Password Cracking

Sam Martin and Mark Tokutomi

1 Introduction

Passwords are a system designed to provide authentication. There are many different ways to authenticate users of a system: a user can present a physical object like a key card, prove identity using a personal characteristic like a fingerprint, or use something that only the user knows. In contrast to the other approaches listed, a primary benefit of using authentication through a password is that in the event that your password becomes compromised it can be easily changed. This paper will discuss what password cracking is, techniques for password cracking when an attacker has the ability to attempt to log in to the system using a user name and password pair, techniques for when an attacker has access to however passwords are stored on the system, attacks involve observing password entry in some way and finally how graphical passwords and graphical password cracks work.

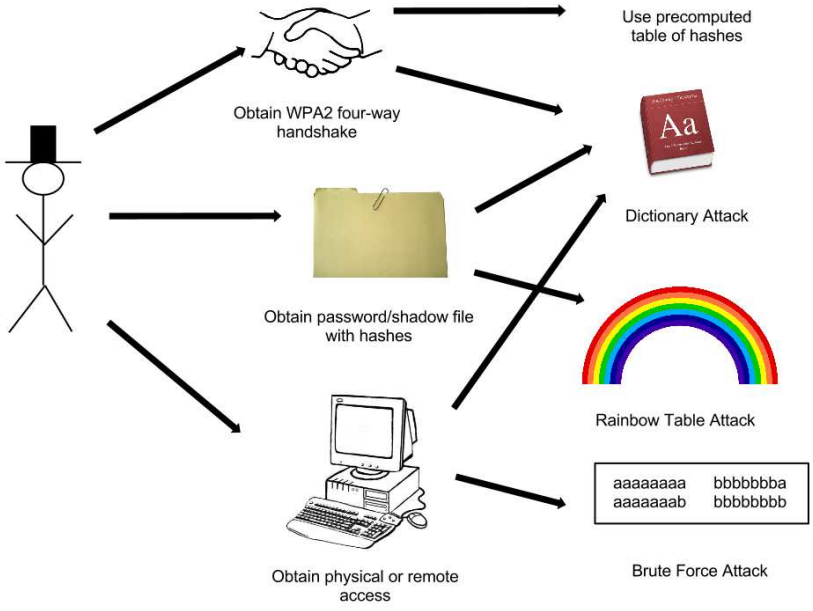


Figure 1: The flow of password attacking possibilities.

Figure 1 shows some scenarios attempts at password cracking can occur. The attacker can gain access to a machine through physical or remote access. The user could attempt to try each possible password or likely password (a form of dictionary attack). If the attack can gain access to hashes of the passwords it is possible to use software like OphCrack which utilizes Rainbow Tables to crack passwords[1]. A spammer may use dictionary attacks to gain access to bank accounts or other

web services as well. Wireless protocols are vulnerable to some password cracking techniques when packet sniffers are able to gain initialization packets.

2 How Passwords are Stored

In order to understand how to compromise passwords, it is first necessary to understand how passwords are stored on typical systems. Storing user names and corresponding passwords in clear text is an unacceptable solution. Attempting to hide passwords stored as clear text (such as putting the password file deep in a convoluted directory hierarchy) would amount to "Security Through Obscurity" which would also be unacceptable. The Unix system of file management provides a better solution though: one of permissions. Initial versions of Multics (the precursor to Unix) stored the password file in clear text, but only viewable with superuser permissions. This solution also failed when a bug switched some temporary files around and the password file (in clear text) was printed for every user upon login.

Unix, instead, stores the hashed value of passwords in the password file instead of the actual passwords. Then when a user inputs their password, the system can simply take the hash of the input and compare it to the stored hash value. To get an idea of how a password file is stored, let's look at the Unix password file scheme.

