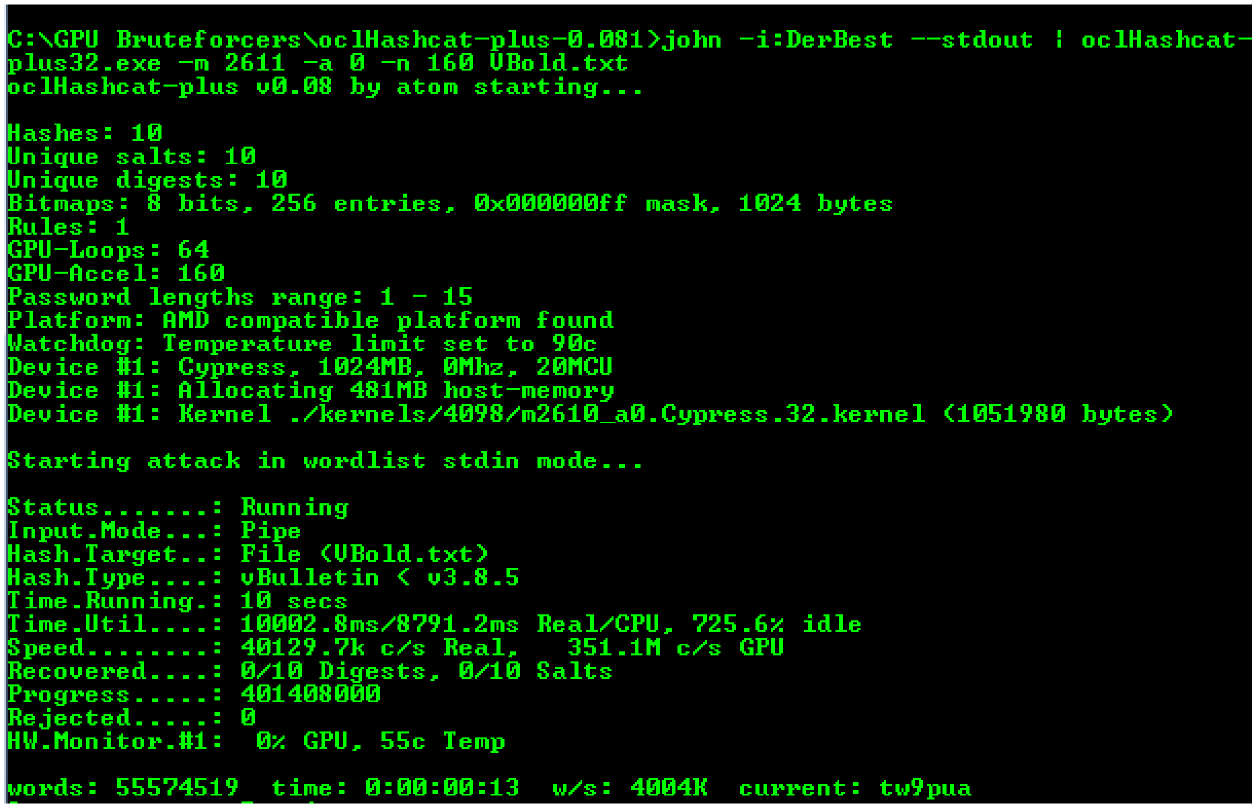
```
./john -w:wordlist.txt --rules=Add3Num --stdout
```

JTR will quickly populate the console with the list of generated word combinations.

A nice way to use --stdout option is to pipe the output of JTR into another hash cracking application like oclhashcat-plus. This can be really useful as we are combining the processing power of both CPU and GPU.

CPU will generate the word combinations by applying word mangling rules and pass them to oclhashcat which will use the parallel processing power of GPU to crack the hashes.

