Ball and Chain: Hashing is Dead, Long Live the Password

Abstract

Ball and Chain offers the possibility of perfect security for even the weakest of passwords. For far too long, password breaches have plagued the information technology industry. For far too long, the strength of a user's credentials has been dependent on the complexity of the plaintext password. For far too long, algorithmic cost has plague the adoption of slow hashing algorithms. Ball and Chain makes use of a vulnerability in the process of a malicious attacker. Ball and Chain can make password breaches a thing of the past. Ball and Chain can make password policies a thing of the past. Ball and Chain can push us towards a safer, smarter, lighter web.

1. The Problem

Ever since the virtual dawn of the Internet, the security of our identity, systems, and information has been guarded by the venerable password. From "letmein" to "god" these secret sequences have protected the gates of the web from the chaos of a world without authentication. But what was once the golden standard has fallen low in recent years. It feels like not a month can go by without the headlines screaming about some new database breach in which millions of passwords are leaked onto the Internet. But I kid you not. We can solve this once and for all. Let's talk about the problem in detail. The methods that developers use to store passwords are intrinsically broken.

1.1. Plaintext or Encoded

If you store the passwords for all your users in plaintext or encoded with something like base64, you're gonna have a bad time. It still surprises me just how common these techniques are, even in today's decently paranoid climate. Obviously, if a malicious actor gains access to a database containing user passwords stored in the plain, they're not going to have any trouble reading, using, and disseminating those passwords to the world. This is simply, a terrible password storage mechanism.

1.2. Encrypted

You may occasionally happen upon an application that stores the passwords for its users in encrypted form. If done properly, encrypted credentials can be nefariously difficult to decrypt and make use of. That being said, encryption is quite difficult to implement correctly. From weak cipher usage, to poor implementation of block cipher chaining. If you mess up even a tiny bit, the attackers will easily find a way to discover the plaintext of the passwords from the access ciphertext. Beyond just a failing of the method by which the ciphertext is created the encryption of any plaintext data requires the storage of a plaintext key for later decryption. This key is often taken by the attackers and used to decrypt all of the stored passwords. Rendering them again, as plaintext.

1.3. Fast Hashing

The majority of modern applications call some type of fast hashing algorithm for the storage (and by extension) protection of user credentials. Hashing algorithms have risen to the top for their ease of use, as well as their strength when properly utilized. Since a hashing algorithm destroys the plaintext data that is sent into it at creation, the only way to find the plaintext password again, is to "crack" by way of brute force (dictionary, hybrid, etc..) the plaintext which was used to create the hash. If the plaintext used in the creation of the stored hash was of sufficient complexity, the attacker will not be able to determine what it was within a reasonable time frame (before the heat death of the universe). It is because of hashing algorithms (caveat: as well as unrestricted online attacks) that users are constantly bombarded with password complexity requirements. It is because of hashing that we emphasize the "strong" password. I want you right now, to briefly imagine a world without the need for complex or complicated passwords. A world in which the password "goat" would give you the same level of security as the password "nicklebackisthebestBand5ev4r!!!"; Because that is the world that we are working towards. Let's talk about a decent solution to the issue of password complexity next.

1.3. Slow (Adaptive) Hashing

Slow or Adaptive hashing algorithms exist for one simple purpose, to take a long time to compute. The process of going from a plaintext to a computed hash value is elongated in order to make it far harder for an attacker to uncover the plaintext value via some sort of exhaustive search; since now the attacker must work far harder for each and every guess. This is not a terrible solution to the issue of password complexity on the Internet today. That being said, this isn't the solution that you're looking for. For starters, if a user picks a terribly "weak" password; it can still be cracked with high certainty over a reasonable period of time. That is, password complexity still matters when dealing with these hashing algorithms, it just matters less. But wait, there's more. The use of slow hashing algorithms puts us in a bind. In order to be able to support infrastruc-