2.1 Unix Password File

On most Unix-based file systems the password file is located at /etc/passwd[8]. Each line in this file contains information about one account on the system. The file itself is readable by all users, but is only writable with superuser privileges. Each entry in this password file has seven fields, like so:

Figure 2: An excerpt from a Unix password file

The first field, "ender", is the account or user name. The second field contains the letter 'x', explained below. The third field is the user number, an integer identifier for the user. The fourth field is the group identifier, which shows the primary group of this user. The fifth field is a comma separated value list known as the Gecos field, information regarding the user's full name and contact information. In this particular case most of the fields are blank in the Gecos field. The sixth field is the home directory for the user. The final field is the program which should be executed when the user logs into the system; in this case the bash shell is used when ender logs in.

In early versions of Unix the second field for entries into the password file (the letter 'x') contained the one-way hash values for the passwords on each account. Over time those were moved to a shadow file (located in /etc/shadow), only readable by those with superuser privileges. This provides an additional layer of defense.

2.2 Windows Password File

The Windows system for storing the password file is similar to the Unix strategy. The password file for Windows, known as the Security Accounts Manager (SAM) file, is located in

`C:\windows\system32\config\sam.`

An entry in the SAM file contains seven colon deliminated fields: the user name, user number, encrypted password, hashed password, hashed password under a different algorithm, full name of user, and finally home directory. In contrast to the Unix password file, the Windows SAM file is not readable once the operating system has booted. The password file for Unix needs to remain readable for all users so that certain programs can access user information. Because of this, in order to read the SAM file, it is necessary to get access to it before the system has booted. Alternatively, people have installed secondary operating systems to read the SAM file from there.

2.3 Online Password Storage

Many websites and online services require users to log in with a typical password scheme. This necessitates the storage of password information. However, online services typically store passwords for their system in a non-standardized way, and these systems are not always designed by engineers with backgrounds in privacy or security. For instance, Sony had a security breach in which it was discovered that passwords were being stored in cleartext in a SQL database[4]. Password cracking for a system such as this only involves gaining access to the password storage system.

2.4 Password Salts

Storing the hashed or encrypted values for passwords is certainly much more secure than storing their plain text in a password file, but there is a common additional measure of security that can be implemented. This tactic involves adding additional randomness to passwords and is known as a password or cryptographic salt. When a password is created, the system will add some sort of randomness to the password. This randomness (or salt) should be different every time a password is made. Because of this, the exact same password will be stored as a different hash on different machines or different accounts on the same machine.

	Password	Hashed Value
No Salt	this1sAg00dPASSword!!	a5a5baa0c16166260e9ef8a48dbde112
Salted	6789o3uigtbgeat7this1sAg00dPASSword!!	53cffc58904a10b9dcc40345433862dc
Salted	v8734ihv6!nre432this1sAg00dPASSword!!	28b8f782262a890b4d730f8001d23bd5
No Salt	love	b5c0b187fe309af0f4d35982fd961d7e
Salted	12bg55tygsdf4gvi9yrdslove	65c96e15930d34dd9a9ce916b81fb044
Salted	879rughq2ebt5dfxcasedlove	a35436c0e0f2821db2703c1983a641ab

Figure 3: A table showing the md5 hash values for unsalted and salted passwords.

Figure 3 shows the effects of password salts on a strong password and a weak password. Without password salts, even a strong password will hash to the same value. This is a vulnerability in the event that an attacker gains access to the shadow file. An attacker could simply store what a few common passwords' md5 hashed values are and then be able to check if any passwords on a system match that hash. Salting causes this sort of attack to become much more difficult; the randomness added to the passwords causes a large amount of entropy in the hashed value, even for weak passwords.