An example is shown below:



```
C:\GPU Bruteforcers\oclHashcat-plus-0.081>john -i:DerBest --stdout | oclHashcat-
plus32.exe -m 2611 -a 0 -n 160 VBold.txt
oclHashcat-plus v0.08 by atom starting...

Hashes: 10
Unique salts: 10
Unique digests: 10
Bitmaps: 8 bits, 256 entries, 0x000000ff mask, 1024 bytes
Rules: 1
GPU-Loops: 64
GPU-Accel: 160
Password lengths range: 1 - 15
Platform: AMD compatible platform found
Watchdog: Temperature limit set to 90c
Device #1: Cypress, 1024MB, 0Mhz, 20MCU
Device #1: Allocating 481MB host-memory
Device #1: Kernel ./kernels/4098/m2610_a0.Cypress.32.kernel (1051980 bytes)

Starting attack in wordlist stdin mode...

Status.......: Running
Input.Mode...: Pipe
Hash.Target..: File (VBold.txt)
Hash.Type....: vBulletin < v3.8.5
Time.Running.: 10 secs
Time.Util....: 10002.8ms/8791.2ms Real/CPU, 725.6% idle
Speed........: 40129.7k c/s Real,   351.1M c/s GPU
Recovered....: 0/10 Digests, 0/10 Salts
Progress.....: 401408000
Rejected.....: 0
HW.Monitor.#1:  0% GPU, 55c Temp

words: 55574519  time: 0:00:00:13  w/s: 4004K  current: tw9pua
```

In the above example, JTR is using the DerBest Charset in Incremental mode to generate candidate passwords and passing them to Oclhashcat-plus which will in turn apply the hashing algorithm on it and perform the hash cracking.

However, the efficiency of this method will be limited by the word generation speed of JTR.

JTR POT Analyzer

I have prepared a small Perl script which can be used for Parsing the JTR POT and look for specific patterns. It will extract all the occurrences of the Pattern you specify from the JTR Pot and save it to a new file. You can use that text file to create a fake JTR POT and generate a custom charset. This can be very effective and useful for a focused attack.

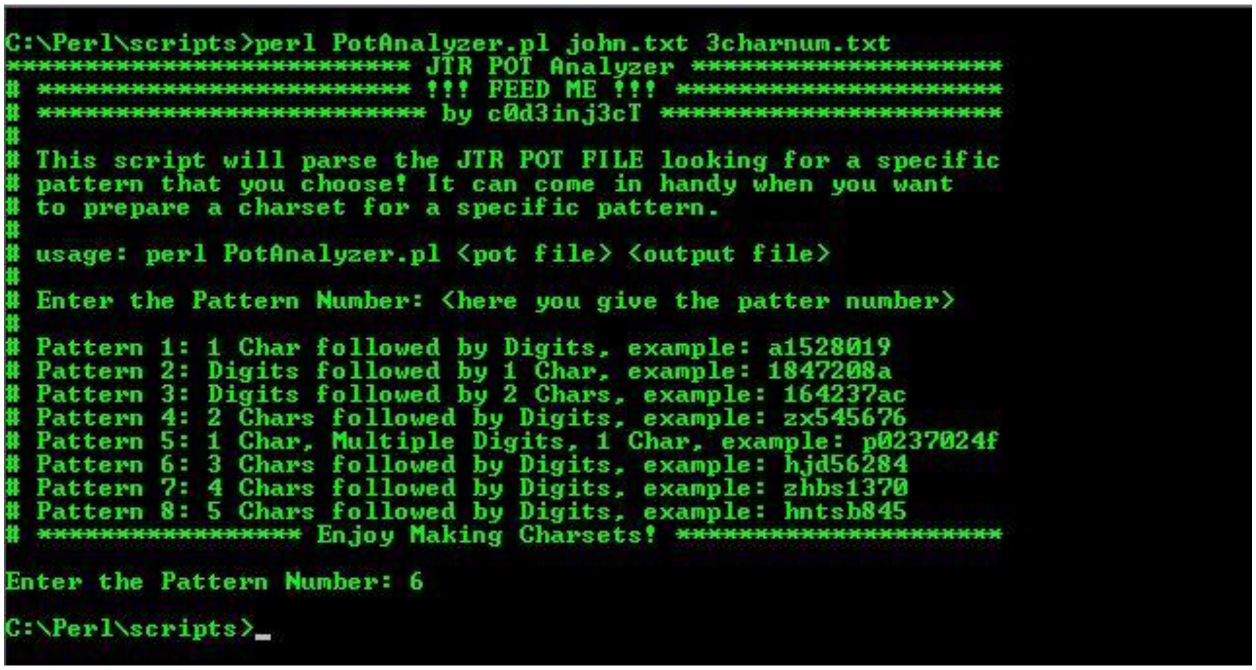
Korelogic provided a custom charset based on Rockyou Wordlist. It's good. However, we may have to pay close attention to the most frequent pattern in the cracked hashes and generate our custom charset based on that.

