Password Theory

Author: Aaron Toponce
Email: aaron.toponce@gmail.com
PDF: http://goo.gl/Ay39
Date: 2010-10-28

License

Overview

The structure of the presentation will be as follows:

- Cryptanalysis

Cryptanalysis: In order to understand what a "strong password" is, we need to understand what software is used and how it is used to attack passwords. This will help us get a handle on what qualifies a password as strong versus weak.

- Password structures

Password structures: We'll cover how passwords are stored in databases, hard drives, and mediums. We'll cover passwords stored in plain text, verses hashed passwards, versus salted passwords. Briefly.

We'll also take what we learned about how they are stored, and start looking at the numbers and combinations of passwords. This will prepare us for learning about entropy, which is coming up. We'll also add rainbow tables to the mix, and why salted passwords increase the strength of your password.

- Strength qualities

Strength qualities: After analyzing how to attack passwords, we can get a sense of what makes passwords strong versus weak. We learn quickly that length becomes a key factor as well as adding different character sets to the password.

- Entropy

Entropy: This is to cover the amount of work it takes for an attacker to gett access to your account if he knows your username, and is only guessing your password. We'll see how length adds much more entropy to a password than just adding random characters. We'll run some numbers on the time it would take to crack these passwords.

- Implementations

Implementations: We'll look at some various password managers and how you can keep your password safe, while easily accessible at the same time.

Password Requirements

1. "Strong" (whatever that means)

We're told that we need to make our passwords "strong", but defining what "strong" means seems to confuse people. Some will say that a strong password is a long password. Others might say that a password uses a combination of lowercase and uppercase letters, numbers and punctuation. Others, maybe more anal retentive, will say a strong password is one that is built entirely off random characters.

Needless to say, I think we can do better, and we will. Later on, we'll quantify exactly what it means to have a "strong" password

2. Easy to remember

Having a "strong" password and one that is easy to remember seem to conflict each other. The reason for this seems obvious. If I generate a "strong" password, then in becomes more difficult to remember. The stronger the password it semes, the harder it is to recall from memory. So naturally, we want to keep our password easy to remember, so we sort of compromise on the strength of the password to accomplish this aim.

Case in point, and part of the motivation for developing password theory; I had a friend who is in the computer science field. He's even a mathematician here at Weber State. Recently, he had his Gmail account compromised by an attacker. The attacker was able to crack his password, and take out the account. The reason the attacker got the password was because it wasn't "strong", but instead, easy to remember. It was a valuable lesson to him to change his password strength, not only for Gmail, but for the other accounts he had.

Hopefully, I'll be able to show that by the end of this presentation, you can have both a very strong password, and one that's easy to remember.

3. Unique for each account

Of course, if a password does get access to the password for our account, the last thing we want him to have access to is our bank account, or other accounts that we might consider a sensitive nature.

So, it's a good idea to create a unique password for each account that we have. However, this also seems to be in disagreement with having passwords that are easy to remember. I'm sure each of us have many accounts that we interface with. Is it easy to remember which password goes with which account? It might be if we interface with those accounts daily, but what about the accounts that we only access once a month, or even less than that? Forgotten passwords are very common.

At least having an unique password for each account guarantees that if one account gets compromised, the others won't be with that same password. Again, hopefully I'll be able to show that not only can you have a "strong" password that's easy to remember, but it will be easy to remember "strong" passwords for every account you have.

4. Stored securely

In reality, passwords are only as secure as their storage, with the most secure storage being your brain. However, you can write passwords down in a secure manner. You could have an encrypted database with one single master password that contains all your account passwords. Unless someone knows the master password, they won't get access to the others. You could also store your passwords in your wallet or purse, as these items are likely the most tracked items in your possession, minus maybe your kids.

There are all sorts of implementations, both software and hardware, for storing your passwords securely, and it can be done. We'll look at only one method at the very end of this presentation for securely storing your passwords.