3 Brute Force Attacks

The feasibility of brute force depends on the domain of input characters for the password and the length of the password[5]. Figure 4 shows the number of possible passwords for a given password length and character set, as well as how long it would take to crack passwords of that type.

	lower case	lower/upper	lower/upper/digits	lower/upper/digits/symbols
1	26	52	62	95
2	676	2704	3844	9025
4	456,976	7,311,616	14,766,336	81,450,625
8	$2.09x10^{11}$	$5.35x10^{13}$	$2.18x10^{14}$	$6.63x10^{15}$
16	$4.36x10^{22}$	$2.86x10^{27}$	$4.77x10^{28}$	$4.40x10^{31}$

(a) Password search spaces

	lower case	lower/upper	lower/upper/digits	lower/upper/digits/symbols
1	26 microseconds	52 microseconds	62 microseconds	95 microseconds
2	676 microseconds	2.704 milliseconds	3.844 milliseconds	9.025 milliseconds
4	≈.5 seconds	≈7 seconds	≈14 seconds	≈81 seconds
8	≈2.42 days	≈1.7 years	≈6.9 years	≈210 years
16	≈1.38 billion years	≈91 trillion years	≈1.5 quadrillion years	≈1.4 quintillion years

(b) Desktop cracking times

	lower case	lower/upper	lower/upper/digits	lower/upper/digits/symbols
1	9 nanoseconds	19 nanoseconds	22 nanoseconds	34 nanoseconds
2	241 nanoseconds	966 nanoseconds	1.373 microseconds	3.223 microseconds
4	≈163 microseconds	≈2.61 milliseconds	≈5.28 milliseconds	≈29.1 milliseconds
8	≈74.6 seconds	≈5.307 hours	≈21.6 hours	≈27.4 days
16	≈.5 million years	≈32 billion years	≈.5 trillion years	≈.5 quadrillion years

(c) ElcomSoft Co. cracking times

Figure 4: Password search spaces and how long it would take to brute force them

A desktop computer could attempt one million passwords per second when trying to brute force a password. Figure 4b shows how long it would take in the worst case for each of the input domains and password lengths. ElcomSoft Co. claims the ability to test 2.8 billion passwords per second using a high end graphics processor on a single machine. Figure 4c shows how long it would take to do the same tasks with a powerful graphics card and cutting age technology.

3.1 Dictionary Attacks

Brute force password cracking won't work for sufficently long passwords. Furthermore, passwords are very rarely actually random. Many passwords will be English words, perhaps with the first letter capitalized and a digit appended or prepended[9]. The search space for common or reasonable passwords is much smaller than $2.18x10^{14}$ for passwords of length 8 or less (even for passwords with upper case, lower case and digits available).

From the password data leaked from Sony, over fifty percent of passwords were less than eight characters long[4]. In addition, only four percent of passwords actually had lower case, upper case

and digits and only one percent of the leaked passwords contained a non-alphanumeric character. Some of the most common occuring passwords were seinfeld, password, 123456, abc123 and purple. Creating a dictionary that contains commonly occuring passwords like these would help an attacker guess correct passwords more often than brute forcing. For instance, a publicly available dictionary matched over 35 percent of the passwords leaked[2].

An attacker who gains access to the password or shadow file may not even need to actually try all the passwords in the dictionary (list) of common passwords. An attacker could precompute the hash value of each of those passwords and then simply match those to the hashed values stored in the password file.

This precomputation can potentially be a very serious threat to password security. However, as mentioned above, many systems implement a salt value that adds randomness to passwords for that machine. Now if an attacker would like to precompute hashes of common passwords, he or she would actually need to precompute those hashes for each password coupled with each possible salt. If the salt is sufficiently large, this can be a significant deterrence for the attacker, even for relatively short passwords.

4 Rainbow Tables

The idea of a time-memory trade-off in computer science is very common. The principle is straightforward: by precomputing some part of the problem (commonly either by solving subproblems or by finding common solutions), the time cost of solving the problem in general is decreased while the space requirements are substantially less than what would be needed to fully precompute the solution. This section details how a similar approach can be applied to the problem of password cracking, beginning with an innovation first described decades ago, and from there describing incremental improvements contributed which have led to the development of Rainbow Tables.