So, I provide this script below:

```
#! /usr/bin/perl

use strict;
use Switch;

my $input = $ARGV[0];
my $output = $ARGV[1];

chomp $input;

print q(************************** JTR POT Analyzer ********************
# ************************ !!! FEED ME !!! *********************
# ************************* by c0d3inj3cT **********************
#
# This script will parse the JTR POT FILE looking for a specific
# pattern that you choose! It can come in handy when you want
# to prepare a charset for a specific pattern.
#
# usage: perl PotAnalyzer.pl <pot file> <output file>
#
# Enter the Pattern Number: <here you give the patter number>
#
# Pattern 1: 1 Char followed by Digits, example: a1528019
# Pattern 2: Digits followed by 1 Char, example: 1847208a
# Pattern 3: Digits followed by 2 Chars, example: 164237ac
# Pattern 4: 2 Chars followed by Digits, example: zx545676
# Pattern 5: 1 Char, Multiple Digits, 1 Char, example: p0237024f
# Pattern 6: 3 Chars followed by Digits, example: hjd56284
# Pattern 7: 4 Chars followed by Digits, example: zhbs1370
# Pattern 8: 5 Chars followed by Digits, example: hntsb845
# ***************** Enjoy Making Charsets! *********************
);

my $pattern;

my ($hash,$pass);

open(DATAFILE, "$input") || die("Can't open $input:!\n");
open(OUTFILE, ">>$output") || die("Can't write to $output:!\n");
```

```
print "\nEnter the Pattern Number: ";

$pattern=<STDIN>;

chomp $pattern;

switch ($pattern) {
case 1 {
while (<DATAFILE>)
{
          chomp $_;
          ($hash,$pass) = split(":", $_);
        if($pass =~ /^[a-z]\d+$/)
        {
                print OUTFILE "$pass\n";
        }
}
}
case 2 {
while (<DATAFILE>)
{
          chomp $_;
          ($hash,$pass) = split(":", $_);
        if($pass =~ /^\d+[a-z]$/)
        {
                print OUTFILE "$pass\n";
        }
}
}
case 3 {
while (<DATAFILE>)
{
          chomp $_;
          ($hash,$pass) = split(":", $_);
        if($pass =~ /^\d+[a-z]{2}$/)
        {
                print OUTFILE "$pass\n";
        }
}
}
case 4 {
while (<DATAFILE>)
{
          chomp $_;
          ($hash,$pass) = split(":", $_);
        if($pass =~ /^[a-z]{2}\d+$/)
        {
                print OUTFILE "$pass\n";
        }
}
}
case 5 {
while (<DATAFILE>)
{
          chomp $_;
          ($hash,$pass) = split(":", $_);
        if($pass =~ /^[a-z]\d*[a-z]$/)
        {
                print OUTFILE "$pass\n";
        }
}
}
case 6 {
while (<DATAFILE>)
{
          chomp $_;
          ($hash,$pass) = split(":", $_);
        if($pass =~ /^[a-z]{3}\d+$/)
        {
                print OUTFILE "$pass\n";
        }
}
}
case 7 {
while (<DATAFILE>)
{
          chomp $_;
          ($hash,$pass) = split(":", $_);
        if($pass =~ /^[a-z]{4}\d+$/)
        {
                print OUTFILE "$pass\n";
        }
}
}
case 8 {
while (<DATAFILE>)
{
          chomp $_;
          ($hash,$pass) = split(":", $_);
        if($pass =~ /^[a-z]{5}\d+$/)
        {
                print OUTFILE "$pass\n";
        }
}
}
else { print "No patterns for you!\n"; }
}

close(OUTFILE);
close(DATAFILE);
```

