Password Theory & Defeating Encrypted Filesytems

Author: Aaron Toponce
Email: aaron.toponce@gmail.com
PDF: http://goo.gl/QF8du
Date: 2011-09-10

Referenced Documentation

This presentation is presented as slides in standerds-compliant HTML using S5. This presentation references additional documentation that is not available in the slides themselves, but in the attendee handout. A compressed tarball of the ReST source, S5 HTML, Latex and PDF formats are available at http://aarontoponce.org/presents/olf2011/ohio.tar.xz.

License

  - The author's moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to the web page provided above or below.

The above is a human-readable summary of the license, and is not to be used as a legal substitute for the actual licnse. Please refer to the formal legal document provided here: http://creativecommons.org/licenses/by-sa/3.0/legalcode

Overview

- Password Theory
- Cryptanalysis of Passwords
- Defeating Encrypted Filesystems

These are some thoughts and ideas that I have been working on over the past few years. The goal of this subject is to provide a metric that people can use to identify the strength of their passwords

Cryptanalysis of Passwords: The subject will cover just a couple of the many tools that are used to defeat passwords. Some theory and metrics will be covered.

Defeating Encrypted Filesystems: Lastly, we'll learn the tools and steps necessary to defeat encrypted filesystems using the Cold Boot Attack and the Evil Maid Attack.

Preliminaries

- Physical access is assumed
- Getting root access on any Unix-like machine
- Goal: Education

This presentation and lecture will cover how do defeat passwords on any Unix-like operating system if you have physical access to the machine. We'll cover getting root on the box with little effort and time. After gaining root access, we'll discuss unshadowing passwords on the system using strong tools. We'll also look at defeating encrypted filesystems that are common on many latops and government furnished computers.

DISCLAIMER: This lecture is not intended to teach you how to crack machines for illegal reasons. Rather, the intent of this presentation is to teach system administrators and users alike how they can guard themselves against such attacks. As Stan Lee immortalized:

> "With great power there must also come -- great responsibility."

The attacks documented in this presentation are for informational purposes only, and the author holds no responsibility for the actions of his students.

Getting Root

- Physical access? Security is compromised!!!
- Runlevels - single user mode
- Interact with the boot loader
- Boot from a live medium (LiveCD)
- Flash the BIOS
- Chain cutters

It's important to understand that when you have physical access to a computer, any and all security has been compromised! It doesn't matter about the operating system- Unix, Linux, Mac OS X, BSD or Windows. If you have physical access, then you have immediate access to the data.

All that is required, is interacting with the boot loader, telling the kernel to boot the system into single user mode. Single user mode is necessary, and it drops the user to a root shell, without authenticating as root. At this point, the root filesystem has been mounted, and the data is at the mercy of the user at the machine.

If you decide to password-protect your boot loader, which is possible on many bootloaders, the attacker could just boot off another medium rather than the hard disk to get at the data. At which point, he could either mount the filesystem, and remove the password restriction, or chroot to the mounted filesystem, and start working.

You could protect your BIOS, by also password protecting it, so it only boots from the hard disk, and nothing else. This is common in universities, for example, who don't want students playing around with the system. The way around this, is to either flash the BIOS by moving jumpers, or pulling power. Or, because you have physical access, you could just remove the drives and clone them to another drive for further processing.

Then of course, you could have the case locked down under lock and key. All I need is my lockpick set and some knowledge picking locks. If you chain it down, I'll bring my chain cutters.

You see the point. If an attacker has physical access to the machine, your data is compromised. No ifs, ands or buts about it.

Or is there?

Some hard drives come with hard drive passwords, which if forgotten, cannot be reset, even by the manufacturer. They are hardware-based, not software-based. So, even if you pulled out the drive, and plugged it into your new computer, you would still be asked for the drive password. Again, even manufacturers can't get around it in some cases.

Further, encrypted filesystems guard against these attacks. Regardless of a BIOS password, GRUB password, and so forth, ultimately, before the root filesystem can be mounted, and therefore decrypted, the user must supply the key to decrypted the filesystem. If the key is unknown, the data is safe. Even powerful electron microscopes would be no good, as the encoded data on the platter is encrypted. We'll discuss this a bit more at length later in this presentation.

Single-User Mode Rosetta

- Single user modes listed for:

  - FreeBSD, OpenBSD, NetBSD, Solaris
  - Red Hat Enterprise Linux
  - Mac OS X
  - Debian
  - HPUX
  - AIX

This listing does not list every possible Unix and how to get root access on every operating system. The principle is the same- the devil is in the details. All that is really needed is either booting into single-user mode, or booting off a live media, such as a CD or USB drive, and mounting and chrooting into the root filesystem.