4.1 A Time-Memory Trade-Off

The idea of applying a *time-memory trade-off* to the problem of cryptanalysis was first proposed by Martin Hellman in 1980. Though his approach attacks a cipher text encrypted using the Data Encryption Standard (DES), it requires only minor modifications to instead attack a password hashing scheme. Given a precomputed table smaller than what would be employed in an attack

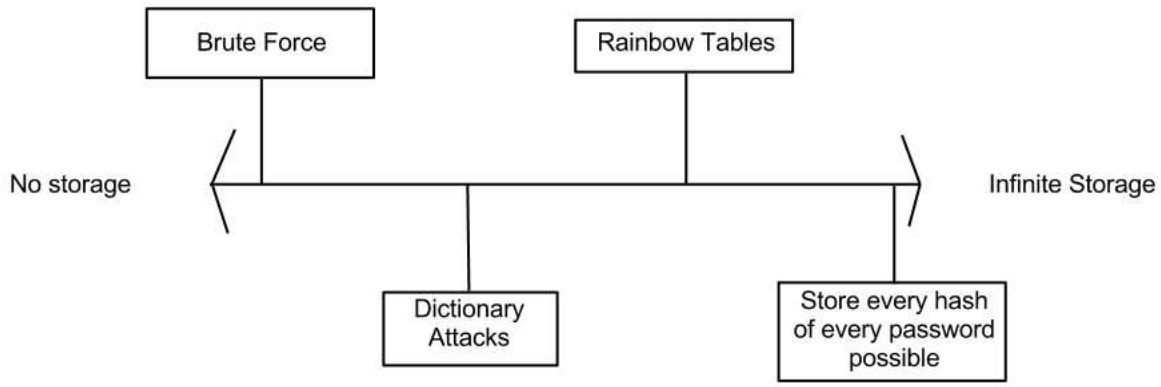
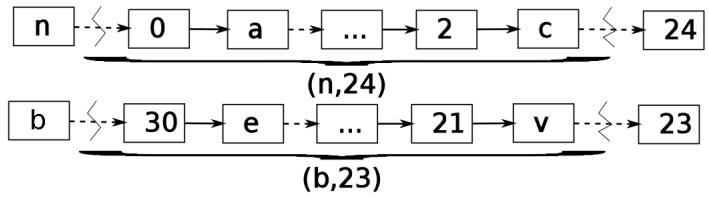
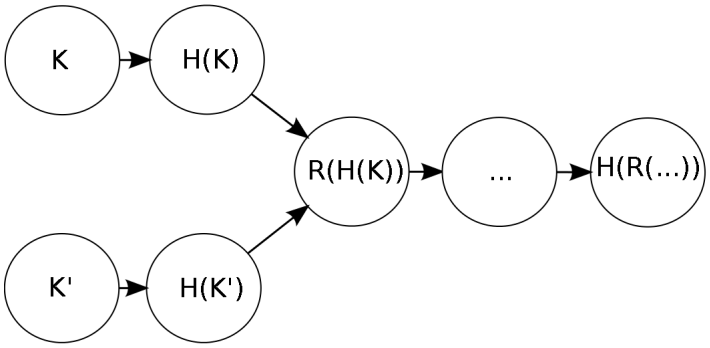


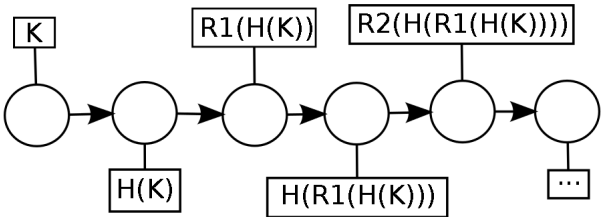
Figure 5: The spectrum of possibilities for password cracking attacks.