ture at scale with slow hashing algorithms, your application will need to pay in computation. You will need faster computers, which cost more money, in order to be able to keep up with the necessary computational requirements at the same speed as before. This is often why developers forgo any sort of adaptive hashing algorithm for a simple fast hashing algorithm. Why purposefully take more resources for the same exact work? Especially if your developers just simply don't expect that there will ever be a hack, or simply don't really care… etc. Furthermore, when we go to using slow hashing algorithms, we do not eliminate the threat. We simply raise the bar of entry. That is, a 12 year old on their laptop may no longer be able to crack all your passwords in a reasonable amount of time. But he can just post them up for all his friends. All he needs in order to one up you yet again, is more computational power. Heaven help you if your adversary is organized crime, or a nation state. When you use adaptive hashing algorithms, all you're really doing is entering into a computational war with your adversary. And yes, that's certainly a step in the right direction. But secure concepts should not be built to be "better than" or "good enough for now." Great security is about completely halting your adversary in his tracks. Great security, gives up no ground.

2. The Solution

I would like to humbly present my own original work to you as the solution to all of this. Ball and Chain presents us with the possibility of a password breach free future by entirely disrupting the ability for any adversary to perform an offline attack against any passwords stored in this manner. Before I get into the specifics of its operation however I would like to give a little background on how it is that Ball and Chain was invented.

2.1. The Backstory

For the longest while I have been thoroughly terrified by the nature of the information security industry. I don't think it's any secret that we are failing at securing the Internet in any meaningful fashion.

I don't write up this backstory as a pat on the back for myself. In fact, I'm not writing in detail at all. What I want to stop in the middle of the flow of this article for is far more important. Yes, I am quite proud of the work that I did in discovering this technique. But this is not about me. This is not about money. This is not about fame. This is a call to action. Forget exploitation frameworks. Forget sweet backdoors. Forget incredible exploits. Forget making a splash. The Internet, and by extension, the world, needs our help. We have a duty in this pivotal time to secure the future. We are the modern equivalent of mathematicians in the time of Pythagoras, and what we do today has the ability to literally change the world. Forget about thinking small. Forget about aiming for what you can hit. We need people who are willing to fail for a chance to win big. We need people who are capable of dreaming big, losing repeatedly, and maybe one day, winning big. We need people who are willing to give up on trying to look cool with the next clever zero day in some unknown protocol. We need people who are willing to work on the things that actually matter. People who are willing to solve problems, not puzzles. Puzzles are academic. Problems have real, kinetic weight. If the story of what I did here can help even one person to dream big, from the bottom of my heart, it will have been worth it.

I invented Ball and Chain in the middle of a panic attack while driving through Montana. Staring out the window, looking at the mountains towering over me. For over a year I had been dreaming of finding a better way to store passwords. I felt a bit crazy for even thinking that it was possible I could come up with something that might do better than anything ever invented before. But I kept thinking anyway. I kept working anyway. Before Ball and Chain, I had been stuck working on a theory of ephemeral entropy for a while. But it wasn't getting anywhere. Ball and Chain started as a simple idea, and quickly blossomed into something capable of storing passwords with infinite strength, something capable of completely removing the need for complex passwords, and something capable of storing ordered data (credit cards) in the same way (via a permutation of the algorithm) with the same level of secrecy.

2.2. High Level Overview

Here's a high level view of how Ball and Chain works. It is first important to note that the reason it works at all is because it is quite literally an exploit against a vulnerability in the attack chain of an adversary as they attempt to breach your network. Allow me to illustrate.

Imagine you are an attacker, attempting to steal some information (like a credit card) from the corporate network of Company-X. Imagine you have shell access to the machine on which the information is stored. That's usually not too hard right? Data Loss Prevention is actually quite difficult. Keeping someone from being able to sneak information out of your network is quite a task. So in this scenario, if what you're after is something like a credit card, you'll probably be able to get it. Maybe you simply read it via your shell access, then take a screenshot of the shell on your screen. And voila, you have the information you were after.

