Digital Credential Analysis (Mathematical Analysis)

May 21st 2017 Douglas Berdeaux, weaknetlabs@gmail.com

Abstract

In this paper, we will explore a new method, using smarter character sets, for sequential byte-by-byte offline[1] attacking of password hashes during a penetration test. This is a companion to the *Digital Credential Analysis* ((WeakNetLabs), 2015) paper where we explored using smarter word-lists and permutations of strings, rather than building the strings on a per-byte basis as we do in this document, to attack password hashes. Here we explore how a simple default character set is also too mature for any real-world penetration testing example and how we can build smarter character sets for quicker password recovery. There will be a small amount of programming examples in the paper, but it is not necessary to fully understand the C programming language to follow along as each example function is fully explained. Also, a few key concepts of programming were left out of this document, to hopefully broaden the audience, such as recursion and commonly used, tested and known fast, algorithms for calculating permutations. There will also be a few mathematical concepts introduced using basic Algebra and Calculus.

Programming Permutations

In the next few exercises, we will explore how to make permutations of characters procedurally using C programming. This is not, by any means, a tutorial in C programming and the code can be skipped over if desired as it is explained in the text that follows. It is important though, to read through the descriptions.

A *permutation* is a way to represent all possible ways of a given set for a given length. For instance, if we're allowed to choose from the character set "abc" then all permutations would be:

```
"aaa", "aab", "aac", "aba", "abb", "abc", "aca", "acb", "acc",
"baa", "bab", "bac", "bba", "bbb", "bbc", "bca", "bcb", "bcc",
"caa", "cab", "cac", "cba", "cbb", "cbc", "cca", "ccb", "ccc"
```

Permutations differ from *combinations*, as order is important, since we are looking for a specific string (the password) and using a specific character set while doing so. This difference is important to understand because the mathematics involved for all possible combinations vs. the mathematics involved for all permutations is very different and can produce vastly different results.

Character Sets

In the upcoming examples, we will be constructing our *character set* or specific list of characters that we will use to try to guess, or crack, a password (on a per-byte basis) by trying to re-create it's associated hash. Our final set will include special characters[2], uppercase characters, lowercase characters, and numbers 0-9. Let's consider the simple application below. This application runs through the integer values of 97 to 122. These are the corresponding ASCII values to all lower case letters, 'a','b','c',…'z'.

[1] "Attacking" password hashes in this paper refers to the offline process of comparing sequential character strings passed through a hashing algorithm to the password hash in an offline manner. This paper does not explore how to brute-force passwords via the web, or any other authentication system.

[2] Characters that reside outside of the typically used alpha-numeric characters.

```
#include<stdio.h>
int main(void){
 register short i; // non C99
 for(i=97;i<=122;i++){ // "a"=97..."z"=122
  printf("%c\n",i);
 }  // O(n) where n = 123-97 ASCII bytes[3]
 return 0;
}
```

This, obviously, requires 26 iterations. If we were trying to guess a single-character, lowercase, password, we would only need a *rainbow table*, or a file or data storage containing all possible permutations of a given set that will include the user's password, of 26 lines. Now, what if we need to include all uppercase characters as well? This increases our rainbow table and cracking attempts in a *linear* fashion by simply doubling it in size,

$$26 \text{ bytes} * 2$$

by adding 26 more lines to the rainbow table. So, let's update our character loop tool above to include the uppercase bytes. These byte's ASCII values are 65 through 90.

```
#include<stdio.h>
int main(void){
 register short i; // non C99
 for(i=65;i<=122;i++){ // "a"=97..."z"=122
  if(i==91){ i+=5; continue; } // skip 91-96
  printf("%c\n",i);
 } // O(n) where n = (123-65)-(97-91)
 return 0;
}
```

The above C program will print A-Z then jump the variable `i` to the ASCII value of the first lowercase character, 'a', with the value of 97, and print a-z before returning to the shell. This will be $26 * 2$, for 52 bytes, of iterations through the `for()` loop.

Next, let's explore how our single-character password character set could include digits. This does not increase the strength of our password policy, since the digit is not a requirement, just an option. This is obvious, since the password is only a single character in length. We can think of this as simply being Boolean (OR) logic, meaning the digit is a possibility, but not (AND) a requirement. This simply makes the job of the password cracker *slightly* more difficult, but not much more, since we need to add $0,1,2,3,4,5,6,7,8,$ and 9 to our rainbow table. Let's update our C program to return the digits as well to our rainbow table.