- FreeBSD, OpenBSD, NetBSD, Solaris:
  - 'boot -s' from the boot menu
- RHEL:
  - append ' 1' as a kernel argument
- Mac OS X:
  - Press and hold the Apple and 's' keys immediately after power on
- Debian:
  - append ' single' as a kernel argument
- HPUX:
  - Press ESC to interrupt booting
  - 'boot pri'
  - 'hpux -is'
- AIX:
  - Boot off the AIX installation media
  - Enter SMS menu
  - Mount root partition
  - chroot to the root filesystem

Password Basics

- Password auditing
- Some numbers
- How many passwords with 94 ASCII characters?
- Cracking at a rate of 30,000 per second:
  - 4 = 78,074,896. 43 minutes

- 5 = 7,339,040,224. 67 hours
- 8 = 6,095,689,385,410,816. 6,443 years

Attacking passwords on systems can happen using a variety of methods. The most common form of attack is known as a dictionary attack. This attack is done by sending password guesses by means of an attacking program to either the login process or against a password database. Of course, if the password database stores passwords in plain text, there is no need for the attacking software. If the password is hashed, as is usually the case, then we need to guess passwords, comparing our hash against the database. If the hashes match, we found our password. It's referred to as a "dictionary attack", because we are using common words found in a dictionary, and comparing the results against that in the database.

Passwords should be audited on a regular basis by system administrators. Password auditing can ensure that users are using strong and long passwords to prevent dictionary attacks, should one be mounted.

Strong passwords include not only lowercase letters, but uppercase, numbers and special characters in the passwords. However, the longer a password is, the more entropy the password carries. Entropy refers to the direct measure of the randomness of the password. An example can be illustrated as follows:

Suppose we use only the 26 characters in the English alphabet, lowercase only. Suppose our password is 5 characters long. This means that there are 26 possible letters each character could be:

```
26 26 26 26 26    or 26^5 (26 to the 5th power)
```

This means there could be a possible 11,881,376 passwords. Now, imagine that we've added uppercase, numbers and all 32 special characters to the potential realm of characters. This means there could be 94 possible characters for each character in the password:

```
94 94 94 94 94    or 94^5 (94 to the 5th power)
```

This is a grand total of only 7,339,040,224. While the search space is substatially larger, suppose you had a 6-character password, rather than 5. With 94 possible characters, the search space just expanded to 689,869,781,056. That's some serious exponential growth! Now, let's look at an 8 character password:

6,095,689,385,410,816 possible passwords! So, while adding "leet speak" does add entropy, entropy exponentially grows with length over just increasing the pool of characters to draw from.

Password Theory

- Cryptanalysis
- Password structures
- Strength qualities
- Entropy
- Implementations

The structure of the presentation will be as follows:

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: 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: 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: This is to cover the amount of work it takes for an attacker to get 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: 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)
2. Easy to remember
3. Unique for each account
4. Stored securely

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

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 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.

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 compromised. 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.

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 some methods towards the end of this presentation for securely storing your passwords.

Cryptanalysis

- Hashing plain text
- Rainbow tables
- Salted passwords
- Shadowed passwords
- John the Ripper

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.

Hashes are one-way algorithms which 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 will always be produced. This means that there is a one-to-one relationship with the data and the hash.

Mathematically speaking, if a function is "one-to-one", then an inverse exists to reach the domain from the range. However, in cryptography, this logic doesn't necessarily hold. For example, suppose my function truncates data. Even though the output is unique for the input, unless you know additional information about the system, and which data was truncated, you won't have everything needed to apply an inverse function to get to the original data.

Hashes are one-to-one, but they are also one-way, meaning that it's trivial to create the hash, but we shouldn't be able to 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.

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

In order to work around rainbow 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 uses a base-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.

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.

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 string of data that produced 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.

Hashes & The Avalanche Effect

- 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 letter 'f' are several bytes apart, as far as the computer is concerned, even if only one bit is flipped. However, the letter g is one character from the letter f, so you would expect the hash to be much more similar, but it's just as radically different.

This is known as 'The Avalanche Effect'. Even the slightest change in the source input, has a massive effect on the output.

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 improbable, to reverse the hash into the originating data.

Further, hashes 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 exist, due to the infinite nature of the data, and the finite nature of the hash. However, the 128-bit MD5 algorithm can produce 340,282,366,920,938,463,463,374,607,431,768,211,456 unique hashes. This should be large enough to avoid unnecessary or frequent collisions. In other words, the output is "unique enough".

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

Common Hash Algorithms

- 128-bit: HMAC, MD2, MD4, MD5, RIPEMD-128
- 160-bit: SHA1, RIPEMD-160
- "SHA2":
  - 224-bit: SHA224
  - 256-bit: SHA256

  - 384-bit: SHA384
  - 512-bit: SHA512

- "SHA3" finalists: BLAKE, Grostl, JH, Keccak & Skein

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:

- MD5: 3.4028E38 unique hashes
- SHA1: 1.462E48 unique hashes
- SHA224: 2.696E67 unique hashes
- SHA256: 1.158E77 unique hashes
- SHA384: 3.940E115 unique hashes
- SHA512: 1.341E154 unique hashes

The US National Institude of Standards and Technology (NIST) announced in 2007 a competition to develop one ore more additional hash algorithms to become a standard known as "SHA3". This competition is similar to the competition that the Advanced Encryption Standard (AES) went through.