A Sample of the Script in Action:



```
C:\Perl\scripts>perl PotAnalyzer.pl john.txt 3charnum.txt
************************** JTR POT Analyzer ********************
# ************************ !!! FEED ME !!! *********************
# ************************** by c0d3inj3cT *********************
#
# This script will parse the JTR POT FILE looking for a specific
# pattern that you choose! It can come in handy when you want
# to prepare a charset for a specific pattern.
#
# usage: perl PotAnalyzer.pl <pot file> <output file>
#
# Enter the Pattern Number: <here you give the patter number>
#
# Pattern 1: 1 Char followed by Digits, example: a1528019
# Pattern 2: Digits followed by 1 Char, example: 1847208a
# Pattern 3: Digits followed by 2 Chars, example: 164237ac
# Pattern 4: 2 Chars followed by Digits, example: zx545676
# Pattern 5: 1 Char, Multiple Digits, 1 Char, example: p0237024f
# Pattern 6: 3 Chars followed by Digits, example: hjd56284
# Pattern 7: 4 Chars followed by Digits, example: zhbs1370
# Pattern 8: 5 Chars followed by Digits, example: hntsb845
# **************** Enjoy Making Charsets! *********************

Enter the Pattern Number: 6

C:\Perl\scripts>_
```

A sample output:

```
sec510
mzg123
kos123
meo1991
qwe1234
uja321
meg123
rad2009
iza200885
kat9566
lol098
elo600
pol933
```

We can create a New POT File using this and create a custom charset from it.

Word Mangling Rules

It is helpful to have good wordlists of real life passwords. There are several resources on Internet where you can find a wordlist. However, there is a common misconception that the larger your wordlist is, the higher are your chances of cracking a hash.

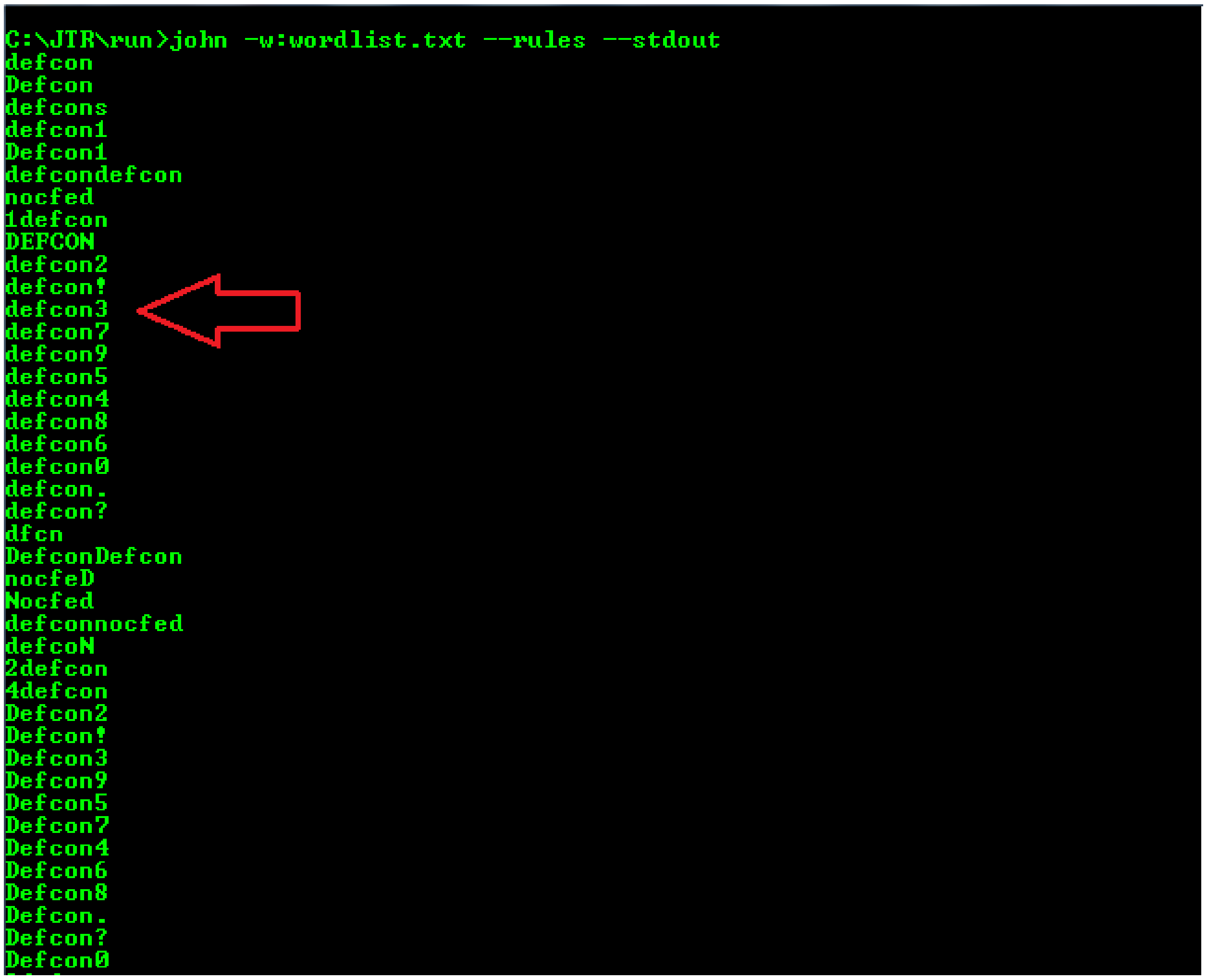
Wordlists form the base of your attack but your complete attack should not rely on the wordlist. Users often chose passwords which are a modified version of a Dictionary Word.
It is possible to generate all such combinations and keep adding them to a wordlist. However, there is a more efficient way. These combinations can be generated at run time.