Cryptanalysis

In order to get some sort of sense on what makes a password "strong", we need to look at the amount of effort it takes to attack a password from a cryptanalysis point-of-view.

Cryptanalysis is a branch of cryptography that studies breaking down encrypted data without the access to the secret algorithms that were used to generate that encrypted data. These methods use a variety of algorithms and theoris, such as birthday attacks, rainbow tables (of which we'll talk about here), boomerang attacks, brute force attacks and other methods.

By analyzing some of the cryptanalysis methods used to attack and recover passwords from encrypted means will help us get a handle on the hardware and software implementations needed to make these attacks practical, which means we might get a better understanding what it means to have a "strong" password.

- Hashing plain text

Hashes are one-way algorithms with take any form of data, and produce a unique hexadecimal string of characters. If the same data is provided to the hashing algorithm, the same hexadecimal string with always be produced. This means that there is a one-to-one relationship with the data and the hash.

Hashes are also one-way, meaning that it's trivial to create the hash, but we can't go in reverse. In other words, we can't take a hash, and reproduce the data that created it. An analogy to this would be creating pie crust. It takes water, flour, sugar and other ingredients. It's easy to create the dough, and eventually the crust by mixing the ingredients, but it's near impossible to take a crust, and reduce it to the ingredients necessary to create it. This is partly due to evaporation of water during heating in the oven, chemical bonding, and other factors.

- Rainbow tables

Because the relationship to the data and the hash is one-to-on (1:1), this means we could create a database of text that produces the unique hash. Although we can't revers the hash into the data, we can look up the hash in the

database, and if we find a match, then we know what the data is that produced the hash. These databases are known as rainbow tables. You can find many large and small rainbow tables on the Internet.

- Salted password

In order to work around tables, passwords are salted. Suppose a password is 8 characters long. It contains exactly one unique hash. However, what if we were to add a "salt" to the password string. Suppose the password salt is 6 characters in length and that the salt is based on a 64-character set. Then, this means that the password could produce 64^6 possible hashes. This would make our rainbow table 64^6 times as large.

64^6 = 68719476736

If all we have access to is the hash, and not the salt, then we have a 64^6 possible combinations to search through with the salt, assuming it is indeed 6 characters long, to find the right password that produced that hash. This makes rainbow tables infeasible.

- Shadowed password

Further, most passwords in databases and operating systems are "shadowed". This means that only the database or operating system administrator will have access to the salted password hash. This keeps prying eyes of regular system users away from attempting to attack the hash, and find the password that produced it. We assume that the database and operating system administrators can be trusted, due to the nature of their job.

- John the Ripper

If we can get access to a shadowed password, then we will also likely have access to the salt. This means that we've got enough data to begin searching for a strong of data that produces the hash (combined with the salt appropriately, of course). One such tool to do this is John the Ripper. John the Ripper takes an "unshadowed" file, that is the username, the salt, and the hashed password, along with some other metadata, to begin attacking the password. It does so through brute force.

John the Ripper reads a database of words to attack from, or it can start using incremental mode. The "database" of words can be a regular text file with one word per line, that is commonly used in spell checker programs to assure spelling accuracy. You can also supply options to John to reduce the search space, such as restricted character sets, suspected password length, and potential dictionary databases to search from.

John the Ripper is designed to be fast. It can take advantage of multiple cores in your CPU, to linearly increase the speed. On my 16 core system at work, I can achieve a speed of 192,000 passwords per second. With a cluster of 20 machines, with 16 cores a piece, this means I can achieve a speed of 3,840,000 passwords per second.

For an 8-character password, this means at this pace, it would take 59 years at most to find the right key, combined with that salt, to match the hash given in the shadowed password database. We'll talk more about this in a bit.