[3] 123 is because we want 122, 121, … and 97.

```
#include<stdio.h>
int main(void){
 register short i; // non C99
 for(i=48;i<=122;i++){ // "a"=97..."z"=122
  if(i==58){ i+=6; continue; } // skip 58-65
  if(i==91){ i+=5; continue; } // skip 91-96
  printf("%c\n",i);
 } // O(n) where n = (123-48)+(97-91)+(64-58)
 return 0;
}
```

This linear function will print 0-9, then A-Z, then a-z and return to the shell. This produces $10 + 26 + 26$, or 62 lines for our rainbow table to crack a single character password. Since most password policies require us to use "*special characters*", or characters that exist outside of the typically used alpha-numeric set, let's modify our C program to output special characters also. This actually simplifies our C program since our `for()` loop's `i` variable only has to go from 33 to 126 without any jumping.

```
#include<stdio.h>
int main(void){
 register short i; // non C99
 for(i=33;i<=126;i++){ // "a"=97..."z"=122
  printf("%c\n",i);
 }
 return 0;
} // O(n) where n = 127-33
```

This linear function will print all 94 ASCII characters usable in our password policy: 0-9, A-Z, a-z, and !"#$%&'()*+,-./0123456789:;<=>?@ [\]^_`{|}~

The special character set alone is 32 bytes in size when including all characters listed above. ASCII values 34-48, 59-65, 92-97, and 124-127 are 32 bytes total:

```
Dec: 34, ASCII: !
Dec: 35, ASCII: "
Dec: 36, ASCII: #
Dec: 37, ASCII: $
Dec: 38, ASCII: %
Dec: 39, ASCII: &
Dec: 40, ASCII: '
Dec: 41, ASCII: (
Dec: 42, ASCII: )
Dec: 43, ASCII: *
Dec: 44, ASCII: +
Dec: 45, ASCII: ,
Dec: 46, ASCII: -
Dec: 47, ASCII: .
Dec: 48, ASCII: /
Dec: 59, ASCII: :
Dec: 60, ASCII: ;
Dec: 61, ASCII: <
Dec: 62, ASCII: =
Dec: 63, ASCII: >
Dec: 64, ASCII: ?
Dec: 65, ASCII: @
```

```
Dec: 92, ASCII: [
Dec: 93, ASCII: \
Dec: 94, ASCII: ]
Dec: 95, ASCII: ^
Dec: 96, ASCII: _
Dec: 97, ASCII: `
Dec: 124, ASCII: {
Dec: 125, ASCII: |
Dec: 126, ASCII: }
Dec: 127, ASCII: ~
```

The full sum will be our final character set used throughout this paper,

```
!"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Remember that order *is* important. This is a specific set, that requires a specific order for the permutation mathematics used. As we can see, adding special characters, numbers, and uppercase characters, a possibility in our password policy made our rainbow tables larger.

Strength Analysis

Okay. Let's recap a bit, shall we? So far, with our single character password, "*strength*" depends on two factors,

1. *What character (password) the user has chosen,*
2. *The order of the password cracking program's output (character set)*

Linear Functions $O(n)$

For instance, if the user chose the password '!', because we allowed the *special characters*, then our ordered character set provided by our C program would have guessed the password in the first attempt. Does that mean the user has chosen a "*weak*" password? No. All passwords can be cracked, knowing the hashing algorithm and given the salts, if required, and the password hash itself. This just proves that some passwords can be cracked much faster than others and, in this case, that's arbitrarily up to the attacker's character set. If you have noticed the scientific *Big O* notation, a way of representing the efficiency of a function or algorithm, remarks that I have put into the C function examples, each has been a simple linear $O(n)$ where n was the number of characters in the character set. Obviously, our cracking program would also be $O(n)$, but n depends on the character chosen as the password by the user and the character set used by the attacker. In the previous example of , '!', it was $O(1)$, or as dropping all constants and absolute magnitude in Big O, becomes again $O(n)$.

Let's now approach the idea as to why we want to increase the work needed by the attacker to crack the password hash, *exponentially,* thus increasing the "*strength*" of the password.

Exponential Functions $O(n^2)$

So, we now have a C program that will guess, or crack, a password with no problem. Unfortunately, this is for cracking a single-character password, and that usually isn't the case! We will typically need much more than 1 character, or byte. In fact, many password management tools and services require its users to choose a password of *at least* 8 characters, or bytes, in length. And as standards grow stronger from influence in the information security and data science fields, this value will ultimately increase.

Before we dive right into the mathematics and programming functions required for 8 character passwords, let's take it slow and start at 2-byte passwords first. Let's increase our C program to make a rainbow table of 2 character passwords using all the characters from the set described above, A-Za-z0-9 and special characters.[4]