(a) Building a Hash Chain



(b) Merged Chains



(c) A Rainbow Chain

Figure 6: Building hash chains according to Hellman's original algorithm, two chains which have merged (storing almost entirely duplicate data), and a Rainbow Chain

where the hash for every candidate password was precomputed, Hellman's algorithm significantly reduces the search cost compared to a purely brute force attack [3]. The main innovation introduced by Hellman is the idea of using a *reduction function* which maps hashed output back into key space to produce chains of hashes and keys (the relationship between these functions is illustrated in figure 7). The full table is constructed by building a series of rows (the construction of two rows is illustrated in figure 6a). Each row begins with some random value (in this case n), which is hashed (producing the value 0 here). The hash of the starting value is then reduced (in this case producing the value a), and these operations are repeatedly chained together in this fashion until the row is of the length desired, terminating with some final hash value (in this case 24).

Note that the attack employed after computing the table (detailed below) requires only the start and end point for each chain, which means during precomputation all intermediate points can be discarded as they are used.

In order to attack a hash value, the hash and reduction functions are once again chained together; now, however, each hash value must be checked to see whether it is in the list of the table's row end points (see figure 8). In this case the hash being attacked is 13, and after several iterations of reduction and hashing, the hash value 26 is calculated, which is one of the table's end points. This end point's corresponding start point (which in the figure is s) is then looked up, and

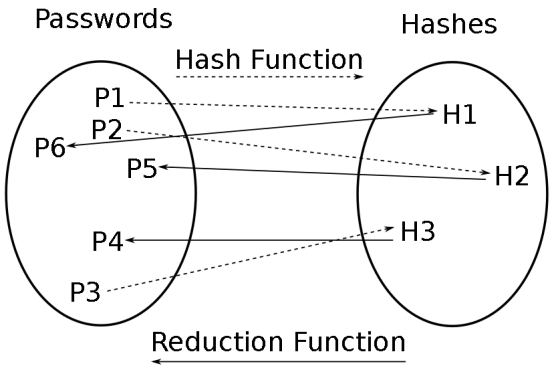


Figure 7: An illustration of the the hash and reduction functions used to generate tables for Hellman's approach

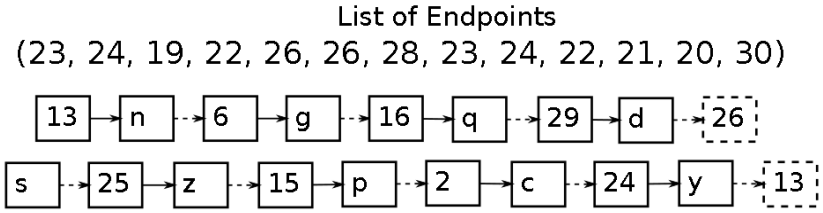


Figure 8: An illustration of the the hash and reduction functions used to generate tables for Hellman's approach

that row of the table is reconstructed (exactly as before) until either the end point's value (26 in this example) or the target hash (13 in this example) is produced. In this case, several iterations of hashing and reduction produce the value 13, and by examining the value immediately preceding it, the attacker can determine that the user's password (or at least a collision for it) is y. Had the end point's value been reached instead of the target hash value, the attacker would have continued building the original chain (which began with 13) until another end point was generated [3].

Hellman's paper uses tables of size $N^{2/3}$ (where N is the size of the search space for passwords) as a value which produces a table which will minimize the number of keys which miss the table completely, while still providing an appreciable increase in computation speed. Although this is a significant improvement, it is still insufficient when attacking all but the weakest passwords: the 16-character lowercase-only password from Table 4c could be cracked in around 123 hours (using the same value of 2.8 billion passwords per second from that table), but a password of the same length using both upper- and lowercase letters as well as numbers and symbols would still take over three hundred thousand years, and even at a severely reduced size, the lookup table would require nearly thirty petabytes of storage. Clearly, more improvement is needed to this approach if we plan to use it successfully.

4.2 Further Improvement

The innovation *Rainbow Tables* offer as compared to previous approaches is the use of a series of reduction functions (as opposed to a series of tables, each generated with a single reduction function). When the chains are generated (as in figure 6a), the position in the chain determines which reduction function is applied to each hash, whereas before the same reduction function was applied at every position [6].