Some Hashes

- Using the MD5 hash algorithm
- `foo - d3b07384d113edec49eaa6238ad5ff00`
- `Foo - cbd8f7984c654c25512e3d9241ae569f`
- `goo - 7361528e901ca2f2f1952a68ad242d79`

Above is an example of the text 'foo' hashed with the MD5 hashing algorithm, compared to hashing 'Foo' and 'goo'. Notice how all the hashing strings are radically different from one another. The difference of letter case matters, as can be seen with 'f' and 'F'. However, the capital letter f compared to the lowercase latter are several bytes apart, as far as the computer is concerned. However, the letter g is one byte different from the letter f, so you would expect the

hash to be much more similar, but it's just as radically different.

As mentioned with hashing algorithms, they are one way functions, meaninig that it's asy to take any stream of data, binary or otherwise, and get a hash from that stream. However, it should be impossible, or highly improbably, to reverse the hash into the originating data.

Further, hashesh should be computationally secure, meaning that it should be able to stand against cryptanalysis attacks and there should be no collisions in the hash. A collision is given, when many different streams of data provide the same hash. Of course collisions will occure, due to the infinite nature of the data, and the finite nature of the hash. However, the MD5 algorithm can produce 5,444,517,870,735,015,415,413,993,718,908,291,383,296 unique hashes. This should be large enough to avoid unnecessary or frequent collisions.

Other hashing algorithms increase the search space substantially, and some of the more common hashes are given below.

Common Hash Algorithms

Most 128-bit hashes have been shown that computationally feasible cryptanalysis can reduce the search space significantly enough to make attacks practical. MD5 and RIPEMD-128 seem to be the last remaining 128-bit hashes that have withstood practical cryptanalysis. Most security experts will advise using at least 160-bit or stronger hashing algorithms. The number of unique hashes (the total search space) of each hash is given below:

- 128-bit: HMAC, MD2, MD4, MD5, RIPEMD-128

MD5: 5.44E40

- 160-bit: SHA1, RIPEMD-160

SHA1: 2.34E50

- Others: SHA224 SHA256 SHA384 SHA512

SHA224: 4.31E69 SHA256: 1.85E79 SHA384: 6.30E117 SHA512: 2.15E156

Example of a shadowed password

- `root:$1$NSESuz4A$22uWH1mOPnka4zTdnx3jx1`

This is an example of a shadowed password from the `root` account on a Unix server. Each field in the line is separated by a colon. Thus, we can break don the line as follows:

`root` `$1$NSESuz4A$22uWH1mOPnka4zTdnx3jx1`

The only daa we're going to concern ourselves with is the password string: `$1$NSESuz4A$22uWH1mOPnka4zTdnx3jx1`. The password is further divided into subfield separated by dollar signs. Let's lookt at each one:

`1` - Tells us the MD5 hashing algorithm is used for the hash. `NSESuz4A` - Is the salt used with the password for the algorithm. `22uWH1mOPnka4zTdnx3jx1` - The actual hash of the salted password.

As you can see, the salt is 8 characters in length. On Unix-like systems, the salt uses a character base of `[a-zA-z0-9./]`. This means there are a total of 64 possible characters that each element in the salt can be. So, the salt added with the password could produce 64^8 or 281,474,976,710,656 total passwords.

John the Ripper

- http://www.openwall.com/john/
- Available for Windows, Mac OS X, GNU/Linux and UNIX
- Free of charge
- Free and Open Source Software
- Wordlists available in 20+ languages containing 4 million entries
- Supports multiple processors

John the Ripper is a utility for cracking passwords. It takes a wordlist and an unshadowed password dababase file, and attempts to recover the password by hashing the entries in the word list and matching them to the entry in the unshadowed password file.

As we already discovered, the shadowed password contains a bit of information. It shows is the hashing algorithm used to create the hash, it gives us the salt that is combined with the password, and of course it gives us the hashed password.

When john gets an entry from the wordlist, it looks for the hashing algorthim it needs to use on this account, grabs the salt, combines the salt and the wordlist entry together, hashes the combined result, and compares that hash to what is in the unshadowed file. If the hash matches, then we have found our password. If it doesn't match, then we continue working our way through the wordlist in a like manner until we find a match.