Today, all the hash cracking softwares such as Hashcat/Oclhashcat/Passwords Pro/JTR have a good support for Word Mangling Rules.

The most sophisticated among all of them is the Rule Based Engine of JTR. To understand this better, let us consider the password of a user is the word defcon3.

There is a close to 100% probability that the word defcon will be present in your dictionary, however, the likelihood of the word defcon3 being present in your dictionary is very less.

Now, let us see various combinations from the word defcon that JTR can generate with its default rule set during run time.



```
C:\JTR\run>john -w:wordlist.txt --rules --stdout
defcon
Defcon
defcons
defcon1
Defcon1
defcondefcon
nocfed
1defcon
DEFCON
defcon2
defcon!
defcon3
defcon7
defcon9
defcon5
defcon4
defcon8
defcon6
defcon0
defcon.
defcon?
dfcn
DefconDefcon
nocfeD
Nocfed
defconnocfed
defcoN
2defcon
4defcon
Defcon2
Defcon!
Defcon3
Defcon9
Defcon5
Defcon7
Defcon4
Defcon6
Defcon8
Defcon.
Defcon?
Defcon0
```

We can see that the word defcon3, which is the password of user in our example, was generated by JTR at runtime. Hence, even without having this password in your dictionary, you can crack it easily.

Let us now understand the Word Mangling Rule which was used to generate this candidate password.

In the JTR’s Configuration File, locate the section with the name, [List.Rules:Wordlist]

In this section, we have the following rule which was used to generate the word defcon3 from defcon:

<* >2 !?A l $[2!37954860.?]

The rule engine parses the rule from left to right evaluating each command as it encounters it. Each command is applied to the output of the previous command. If a condition does not satisfy for a command then the rule skips to the next candidate password.

The first command "<*" indicates that a candidate password will be rejected if it is not less than the maximum length supported for a hash type.

For instance, hash types such as DES support only a length of up to 8 chars. Any word with a length greater than 8 characters will be truncated.

So, if we are cracking a DES hash, the rule will ensure that it will reject candidate passwords which are of a length greater than 8.

The second command, "**>2**", will ensure that the length of the candidate password is greater than 2.

The third command, "**!?A**" will reject the word if it belongs to character class A. Character class A is the complement of the character class, ?a which is [a-zA-Z]

Hence, the rule will reject all candidate passwords which contain a character that does not belong to the character class, [a-zA-Z]

The next, command, "**l**" is used to convert the result of the previous command to lowercase.

The last command, "**$[2!37954860.?]**" is used to append characters to the candidate password.

The characters within square brackets are expanded using preprocessor expansion. So, it will expand this into separate rules,

```
<* >2 !?A l $2
<* >2 !?A l $!
<* >2 !?A l $3
```

And so on.

The rules engine of JTR makes use of a strong syntax and it allows you to write your own Word Mangling Rules.

It is worth mentioning about the Hashcat-utils package at this point. The hashcat developers have been generous enough to prepare a set of programs that allow dictionary manipulations, wordlist optimizers and other utilities to tweak your cracking sessions.

You can read more about them here:

http://hashcat.net/wiki/hashcat_utils

From an advanced hash cracking perspective, there are a lot more ways you can tweak your attack. The aim of this paper was to introduce you to methods which can be extended as well.