This provides a significant advantage in terms of precomputation time: if a collision occurs, it degenerates into merged chains *only* if the collision occurs at the same position within the chain (since otherwise the next reduction function applied would be different and the chains would likely split again). While a Rainbow Table chain could conceivably contain a series of repeating values, the probability of this event occurring is incredibly low, as is the probability that two chains will contain a series of identical values of any appreciable length [6].

5 LanManager Hashes

An important point to make about Rainbow Tables is that, despite their greatly improved speed compared to previous such attacks, they are still rendered largely ineffective by the inclusion of a salt in the password hashing scheme. Why, then, are there many widely-available tools which use Rainbow Tables to crack the hashes stored in the Windows Security Accounts Manager (SAM) file? This widespread vulnerability is attributable in large part to the Lan Manager (LM) hash system used in older versions of Windows.

5.1 Usage

The LM Hash was the only password hashing protocol used on Windows systems until the launch of Windows NT; however, even after the introduction of newer hash schemes in later versions of Windows, it continued to be stored by default (for backwards compatibility) until the release of Windows Vista (which is why, as described in Section 2.2, the SAM file contains multiple hashes of

each password). Unless a sufficiently long password is used or this behavior is turned off, Windows XP installations still store this hash in the SAM file.

5.2 Weaknesses

There are multiple odd behaviors in the implementation of the LM Hash which render it more vulnerable to attack; one of the most significant is the fact that the hashes are not salted. This means that the same password will always produce the same hash on any Windows installation, which is what allows for the widespread availability of tools that use Rainbow Tables to crack passwords. Additionally, however, the protocol splits the password into halves and hashes each separately; this allows the two hashes to be attacked in parallel, in addition to enormously reducing the search space for an attacker (for reference, when dealing with 16-character passwords, this corresponds to moving from approximately .5 quadrillion years to just over twenty-seven days). The hash also converts all alphabetic characters to uppercase before hashing the password, which, while less significant than the fact that it hashes halves separately, still substantially reduces the search space (from about 2^{46} to about 2^{43} for a 14-character ASCII password).

5.3 Avoidance

The LM scheme cannot generate a hash for a password longer than fourteen characters; the simplest way to avoid its usage (in more recent versions of Windows) is to use a password which is longer than this. This is due to the fact that the two halves of the password are each used as DES keys which encrypt a fixed plain text; the space of 7-character ASCII passwords is encapsulated by the space of 56-bit keys. Additionally, when dealing with a system in which users cannot "reasonably" be expected to use passwords of sufficient length, versions of Windows which default to storing an LM hash for backwards-compatibility can have this behavior disabled.

6 Physical Attacks

Discovering the password to an account is clearly possible though direct attacks such as brute force, dictionary or rainbow tables. However, these are also far from trivial to implement, and there are serious time-space trade-off concerns. In certain situations it can be preferable to crack passwords through less traditional means. The most basic of physical methods would be to actually physically force someone to give up their password for a very important account. This type of literal attack is commonly referred to as "rubber hose cryptography" and is only effective when an attacker who can be very persuasive is able to gain physical access to a person who knows the password in question.

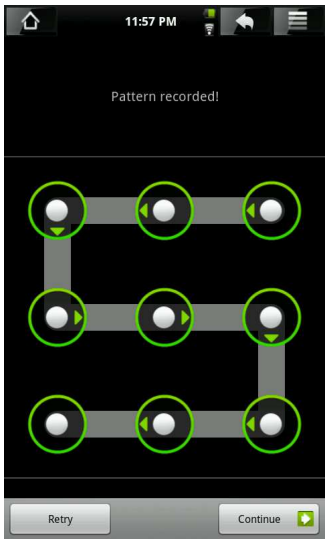
If there are moral or logistical issues with such an attack, there are less invasive ways to obtain passwords as well. A side-channel attack is one which gathers information that leaks out of the system in some way. The most basic form of this would be to observe someone typing in their password (perhaps with a well placed camera). With a tactic like this there is no need to look at password files, worry about salt values or create a chain of hashes; it is only necessary to be able to view the terminal in which someone inputs a password.