But now imagine a second scenario. In this scenario, your job is not to steal one single credit card from my network via your remote shell. Instead, I want you to steal 100 trillion trillion credit cards. That would be quite a different scenario wouldn't it? You couldn't simply copy and paste from your screen through. You definitely couldn't read all of them at once and take a screenshot. Trying to move them all in one file would take some work. And extracting them from the network via something like FTP would be next to impossible considering Company-X only has a finite upload speed onto the Internet. In all reality, you will fail this test. You will not get all the credit cards.

And here you have the simple explanation for what Ball and Chain is. Forget credit cards. We create a huge array of random data. We tie user authentication via the venerable "password" to this titanic array of data. That is, you cannot tell whether or not the proper password was typed in, unless you have access to the gigantic array of random data. If the array is big enough, you'll never be able to move it. You certainly won't be able to move it without being detected and thwarted.

Simply put, if you need the array to be able to even make a guess as to what the password for any user is; and if you cannot get that array (because the array is too big to move out of the network). Then you cannot even make a single guess as to what any user password may be. Offline attacks will become obsolete.

2.3. Technical Rundown

Okay, so lets talk about one model by which you may do this. Ball and Chain is still a work in progress, and so this outline demonstrates a viable model of production by which Ball and Chain may be implemented in code, nothing more, nothing less. There is still a lot of work to be done (see "future direction"). Let's go through the steps of one implementation.

2.3.1 Building the Array

You need to create the huge array of random data first. You can use either true random, or a cryptographically secure pseudo random number generator. If you choose to use pseudo random, make sure to throw out the seed that you used to create the array. You want to be absolutely positive that nobody can recreate the array from something like a seed, which would be easy to sneak out of your network.

The next question to ask is: how large should you make the array? Well, the simple answer is, "it depends." It depends on the specific layout of the application for which you are implementing the Ball and Chain password representation format. Here are the two things to look at.

One, what is the upload speed on your outbound pipe. That could be the connection between you and the Internet. That could be the connection between you and another potentially hostile network. Wherever that pipe is, the rate at which data can be siphoned off through it should determine the size of the array. For example, say that Company-Y has an outbound pipe of 6Mbps, an array on the size of 2 Terabytes would take a month to extract from the network. That is, using the entire upload for an entire month. Of course, this would be ridiculous, and nearly impossible to pull off, without being detected and thwarted. And you can only imagine what would happen if the adversary attempted to download all the data in a more stealthy manner. It might take decades.

The second thing to keep in mind when deciding the size of the array is what kinds of difficulty you would like an adversary who does somehow manage to get their hands on the array, to have in moving it around. For this, imagine that the attacker does somehow manage to get the array. Perhaps they physically break into your datacenter and take the hard drives on which the data is stored. How much difficulty would you like them to have in working with and disseminating the data? Basically, just make sure that the array has a minimum size large enough to cause trouble if the attacker wants to say, put the data in the cloud, or send it to all his friends to distribute the cracking. Because yes, with Ball and Chain, even if they do somehow get the array (which should be basically impossible) then they still have to crack the passwords contained within.

So, pick the size of your array. It should basically be as large as you can make it. Calculated for your specific network. Then fill the array with securely generated/sampled (pseudo)random data.

2.3.2 Creating a Password "hash" (representation)

The storage of passwords via the Ball and Chain mechanism is actually surprisingly straightforward. Assuming we already have our array of random data. Here is the first part of how we tie authentication to that array. When a user account is created, you must calculate the password representation by following something similar to this algorithm. Select for your purposes, some cryptographically secure block cipher (or stream if you are so inclined). I am currently partial to AES-128 in CTR mode. This does take what was formerly a block cipher and turn it into what is in effect, a stream cipher. But the operation of the cipher is still the same. Just with one added portion, the nonce. The nonce func-

tions external to the application in the exact same way as a salt would have under a traditional hashing algorithm.

Here's what we'll do.