```
#include<stdio.h>
#define STARTASCII 33
#define ENDASCII 126

void twoBytes(int i); // prototype

int main(void){
 short int i;
 for(i=STARTASCII;i<=ENDASCII;i++){
  twoBytes(i);
 }
 return 0;
} // O(n^2) where n = 127-33

void twoBytes(int i){
 int ascii; // start at 'a'
 for(ascii=STARTASCII;ascii<=ENDASCII;ascii++){
  printf("%c%c\n",i,ascii);
 }
 return;
}
```

The code above will print all *permutations* of a 2-byte password using the character sets, A-Z, a-z, 0-9, and special characters. We define the beginning ASCII value and end ASCII value as constants to be easily updated in a single place by the user if needed. We also prototype, or clearly define a new function `twoBytes()`'s interface.

The term *permutation* simply means any single way we can match any bytes in the specified set. We must also consider that repetition is allowed, since our user's 2-byte password could be something like "AA", "99", "--", or even "gg".

It might be easier to visualize this as a two-character combination lock,[5] similar to a letter combination bike lock, or Wordlock,[6] but with every single character in our character set on each of the two spinning wheels. As the first wheel is set to '!' the second wheel will spin, we will call this process *rotation*, from '!' to the end of the character set, '~' and that will be one full set of permutations where each starts with '!'. Moving on, the first wheel would move to " (double quotes) and the second would again rotate from '!' to the end of the character set, '~', making another full set of permutations that begin with the character ". When this process is repeated until the first wheel lands on the last character in the character set and the second wheel completes a full rotation, this will complete the process of making all permutations of the character set for two-byte passwords.

[4] Recursion was purposefully avoided for brevity and to not introduce more computer science optimization concepts that aren't necessary for this paper.
[5] "Permutation Lock" from Reddit.com
[6] Wordlock is a brand of combination locks made by Wordlock Inc.

This application main() and twoBytes() is now an $O(n^2)$ function, because for each set of ASCII bytes in `main()`, we loop through them a second time in `twoBytes()`. So, after this edit to our output code, how much exactly did our rainbow table grow when we simply added another byte to the length of the password? Here is a word count for lines of output for the code,

```
trevelyn@80211:~/passwdEquation$ gcc pass.c -o genpass.exe
trevelyn@80211:~/passwdEquation$ ./genpass.exe|wc -l
8100
trevelyn@80211:~/passwdEquation$
```

We can see that our rainbow table grew from all single byte possibilities, $26 + 26 + 10 + 32$, or 94 lines, to all two-byte permutations, or $8{,}100$ lines. This should be concrete evidence that to "*strengthen*" the security of our user's passwords we should be enforcing a policy that requires more bytes, and not necessarily special characters. In fact, requiring a special character only added 32 lines to our rainbow table, and increased the CPU cycles, (work) required for the attacker, linearly. In fact, if we remove the special character, and we run this application again, the difference between file sizes (in lines in our rainbow tables) is still very significant when compared to simply adding 32 new bytes to choose from for special characters.

Let's move one and add another byte requirement to our password policy, 3 total, and check the lines and overall size this adds to our new rainbow table file. Our `twoByte()` function then would look something like so,

```
void twoBytes(int i){
 int ascii; // start at 'a'
 int ascii2;
 for(ascii=STARTASCII;ascii<=ENDASCII;ascii++){
  for(ascii2=STARTASCII;ascii2<=ENDASCII;ascii2++){
   printf("%c%c%c\n",i,ascii,ascii2);
  }
 } // twoBytes() is now O(n²) but called by an O(n), so together make O(n³) total.
 return;
}
```

Again, we have simply nested yet another `for()` loop into the `twoBytes()`'s `for()` loop. The above function `twoBytes()`, when called from `main()`, produces all permutations of our character set for 3 byte passwords. When ran, this produces $830{,}584$ lines to our rainbow table. As the comment mentions, this is an $O(n^3)$ application, as `main()` already loops through n, and for each iteration calls `twoBytes()` who loops over n per n. To double-check this application's output, we use the permutation mathematical equation,

$$n^r$$

Where n is the number of characters to choose from, our character set, 94, and r is the number of bytes the password is in length, in this case 3. So, 94^3 returns $830{,}584$ possible permutations. This means that our Big O notation relies the algorithm in which calculates each permutation, in our case, it becomes $O(n^r)$.

If we update the function to include a fourth byte to our password policy requirement, we have something like so,