Before concluding the paper, I would like to draw your attention to Hashing Algorithm Implementation Feasibility on GPUs and CPUs.

GPU Vs CPU

Do GPUs outperform CPUs for every hash type out there? There is a common misconception that any cryptographic hashing algorithm, if implemented on a GPU will give a faster hashing speed as compared to CPU. This may be true for a few hash types like the MD5, SHA1, SHA256 hashes, but it does not hold true for all.

We will see what factors let us decide the implementation feasibility on a GPU as opposed to a CPU.

32 bit Vs 64 bit Integer Operations:

Cryptographic Hashing Functions operate essentially on Integers and bit vectors. Their ability of Integer Computations determines the performance.

Hashing algorithms such as SHA1, SHA256 and its truncated version, SHA224 operate on 32 bit integers. They perform both logical and arithmetic operations on 32 bit integers. GPUs by Nvidia and ATI provide a good support for 32 bit integer operations.

Below is a snippet of the _Init function from C Implementation of SHA256 algorithm:

```
int SHA256_Init (SHA256_CTX *c)
    {
#ifdef OPENSSL_FIPS
    FIPS_selftest_check();
#endif
    c->h[0]=0x6a09e667UL;   c->h[1]=0xbb67ae85UL;
    c->h[2]=0x3c6ef372UL;   c->h[3]=0xa54ff53aUL;
    c->h[4]=0x510e527fUL;   c->h[5]=0x9b05688cUL;
    c->h[6]=0x1f83d9abUL;   c->h[7]=0x5be0cd19UL;
    c->Nl=0;    c->Nh=0;
    c->num=0;   c->md_len=SHA256_DIGEST_LENGTH;
    return 1;
    }
```

We can see the eight variables initialized with 32 bit integers.

On the other hand, hashing algorithms such as SHA512 and its truncated version, SHA384 work on 64 bit integers for both arithmetic and logical operations. Since the support for these operations is not yet optimized on GPUs, the performance increase by implementing these hashing algorithms on GPU is not as much as their predecessors.

If we look at the developer manual of Nvidia's CUDA, it states:

Operations on 64 bit integers compile to multiple instruction sequences on some GPU depending on compute capability.

Below is a snippet of code from SHA512 implementation here:

```
int SHA512_Init (SHA512_CTX *c)
    {
#ifdef OPENSSL_FIPS
    FIPS_selftest_check();
#endif
    c->h[0]=U64(0x6a09e667f3bcc908);
    c->h[1]=U64(0xbb67ae8584caa73b);
    c->h[2]=U64(0x3c6ef372fe94f82b);
    c->h[3]=U64(0xa54ff53a5f1d36f1);
    c->h[4]=U64(0x510e527fade682d1);
    c->h[5]=U64(0x9b05688c2b3e6c1f);
    c->h[6]=U64(0x1f83d9abfb41bd6b);
    c->h[7]=U64(0x5be0cd19137e2179);
        c->Nl=0;        c->Nh=0;
        c->num=0;       c->md_len=SHA512_DIGEST_LENGTH;
        return 1;
    }
```

In the _Init function, eight 64 bit variables are initialized.

Source can be found here:

http://www.opensource.apple.com/source/OpenSSL098/OpenSSL098-35/src/crypto/sha/sha512.c

Memory Usage:

There are a few hashing algorithms such as **bcrypt** which involve a certain amount of memory usage to carry out the hashing process.

It involves accesses to a table which is constantly altered throughout the algorithm execution. The size of this table is a few KB.

In a GPU, multiple cores have access to a small amount of shared RAM. They can also read from or write to the main GPU RAM with certain access restrictions. Not all the cores can read from or write to the main GPU RAM simultaneously. Due to the size of the mutating table used in the bcrypt hashing algorithm, it needs to be stored by each GPU Core in the main RAM and they will compete for memory bus.

As a result of this, full parallelism cannot be achieved by implementing bcrypt on GPU because at any point of time most of the GPU Cores will be waiting for the memory bus to become free.

Conclusion

After reading this paper, you should be able to understand the importance of a Strong Hashing Algorithm along with a Strong Password.

The techniques presented in this paper should allow you to develop your own methods of Hash Cracking to optimize the speed.

References

http://openwall.com/john/

http://www.openwall.com/john/doc/RULES.shtml

http://hashcat.net/oclhashcat-plus/