1. Create a random nonce. This nonce will be stored and treated just like a salt. It will be supplied to the cipher when requested upon encryption and decryption of the linked ciphertext.
2. Go into your array of random data and select from it a multitude of pointers. That is, specific locations within the array, down to the bit. It is important that you select more than one. The current recommendation that I have to offer is somewhere on the order of ten. However, that is in no way optimized.
3. Pad each pointer to be the same length.
4. From each and every pointer you previously selected, take a sampling of the random data at that point within the array. That is, if you have ten pointers all pointing to random locations within the array of random data, pull some X number of bits from the data immediately following each selected location. (The amount of entropy to pull from each pointer is up to you. It should be great enough so as to avoid collisions. My current recommendation is something between 32 and 128 bits.)
5. Take all the pointers and concatenate them into a single stream. It is very important at this point to make sure that the pointers as data are indistinguishable from random.
6. Take all the random data sampled from each referenced location within the array, and concatenate them all together into a single stream. Feed this stream into a hashing algorithm such as sha256.
7. Take the concatenation of all the pointers, and the hash resulting from the concatenation of all the sampled data. Concatenate these two into a single stream. At this point, the stream you are holding should appear completely random.
8. Now grab your encryption algorithm. Take the plaintext password that the user supplied when creating their account as the key to encrypt the stream of random data. (Don't forget to supply the nonce.) Make sure that the data is random as a stream when provided to the encryption algorithm. You can't hand it to the algorithm in the form of a string, or base64 encoded or something. It has to be true random within the character space. (I'll make sure to explain the emphasis in a second.)
9. Take the generated ciphertext, and store it as the user hash. (Along with the nonce as salt of course.)

2.3.3 Authentication

1. The user types in their password.
2. Take the stored "hash" and nonce. Use the password supplied as the key to decrypt the "hash" with your encryption algorithm. (Don't forget the nonce if using CTR mode.)
3. Take the resulting plaintext. And split the stream back into it's corresponding parts. The individual pointers, the hashed form of the sampled data.
4. Go into your array of random data. Go to each and every location pointed to by the pointers. Sample the data at each of those locations just as you did before.
5. Take all the samples, and hash them in the same order as you did before, with the same hashing algorithm.
6. Compare the two hashed streams (the one that came from the ciphertext and the one you just generated.) If they match, we know with very very very high probability that the proper key was used to decrypt the ciphertext.
7. This means, that the user inputted the proper password. Authenticate the user.

So what happens when an invalid password (key) is used? By the nature of a cryptographicly secure encryption algorithm, you should get some completely random output. This is why it is so important that the plaintext we use is completely indistinguishable from random. We don't want there to be any indicators in the decryption as to whether or not the right key was used. We want the only way to confirm the use of the proper key, to be via access to the array of random data. The array that should be too large for any adversary to ever get their hands on. The "hashes" that we build can still be stored in a database just as before. They're just a simple small ciphertext. What's lovely about them though is that they're meaningless without the array. If you've properly implemented the algorithm, you could post your "hashes" publicly on the web. Without the array, nobody will ever figure out what's inside (what key was used).

2.3.4 Some Known Pitfalls

I really cannot stress enough just how theoretical Ball and Chain is. The logic is sound. It will work. But I am not a cryptographer. I'm not a mathematician. I'm a crazy, idealistic, dreamer of an undergrad. There may be things about this algorithm that could render it useless in its current form that I am just not noticing. I still have quite a bit of work to do before I'll feel safe telling the world that we have one "specific", "secure" implementation. But here are some things to keep in mind when thinking about this algorithm.