If we exhaust the word list and haven't found a match, then john will go into incremental mode, meaning it will start with 'a' then 'b' through 'z', then try 'aa', 'ab' through 'zz', then 'aaa' etc until a match has been found.

John can take advantage of a multiple CPU/core system. It's trivial to have one CPU/core work on one wordlist, a different CPU/core work on a different wordlist, etc. Hashing the text is the most computationally intensive operation, so you could have different CPUs/cores working on hashing texts of different lengths. John is very configurable, with many more options, to speed up the process for searching for passwords.

Wordlists can be obtained from the developers of John the Ripper. Relatively small wordlists are given away for free, while a nominal charge is required for larger wordlists. Other wordlists aronud the Internet could be found, the largest of which might contain 2-3 million entries. Of course, your operating system likely already ships with a wordlist that spellcheckers use as their dictionary database. Check the documentation for your operating system for more information.

Quiz Time - Strong vs. Weak

1. `chameleon`
2. `RedSox`
3. `B1gbRother|$alw4ysriGHt!?`
4. `deer2010`
5. `l33th4x0r`
6. `!Aaron08071999Keri|`
7. `PassWord`
8. `4pRte!aii@3`
9. `passpasspass`
10. `qwerty`

The word `chameleon` is a very weak password, because it's based on a word out of the dictionary. Dictionary words are usually the first passwords guessed.

While `RedSox` is technically not a "dictionary word", because it is two dictionary words combined together, this is

another example of a weak password as it can be guessed at very high speeds.

`B1gbRother|$alw4ysriGHt!?` is an example of a very strong password. It uses random capitalization, uses inconsistent character and letter substitution and it is extremely long, with 26 characters. It's based on a passphrase rather than a password.

`deer2010` is another example of a weak password. Words, especially dictionary words, with numbers appended at the end can be easily tested with very little lost time.

The word `l33th4x0r` is a medium-weak password. First off, it's common in the hacker and geek community to refer to themselves as "elite hackers", or "l33t h4x0rs". Also, the substitution of characters for numbers and punctuation is not random, but consistent. "l33t" dictionaries can easiy be created from standard dictionary wordlists.

The word `!Aaron08091977Keri|` is a medium password in terms of strength. It's not based on a dictionary word, and it incorporates length, a combination of uppercase and lowercase letters, numbers and punctuation. However, if the couple is being targeted for their passwords, this string could be generated without much loss of time.

Common strings such as `PassWord` are horrible passwords, and can be easily guessed in mere seconds if appropriate hardware is used to crack the password. Even with the capitalization of a couple of letters.

The word `4pRte!aii@3` is an example of a very strong password for a few resions. First, the appearance of the letters, numbers and punctuation appears to be entirely random. Also, notice the `ii` in the password. Using a letter twice in succession helps avoid "shoulder snooping", where someone over your shoulder might be trying to figure out your password based on where your fingers land. Because it is very difficult to see multiple presses of consecutive letters if typed quickly, this can add a great deal of strength to the password. It's best to not repeat the letter more than twice, however.

Regardless of the string, repeated characters over and over in the password give a very weak password, such as `passpasspass`. Repeated strings can be concatenated quickly from large wordlists, and cost the attacker very little.

Using sequential strings from a keyboard such as `qwerty`, are weak passwords as a wordlist can quickly be generated that contains such strings, in many directions (we'll cover this in a bit, actually).

Passwords From Weak to Strong

- Dictionary words
- Number appended
- Predictable sequences (from keyboard, etc.)
- Predictable "l33t" speak
- Personal data
- Mnemonics (`BBslwys90!?`)
- Random base-95 strings

Some points to address in this list. Obviously, we covered why dictionary words are a bad idea, words with numbers appended, predictable sequences, such as from the keyboard or repeating characters and words in the password and even predictable "l33t" speak doesn't award you much strength.

The only item in that list that would deserve some mention are Mnemonics, such as `BBslwys90!?` from `B1gbRother|$alw4sriGHt!t?`. Notice that we substituted "90", a right angle, for the word "riGHt". If used correctly, this creates a random string of characters that are meaninful to you, but the attacker would not be able to guess. These need to be used with care, as common phrases turned to mnemonics could be easy to guess.