```
void twoBytes(int i){
 int ascii;
 int ascii2;
 int ascii3;
 for(ascii=STARTASCII;ascii<=ENDASCII;ascii++){
  //printf("%c%c",i,ascii);
  for(ascii2=STARTASCII;ascii2<=ENDASCII;ascii2++){
   for(ascii3=STARTASCII;ascii3<=ENDASCII;ascii3++){
    printf("%c%c%c%c\n",i,ascii,ascii2,ascii3);
   }
  }
 }
 return;
}
```

This will produce all permutations of 4 byte passwords from our character set, or, 94^4 lines to our rainbow table. We can easily keep going, nesting our for() loops to produce all permutations up to 8 bytes and this finally yield, 94^8, or $6.095E15$ lines into our rainbow table. That's 6,095,000,000,000,000 or 6 quadrillion lines. Each line 8 bytes: then that file size would be 4.8E16, or 480,000,000,000,000,000 480Petabytes[7] in size. In other words, a size that is too large for most modern drives as of the writing of this paper.

Rainbow tables exist for a time/space tradeoff. So, is the rainbow table even a good idea when it gets this large? If the password used was *"!!!!!!!!"* which is 8 bytes of the first character in our character set, then it is not efficient. This is simply because the first guess would be correct when our C Program outputs directly to the hashing/cracking program, and we would have 6 quadrillion less one useless lines in our rainbow table file. Let's try a real world example of a commonly used password before exploring this idea any further.

Password Permutation Equation

Remembering from a previous section, the strength of the password policy depends on two factors,

1. *What character password the user has chosen,*
2. *The order of the password cracking program's output*

The password cracking program's output is simply our character set used to guess, or crack the password. So, can we figure out exactly how many attempts, or permutations, it will take to crack a password given the password and the character set?

Well, since we know that a password will simply be a member of a set of permutations, we simply need to add the permutations that it takes to get to the given password. We also know that a full set of permutations is n^r Let's begin with a simple example. Let's choose the password,[8]

"*password123*"

[7] 2^{50} bytes; 1024 terabytes, or a million gigabytes. – Google.com 2017

[8] See "So, Whose Responsibility Is It Anyways?" section of this document for this reference from Carnegie Mellon University.

for our first example. We know, from our experience with the previous C programming examples, that this will require the attacker to develop an $O(n^r)$ algorithm if we continued along the path of nesting `for()` loops for each byte in length as we did moving through the examples. Again, our character set is ASCII bytes $33 - 126$ in that specific order. This is,

```
!"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

So, to begin, we know that our password starts with a 'p' character, and that it is 11 bytes in length. This means that we need to calculate all the permutations it will take to get to pXXXXXXXXXX first, where Xs are used to denote characters that we need to fully rotate[9] per permutation. If we break this string out by shifting off the 'p', we have "XXXXXXXXXX" which considering all permutations would be 94^{10} permutations. Next, we need to multiply this amount of permutations by the amount of rotations it would take our first byte to get to 'p' and that would simply be the sum of bytes from '!' to 'p' sequentially in our character set, which is,

26 *uppercase letters* $+ 10$ *digits* $+ 16$ *lowercase letters (a-p)* $+ 6$ *special* $+ 7$ *special* $+ 15$ *special* $= 80$ *rotations (from '!' to 'p').*

This comes to $94^{10} * 80$. Then, we simply repeat this process to find 'a' plus a permutations of XXXXXXXXX. Again, using n^r we have 94^9 (for all Xs) multiplied by the amount of rotations until we get to the 'a' character in the first letter place, which is,

26 *uppercase letters* $+ 10$ *digits* $+ 1$ *lowercase letter (a)* $+ 6$ *special* $+ 7$ *special* $+ 15$ *special* $= 65$ *rotations (from '!' to 'a').*

Our current sum, so far, is $94^{10} * 80 + 94^9 * 65$. To continue, we need to now find our first 's'XXXXXXXX.