1. This method only protects against offline attacks. You're still on the hook to implement a good mechanism for mitigating online password attacks.
2. Passwords can still be stolen from memory, or via phishing, or with a keylogger… etc.
3. The data taken from each point in the array must be of enough entropy so as to mitigate the possibility of collisions.
4. The number of pointers used in a single authentication must be carefully constructed to ensure that the attacker cannot get away with stealing a small portion of your array. That is, say you only used one pointer per hash. But you had 1000 users. If the bad guy steals one tenth of your array. He should have the ability to determine the passwords of one tenth (100) of your users. Because that one tenth of users have a single pointer, pointing within the one tenth of the array that he stole (approximately). The more pointers that you use, the greater the probability that he cannot get away with stealing a tiny portion of the array.
5. Make sure to hash the sampled data before encryption when creating the password representation. Here's why. This ensures that the attacker needs access to every part of the array referenced by a pointer before he can check to see if the two hashes match. If you stored the sampled data in plaintext, due to the length of the sampled data being great enough in entropy as to avoid collisions, the probability that any one pointer-data pair coming back as "true" and the used key not being the right one, would be astronomical. That is, if you gave the sampled data in the plain within the ciphertext, confirming just one of them would confirm the key. Which is actually even worse than just using one pointer data pair.
6. Attackers can still make guesses against the array via some approved portal. For example. If you have an application that accesses the array over the network, and the attacker compromises the application, he can make guesses against user passwords online as the application. This however, isn't a huge deal. Here's why. Yes, you should build defenses to detect and thwart any odd chain of authentication failures. That should be in your design from tip number one. It's still on you to stop online attacks. That being said, if the attacker has access to a machine that the passwords flow through before or during authentication, he can just take them from memory. This isn't the biggest threat.
7. Man In The Middle. If you choose to store your array at a distance from your application, and access it over some network medium, I advise protection against any form of network level interception or interdiction. SSL is the obvious solution. However it is also important to think about what data the array gives up when queried. Would you want the array to return the data from the locations referenced? Or would you want the array to simply reply back with a true/false to whether or not to authenticate the user? It really depends on your specific architecture and threat landscape.

3. The Future

There's a lot of work to be done. But the future of Ball and Chain is as bright as it gets. We can stop the rising tide. We can make offline attacks a thing of the past.

3.1 In Code

We do have a prototype/proof of concept written in python that demonstrates the viability of the Ball and Chain concept. It is far from cryptographically secure at this time. My hope for the future, is if I can get the resources to accomplish it, I would like to build an open source, public domain, library. Something akin to OpenSSL, which could be quickly and easily implemented in any project.

3.2 Storage

I get asked quite often, "can Ball and Chain store things like credits cards?" Yes, absolutely. I have a permutation of the Vanilla algorithm, which can be used to hide any sort of ordered data within the array. The data can only be unlocked with access to the entire array; and only with the proper password. Again providing the exact same level of security (theoretically infinite) for

the storage of sensitive information like credit cards, social security numbers, etc.

3.3 Further Research

I have also spent quite a bit of time looking into ways in which the concept behind Ball and Chain can be used in other arenas within out field. Suffice it to say, I do believe there are quite a few possibilities open for us in the future.

Additionally, I have been looking into ways in which the random array model can be used as a sort of mathematical playground on which to build never before envisioned security tools.

3.4 The Roadblocks

Money, time, influence, I am a poor, young, unknown, undergrad. I don't have the resources at this time to present this password representation format to the virtual masses as a viable alternative to password hashing. I don't have capability currently to build and cryptographically proof a secure library. More than anything else, I need funding, and a platform from which to present this idea to the world.

4. The Promise

Ball and Chain, completely precludes the possibility of offline attacks against your stored passwords with near certainty. When coupled with a reasonable mitigation of any potential online attacks the need for complex passwords disappears. No longer will users be pressured into creating passwords they cannot remember. No longer will password policies plague the industry. We will literally be able to create, and use reasonable passwords again. Forget the ridiculous modern password. Ball and Chain will fix all of this for us.

Will phishing attacks still exist? Of course they will. Will it still be possible to steal a password from memory? It absolutely will.

But the complexity of the password has absolutely nothing to do with these things. It's just as easy to steal 'golf' from memory as it is to steal something crazy like 'SooperSecret4523#$P33sward'.

Ball and Chain will completely revamp the playing field.