Entropy

- Total possible number of states a password can be in.
- Represented in base-2.
- Increasing the entropy of a password increases its strength.
- Defined as: `H = L*log2(N) = L*log(N)/log(2)`
- H = number if bits in base-2
- L = length of the message
- N = number of possible symbols in the password
- See table in handout

Entropy comes from information theory, where entropy is a measure of the uncertainty of the random variable. In essence, entropy quantifies the expected value of information contained in a message.

For example, a fair coin has a entropy value of one bit. However, if the coin is not fair, then the expected value is lower, due to the uncertainty being lower. The entropy of a coin flip is given by the binary entropy function.

Calculating entropy of a given password is given by the function:

```
H = L*log2(N) = L*log(N)/log(2)
```

where H is the resultant entropy of the password given in binay bits, L is the length of the password and N is the number of possible symbols in the password.

For example, the password `BBslwys90!?` has a length of 11. It also uses characters from the lowercases character set, uppercase character set and the number and "special character" character sets. So, N=94, in this case. Thus 11*log2(94)=72. This password has an entropy of 72 binary bits.

What this means is that a brute force password cracking utility would have a search space of 2^72 or 4,722,366,482,869,645,213,696 possible passwords to search through for a 72-bit entropy password. Of course, understanding probability means that the utility won't have to search every password in the search space. It should stop when the password is found, even if there are more passwords remaining.

Consider the following table:

Entropy (H)	Numbers	Alphabet	Alphanumeric	All ASCII characters
32	10	6	6	5
40	13	8	7	7
64	20	12	11	10
80	25	15	14	13
96	29	17	17	15
128	39	23	22	20
160	49	29	27	25
192	58	34	33	30
224	68	40	38	35
256	78	45	43	40
384	116	68	65	59
512	155	90	86	79
1024	309	180	172	157

Looking at the table above, the "Entropy (H)" column shows the desired bit strength that you wish your password to have. For example, suppose you wanted your password to have a bit strength of 80. Then, if your password consisted of only numbers, it would need to be 25 digits long. If you wanted your password to consist of all chracters from the entire ASCII character set, then you would only need a password of 13 characters for 80 bits of entropy.

Quiz Time - Strong vs. Weak Revisited

1. `chameleon - 38 bit`
2. `RedSox - 34 bit`
3. `B1gbRother|$alw4ysriGHt!? - 164 bit`
4. `deer2010 - 40 bit`
5. `l33th4x0r - 54 bit`
6. `!Aaron08071999Keri| - 125 bit`
7. `PassWord - 46 bit`
8. `4pRte!aii@3 - 72 bit`
9. `passpasspass - 56 bit`
10. `qwerty - 28 bit`

Revisiting at our passwords above, we can see the entropy bits of each password. This should help put into perspective the relative strength of each password.

Remember however, that bits doesn't necessarily mean that your password can't be easily found given dictionary attacks, or other means as we discussed earlier in this presentation. Randomization from any of the 95 possible characters is important.

Also, notice that just adding characters sets doesn't provide a great deal of entropy. Entropy comes most from length, due to the equation

```
H = L*log2(N)
```

Length in your password is more important than anything else.

Putting Entropy Into Perspective

- http://distributed.net
- Cracking 72-bit entropy key
- $1000 for the winner
- 1,385,178,353,379 keys per second
- ~100 years max to completion

My personal opinion would be that your password should contain at least 60 bits of entropy. This will provide enough entropy to make your search space large enough to frustrate most attackers, even with very dedicated hardware, or a distributed attack.

However, as computing strengthens and newer, faster algorithms are discovered, this isn't enough. Eventually, you well need 72-bits of entropy then maybe 80-bits. So, the question remains, as time goes on, how can you manage passwords with this much entropy?