26 *uppercase letters* $+ 10$ *digits* $+ 19$ *lowercase letters (a-s)* $+ 6$ *special* $+ 7$ *special* $+ 15$ *special* $= 83$ *rotations (from '!' to 's').*

This becomes $94^8 * 83$ and our total sum, this far, is $94^{10} * 80 + 94^9 * 65 + 94^8 * 83$. A pattern should be obvious by now. We can write this sum in a much smaller equation using summation notation. Again, this equation and these results rely upon the order of the character set that we defined in the beginning of this study. Let's now look at defining this equation.

Since we have the repeating pattern: *number of characters in our character set to the exponential power of the length of the password less the character place multiplied by the amount of permutations it takes to get to the character*, I have summed this method up into a formula as,

[9] Just as we did with our Wordlock brand Wordlock example: Programming Permutations

$$\sum_{i=1}^{k}(xn^{r-i}) = y$$

Where k is the length of the password, n is the length of the character set, and r is the length of the password to crack, x is the number of characters from the beginning of the character set to the character in the password.[10] So, our password, "*password123*", using the equation above, is:

$$(94^{11-1} * 80) + (94^{11-2} * 65) + (94^{11-3} * 83) + (94^{11-4} * 83) + (94^{11-5} * 87) + (94^{11-6} * 79) + (94^{11-7} * 82) + (94^{11-8} * 68) + (94^{11-9} * 17) + (94^{11-10} * 18) + (94^{11-11} * 19)$$

Which yields $2.55977529E35$ as y, permutations. To possibly better visualize the significance of this number, let's explode it into decimal notation, that's

$$255{,}977{,}529{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000$$

permutations to design a full rainbow table of 11 character passwords from "!!!!!!!!!!!" to "password123" to crack the password. That is a colossal number when speaking in terms of storage and CPU cycles modern-day for computers! Now, take that sum, multiply it by 12 (11 bytes + newline for a line-by-line password rainbow table) and then divide the product by $1{,}024$. This quotient yields the amount of kilobytes necessary for the file size. Divide the quotient again by $1{,}024$ and the new quotient yields the total Megabytes required for a rainbow table of passwords from *"!!!!!!!!!!!"* to "*password123*". Repeating the process will yield, Gigabytes, Terabytes, Petabytes, and so on. As of the date of this paper, that is a daunting number that is physically impossible to store for the, seemingly irrelevant task of cracking a password like "*password123*" that almost always is already in a password list readily available online to be used to crack passwords!

So, Whose Responsibility Is It Anyways?

This was all brought up and designed because someone sent me to the CMU/CUPS Password Meter online. They say that "*password123*" cannot be used because it is "*extremely common*."

[10] Note, the rule of PEMDAS in this situation, calculate the exponent value first, $r - i$, then the exponent and base, n^r, then multiply by x. The exploded view's parenthesis were used to signify importance during the actual summation of each evaluated expression.

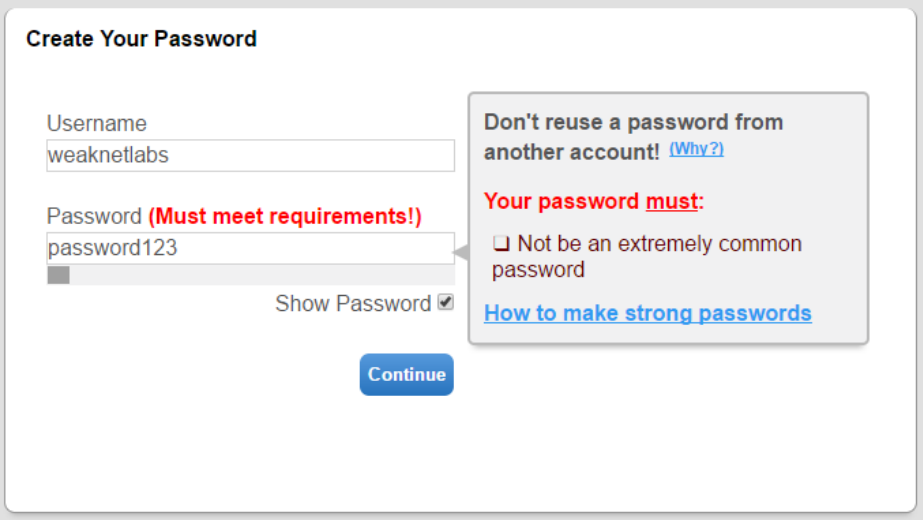

Figure 0. CMU/CUPS Password Meter Screenshot from https://cups.cs.cmu.edu/meter/

We saw that trying to guess every character from *"!!!!!!!!!!"* to "*password123*" was in impossible feat. But, I think what the author of this password meter means is that if a password hash were recovered maliciously, say via SQL Injection[11], then this password, "*password123*," would most-likely be in the password rainbow table used against the hash.