A more sophisticated form of side-channel attack would be to recreate a password from the sounds that typing on a keyboard generates[10]. Especially on older keyboards, each input key generates a slightly different sound upon being pressed. Researchers at Berkeley have utilized extremely sensitive microphones to decipher which keys have been pressed on a keyboard by just the sound. It's not even necessary to supply training data to the program (that is, data which informs the system which sounds correspond with which keys). The system is capable of listening

to ten minutes of normal keyboard input (such as someone writing a paper for a class) and reconstruct which sounds represent which keys using standard machine learning and speech recognition techniques.

In light of the multitude of techniques available for password cracking, it should be clear that there is no distinctly "best" option. The most useful password cracking technique will be highly dependent on the characteristics of the attacker and of the target.

7 Graphical Passwords



(a) A sample input for a graphical password.

(b) Extremely visible oil residue.

Figure 9: Android phone graphical passwords.

The advent of touch screen devices has brought about a new way of authentication that is commonly seen now on Android phone devices. These devices provide users with the option of authenticating using a graphical password, as shown in Figure 9. The default Android program requires the user to create a password which connects at least four dots in any order (though the user can make a longer password by connecting more dots). One benefit to graphical passwords such as this is that humans typically are much better at remembering patterns and pictures than attempting to remember a long string of random characters[7]. In addition, graphical passwords are difficult to attack in most traditional senses. Automating attempts at entry is difficult on a graphical password system, even if the search space is much smaller than typical text passwords. However, the Android graphical password scheme has been found to have a powerful security hole. Many users leave behind a visible oil residue from entering in their graphical password many times into the system. A stolen phone can potentially be broken into just by looking at the screen smudges.

8 Conclusion

In the preceding sections, we have discussed various facets of password-based authentication, including its origins, common implementations, and various attacks against it. It is worth noting, however, that the technology to create a password-storage scheme which will withstand any computational attack (within a reasonable time frame) has existed for decades. The prevalence of attacks against password hashes are attributable almost entirely to a combination of laziness (or ignorance) on the part of people who design and administrate systems which store passwords and laziness (or ignorance) on the part of people who make use of these same systems. Just as storing only salted hashes of passwords reduces the effectiveness of most computational attacks to the point where they are rendered nearly useless, the use of large passwords drawn from as large a search space as possible (i.e. including characters from as large a set as is available) greatly reduces the effectiveness of these same attacks as well. In addition, taking care when entering a password (i.e. not writing it on a post-it on your monitor, and checking for anyone who may be standing over your shoulder as you enter it) is sufficient to deter all but the most dedicated of would-be physical attackers (and greatly reduces the chance of such an attack being financially viable for the attacker).

References

[1] Anonymous. http://ophcrack.sourceforge.net/, July 2009.

[2] dazzlepod. http://dazzlepod.com/site_media/txt/passwords.txt, January 2012.

[3] Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[4] Troy Hunt. http://www.troyhunt.com/2011/06/brief-sony-password-analysis.html, June 2011.

[5] Gershon Kedem and Yuriko Ishihara. Brute force attack on unix passwords with simd computer. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, pages 8–8, Berkeley, CA, USA, 1999. USENIX Association.

[6] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *CRYPTO*, pages 617–630, 2003.

[7] Julie Thorpe and P. C. van Oorschot. Graphical dictionaries and the memorable space of graphical passwords. pages 10–10, 2004.

[8] David Wagner. http://www.nmrc.org/pub/faq/hackfaq/hackfaq-28.html, July 2003.

[9] Thomas Wu. A real-world analysis of kerberos password security. 1999.

[10] Li Zhuang, Feng Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. *ACM Trans. Inf. Syst. Secur.*, 13:3:1–3:26, November 2009.