One Implementation

- http://passwordcard.org

- Uses an hexidecimal number for generation/regeneration
- Print and store in wallet/purse
- Use passwords provided on card
- Remember column/row, direction, length

The password card from http://passwordcard.org is a way to generate strong passwords with high entropy, make the password easy to remember, have a unique password for each account that you have, and store the passwords in a secure place; namely your wallet or purse.

When you visit the site, you will be presented with a form at which you enter a valid hexadecimal number. It doesn't have to be something complicated. If you recognize that the integers 0 through 9 are valid hexadecimal numbers, then a number such as '42' would works just fine. Also ,the letters a through f are valid hexidecimal numbers, so something like 'dead', 'beef' and 'cafe' are all valid hexadecimal numbers as well.

The point of the hexadecimal number is just in case you lose the card, you can regenerate it with that number. This means that there exists only one card for each number provided. The valid input range for the hexadecimal number is anything from 0 to ffffffffffffff, which means you could generate up to 295,147,905,179,352,825,856 password cards.

Once the card is generated, print it off, laminate it, and throw it in your purse or wallet. Now, when you need a password for creating an account, or wish to change existing passwords, pull out the card, find a symbol column and row color that will help you remember the starting location for the password, and go. Pick your destination and password length. Then, everytime you need the password, just pull out your card, and type it in. Eventually, if you use the password often enough. you'll begin memorizing the password.

A Sample Card

Find Every Password on the Card

Given the following restrictions, can we get a total number of passwords that the card can generate?

- Password must travel in straight lines only

The password must travel in straight lines only. It's not allowed to make any bends, turns, zig-zags, spirals, etc. The password must travel in exact straght lines.

- Password can travel horizontally, vertically or diagonally

We'll let the password travel one of 8 directions: up, down, left, right or any of the 4 diagonal directions, traveling at 45-degree angles from the cell.

- When the password reaches the edge of the card, travel stops

When the password reaches an edge of the card, the password travel must stop. It's not allowed to rebound off the wall in any manner, and it's not allowed to wrap around the card, similar to a pacman game. Reach an edge, stop the travel.

- Given these restrictions, how many passwords exist?

A Method of Attack

- For a m-rows by n-columns card

We'll assume a card of arbitrary size with m-rows and n-columns. we'll use m and n as our variables throughout our calculations, and in the final result. Of course, it probably makes sense to look at the trivial cases first, keeping the card small, and see if we can find closed formulas for the larger cases.

- Look at vertical travel first

Looking at vertical travel, let's start with the trivial case first, beginning with a `1x1` card. Obviously, there is only one password, which is a single character in length. Not too exciting, so let's expand our card.

Looking at a `2x1` card, it's clear that there are 2 passwords with length of 1. We then have a password of length 2 that travels down and another password of length 2 that travels up. So, for a `2x1` card, we have 4 total passwords.

Looking at a `3x1` card, again, it's clear that there are 3 passwords with length of 1. We then have two passwords of length two and one password of length three, all of which are traveling down. We have an additional 3 passwords, two of which have length two and one with length three. So, for a 3x1 card, there is a total of 9 passwords that can be found.

Noticing a pattern, it seems that for m rows, the number of passwords that can be generated in that column is m-squared. If we continue our thinking for a `4x1` card, we will indeed for 16 passwords, and for a `5x1` card, we'll find 25 passwords. So it's working out. The only thing left is to multiply the number of columns to our m-squared.

So, I propose that for an `mxn` card, there is exactly `n*m^2` passwords traveling in the vertical direction.

- Look at horizontal travel next

Using an analogous method for the horizontal travel, it becomes clear that for a `1xn` card, the number of passwords on that row can be fonud by squaring the columns. Because there are m-rows that we need to count, the number of

horizontal passwords that can be found can be represented by `m*n^2`.