In this case, the "*strength*" of the password is being left to the user's imagination.

Crypto Methods of Protection

So, how can we, as programmers and application architects, increase the amount of work needed before the password is cracked, thus, increasing the passwords *strength*? Well, there are many cryptographic methods that programmers can employ, such as *key-stretching*, which passes the plain text password's hashed result through the same mathematical algorithm often thousands of times before creating a hashed output.[12]

Another method is called *salting* the password with yet another string before sending into the hashing algorithm. These methods are typically used by programmers as standards for keeping user's passwords in their own databases. One method increases the amount of work the computer needs to do *linearly*, while the other increases the work the computer needs to do *exponentially*.

Locks, Expirations, History, and Services

Servers and services, like Identity Access Management, or IAM, services also employ the idea of *expiring passwords*. This is where we can finally return to the idea of determining the balance between the space/time tradeoff that we mentioned earlier with rainbow tables. If the table gets too large to store on a modern disk, (space) we obviously need to rely on an application which generates the passwords and sends them into the password cracking tool (time). Considering parallel processing and the rate modern

---

[11] SQL and Command Injection are common ways of retrieving password hashes from web applications. https://en.wikipedia.org/wiki/SQL_injection

[12] In WPA2, the passphrase is slated with the ESSID and passed through an algorithm, PBKDF2/HMACSHA1 4,096 times before producing the final MIC / hashed output: https://github.com/weaknetlabs/WPA2-Cracking-Perl/blob/master/wpa-crack.pl

GPU/CPUs are growing in speed and size, this task may seem feasible. Especially if we can properly craft our character sets using basic statistics from reports of leaked *organic digital credential data*[13] (passwords from compromised servers). This means that, in the event of an unfortunate breach and password hashes are compromised, combining the cryptographic methods previously stated with a relatively small life of a user's password (by shortening the time between expirations), we have already lowered the possibility of the leaked password hash to be cracked in the timeframe of the password's life.[14]

Another idea that IAM, services commonly employ is *locking an account* upon a predetermined number of login failures. This is where the CMU/CUPS statement makes sense. *Guessing*, or brute force, is a password is a hack that's been around since the beginning of the idea authentication. If the IAM service locks the account after a very small number of failed login attempts, however, this would prompt the user or the admins of the IAM service, that the account was locked by a potential brute force attack. This is why using a commonly-used password is not a good idea, but, like the byte per byte offline attack on a password hash that relies on the attacker's character set, a successfully attack relies on the order of the dictionary file used in the digital credential guessing/brute force attack software. For instance, if we were to create a wordlist of the most commonly used passwords from the timeframe of the creation of this document,[15] right before a penetration test, it would include,

1. 123456
2. 123456789
3. qwerty
4. 12345678
5. 111111
6. 1234567890
7. 1234567
8. password
9. 123123
10. 987654321
11. qwertyuiop
12. mynoob
13. 123321
14. 666666
15. 18atcskd2w
16. 7777777
17. 1q2w3e4r
18. 654321
19. 555555
20. 3rjs1la7qe
21. google
22. 1q2w3e4r5t
23. 123qwe
24. zxcvbnm
25. 1q2w3e

So, as we can see, "*password123*" was not even in the top 25 most commonly used passwords. If the IAM service locks the account before 25 failed attempts, hopefully sooner than that, then the account holder will be forced to update their password.

[13] From Digital Credential Analysis – 2015 WeakNet labs
[14] Here "*life*" refers to the duration of time the password is valid and not expired.
[15] Telegraph.co.uk - The world's most common passwords revealed: Are you using them?

If the *history* of the password is kept indefinitely, this would mean that the user could never choose this commonly used password again. As developers and administrators, we can only hold the user's hand so much when creating digital credentials. These countermeasures ultimately attempt to restrict the attacker from unauthorized access to account information, but also restrict the user from unwittingly helping the attacker in any way possible.

Statistically Speaking of Organic Leaks

Previously Published Leaked Organic Credential Data

An organic leak, as mentioned in the Digital Credential Analysis paper, is a leak of digital credential data that consists of passwords made by actual users. This data can be used statistically to make our character sets much more efficient. For instance, if many leaked passwords include the word "*password*" shouldn't our character set begin with the characters 'p','a','s','w','o','r', and 'd'? What about passwords that are "*1234567*" ? Shouldn't we add those characters to the *beginning* of our character sets? In doing so, our character sets become more efficient and allowing the passwords to be cracked in much less CPU effort and time. Again, we assume that this is for an offline brute-force password cracking attack on a hashed password.

For instance, if we changed our character set to be

```
pasword1234567bcefghijklmnpqtuvxyz890ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-
./[\]^_`{|}~
```

As intended, we simply moved the most used character bytes to the front of the line, so to speak. This significantly cuts back on the amount of permutations required and increase our chances of cracking the password hash in much more timely manner. If we apply the equation,

$$\sum_{i=1}^{k}(xn^{r-i}) = y$$

To the new character set, we have,

$$(94^{11-1} * 1) + (94^{11-2} * 2) + (94^{11-3} * 3) + (94^{11-4} * 3) + (94^{11-5} * 4) + (94^{11-6} * 5) + (94^{11-7} * 6) + (94^{11-8} * 7) + (94^{11-9} * 8) + (94^{11-10} * 9) + (94^{11-11} * 10)$$

Which is,

$$5386151140948997 0176 + 1145989604457233408 + 18287068156232448 + 194543278257792 + 2759479124224 + 36695201120 + 468449376 + 5814088 + 70688 + 846 + 10$$

This yields the permutation sum of,

$55{,}025{,}985{,}422{,}030{,}354{,}176$ permutations, or $55.025E18$, which is vastly different than $2.55{,}977{,}529E37$, our permutation sum was before manipulating our character set!

Penetration Testing

During a penetration test, if we are testing a service that anyone has access to, we can create an account and test, or read, exactly what characters are allowed. Thus, creating a character set of just those allowed characters and limiting the time/space tradeoff values. For instance, if the allowed

characters are listed as “Upper case, lower case, numbers, and #$%^*_-“ This lessens the size of our character set as we know, as attackers (and users), that these will be the character’s potentially used in the user’s password. Used in conjunction with the idea of moving more frequently used characters to the beginning of the character set, this will increase our chances of recovering the password in a smaller time frame.

Summary

In summary, we know that the cat-and-mouse game of attacking digital credentials will continue indefinitely. Let’s break this down into two perspectives, as both are equally important. This paper offered a mathematical analysis of attacking digital credentials as well as standards for securing services which are meant to safeguard them. Let’s being with defense.

Defense

As computer processing gets faster over time, or changes completely as we have seen in parallel processing on several CPU/GPU chipsets, our password policies will need to fortify to protect our users. More bytes in password length, smaller password life-cycles, good programming practices, good at-rest and during-transit practices such as encryption, strict service configuration of IAM services, and, maybe the most important, education and proof of the data science behind digital credentials are all fundamental to bring our internet to a more solid, secure state.

Offense

As penetration testers, the goal is ultimately create a more secure internet. In doing so, we find and point out the flaws in bad practices. This means that education of the items mentioned in Defense are also very important for the clients of the test. Crafting penetration test-specific character sets to attack any gleaned password hashes, better, more mature wordlists using Digital Credential Analysis, and designing penetration testing tools using efficient, optimized programming practices for speed and memory space will only help in the argument of securing our internet.

References

1. Digital Credential Analysis – https://weaknetlabs.com/files/WeakNet%20Labs%20-%20Douglas%20Berdeaux%20-%20Digital%20Credential%20Analysis%20-%202016.pdf
2. Stronger Password – CMU/CUPS: 2017 Screenshot incl. https://cups.cs.cmu.edu/meter/
3. Crunch – Character Set Wordlist Generator Tool: https://sourceforge.net/projects/crunch-wordlist/ Source Code: https://github.com/crunchsec/crunch
4. Wordlock Inc. – https://en.wikipedia.org/wiki/Wordlock
5. LDAP & IAM – https://iam.harvard.edu/get-started/ldap
6. Most Commonly Used Passwords – http://www.telegraph.co.uk/technology/2017/01/16/worlds-common-passwords-revealed-using/
7. Permutation Lock: https://www.reddit.com/r/Showerthoughts/comments/2yr9k5/a_combination_lock_is_really_a_permutation_lock/