Thus, I propose that the number of passwords that can be generated both in the vertical and horizontal directions can be represented with:

```
n*m^2 + m*n^2
```

However, if you're paying attention, you'll notice that we counted each single-character password in the vertical direction, then counted them again in the horizontal direction. So, we have the single-character passwords counted twice. So, we need to adjust our equation to take this into account. This can be given by subtracting off `m*n` passwords. So, the total number of passwords that can be found in the vertical directions and the horizontal directions is given by:

```
n*m^2 + m*n^2 - m*n
```

- Look at diagonal travel last

Looking at the diagonals is going to be a bit more challenging, but we should be able to tackle it. Let's start by counting the number of passwords that start from the lower left, and work their way to the upper right. Let's start with the left most column, and work our way to the right most column.

we already counted the single-character passwords, so we don't want to count those again. So, rather than counting the the first cell in the first column, we'll start with the second cell in the first column.

Moving up to the right, we find there is only one password of length two.

Now start in the third cell of the first column, and work your way up to the right. We find that there are three passwords: two of length two, and one of length one.

Moving to the fourth cell of the first column, and work your way up to the rigth. We find now that there are six passwords: three of length two, two of length three and one of length four.

Continuing in a like manner, we find the sequence:

```
1 3 6 10 15 21 ...
```

Doing a bit of mathematics, we find that we can represent the sequence with:

```
i*(i+1)/2
```

However, we need sum up each element in the sequence. We started in the 2nd row of the first column, so where do we stop? Do we sum up all the diagonal passwords that start in every row in that column? Well, if we stop at the second-to-last row, then we'll save us some work later.

So, given a row with m-elements, we want to sum `i*(i+1)/2` from 1 to `m-2`. Also notice that not only do these passwords exist in the upper left of the card, but they are duplicated in the lower right of the card. So, counting all those passwords can be represented with:

```
2*Sum(i*(i+1)/2 from i=1 to m-2)
```

Which can be rewritten as:

```
Sum(i*(i+2) from i=1 to m-2)
```

Evaluating this expression gives the following closed form formula:

```
m*(m-1)*(m-2)/3
```

Now, the only diagonal passwords left to evaluate are the passwords that start at the bottom of the card and travel to the top of the card. These passwords will continue with our previous sequence, but we need to make an adustment.

These passwords will travel the full number of columns in the card minus the number of rows. So, rather than our previous index starting with the second column, we need to it to start at the first. Reindexing will give us the following formula for the number of passwords traveling diagonally from the bottom of the card to the top:

```
i*(i-1)/2
```

The only question is: how many of these are there? Well, for a `3x3` case, there is only one. For the `4x3` case, there are two. For the `4x6` case there are three. It seems that there are n-columns minus m-rows plus 1 extra, or:

```
n-m+1
```

So, putting this together, that means there are the following diagonal passwords traveling from the lower left to the upper right:

```
m*(m-1)*(m-2)/3 + m*(m-1)/2*(n-m+1)
```

However, we only went one of four directions for the diagonal travel. This means that there are a total of:

```
4*[m*(m-1)*(m-2)/3 + m*(m-1)/2*(n-m+1)]
```

Putting it All Together

- The total number of passwords:

    ```
    4*[m*(m-1)*(m-2)/3+m*(m-1)/2*(n-m+1)]+m*n^2+n*m^2-m*n
    ```

- Simplified:

    ```
    m/3*(-2*m^2+9*m*n+3*n*(n-3)+2)
    ```

- Our 8x29 card gives 11,264 passwords

Only 11,264 passwords exist in our card given the restrictions that we placed on the card. This is an extremely small search space for passwords. It would be trivial to develop a program that could generate a wordlist containing all the passwords with this restriction for the card, to aid in cracking the password.

- Conclusion? Don't place these restrictions on your passwords!

Moral of the story? Don't use these restrictions!!! Instead, zig-zag around the card, bounce of walls, travel in spiral directions, wrap around the card like pacman, etc. When you do this, you increase the search space dramatically, and make it feasibly improbably for a programmer to create a wordlist of all the possible passwords that you could have.

Conclusion

- Pick passwords with at least 60-bits of entropy
- Use a password card
- Create unique passwords for each account