The SHA3 competition has gone through 3 rounds, allowing only 14 algorithms to advance to the 2nd round, and 5 algorithms to advance to the second. The finalists have been identified as BLAKE, Grostl, JH, Keccak and Skein, the last coming from famous security expert and cryptographer Bruce Schneier.

The winner of the SHA3 competition, and the publication of the new standard is scheduled to take place in 2012.

Password Structures

- Shadowed passwords
  - `/etc/passwd` vs `/etc/shadow`
- Salted passwords
  - `$_$_____$_____`
- Rainbow tables
  - http://freerainbowtables.com

First some definitions:

- A shadowed password is a password stored in a database that is only readable by the root user on a Unix or Linux machine. Unix didn't implement shadowed passwords until 1988 with AT&T SystemV R3.2 and BSD 4.3 Reno in 1990.

- A salted password is a password that uses a random string of text that is used as a seed to build an encrypted hash. Passwords on Unix and Linux are hashed with this salt as a seed. The salt is passed to the hashing algorithm as an argument appended or prepended to the password text itself.

Most Unix and Linux systems are shadowed by default. Some aren't, like HP/UX, which stores the accounts and the password in the `/etc/passwd` file. Only root can read the contents of the shadowed password file, to prevent security breaches, such as password attacks against accounts on the system.

Further, most passwords on Unix and Linux systems are salted. The salt increases the search space dramatically when performing password attacks. It works in this manner:

First, the salt, as already mentioned, is just a random string of characters that are either appended or prepended to the password itself. This newly created string is then hashed, and hashes are compared to those on the filesystem. If the hash maches, the password has been entered correctly.

So, what is this random string of characters? It is chosen from a list of 64 possible characters. All lowercase letters, uppercase letters, zero through nine, the period and the forward slash. But why use the salt? What security does it add to the password?

Suppose a salt was not used. Of course, any developer worth his salt (pun intended) will hash the password in the database rather than store it in plain text. Because hashes are one way, we should not be able to derive the source text out of a hash. As a result, we need to re-hash the source text, compare it against what is in the database, and see if they match. Because the same source text will always produce the same resulting hash, this algorithm is bullet proof.

Or is it?

What is preventing me, for example, from creating a database table of source text and resulting hashes? This way, rather than brute force the password by trying every possible combination that it could be, I just just check to see if the hash in the password database is already in my hash database. This hash database is known as a 'rainbow table', and checking the table for matching hashes is known as a 'rainbox table attack'. One source text to one resulting hash. Always and every time.

This is problematic, so adding a salt to the password string complicates the rainbow table. For example, suppose I just added one character as a salt to my password. Because that character can be one of 64 characters, this means there could be a possible 64 resulting hashes from the same source text. What if I had two characters in my salt. 64*64 or 64^2 = 4096 potential passwords. My rainbow table is growing exponentially per password.

On my Linux box, I have 8 characters in my salt. This means that there are 64^8 potential hashes my password could be. That's 281,474,976,710,656 potential hashes for a single source text!! Can you imagine a rainbow table containing that sort of data?

For your information, there is an online distributed computing project computing such rainbow tables. You can find more info at:

http://www.freerainbowtables.com

Shadowed Password Example

- `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 down 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.

Cracking Password Databases

- John the Ripper
- Dictionary files

So, now we're at the juicy details. How do we crack these passwords? We've already gotten root on the box, now we want passwords. Why? Well, if I can get passwords off one system, chances are good I can get the same password to work for that account on many systems. Maybe even online services. So, the data I'm about to retrieve could be quite valuable.

John the Ripper is probably the best dictionary attacking software available. `john`, as it's commonly referred to, is Free Software, licensed under the GPL. As a result, most up-to-date Unix-like operating systems have `john` available in their software repositories. If John is not available for your operating system, you can grab the source code at http://www.openwall.com/john/, and compile for your architecture.

John the Ripper is very configurable and contains a decent set of features. One useful feature is providing a wordlist, or dictionary file for it to use before going into intremental mode. Wordlists can be found all over the Internet, and your operating system might even include its own wordlist as a database for spell checking. Full wordlists can be downloaded or purchased on CD from the same site where you downloaded `john`, namely: http://www.openwall.com/wordlists/.

First, before `john` can operate, you need to "unshadow" the system. This requires having access to both the account database and the password database. Typically, these are /etc/passwd and /etc/shadow respectively. By unshadowing the system, you are placing the password in the account database in a copy local to you. `john` then reads

this unshadowed file, and stars working, either from a wordlist your provided, for running in incremental mode.

`john` has some useful options, such as telling it the minimum length of the password in the unshadowed file (if you know such information) as well as a maximum length. This can cut down on the search space, and ultimately the time it takes to attack the password. John also has the capability of logging a session, keeping track of its progress. Should for for eany reason `john` quit executing prematurely, you can restore your previous search session, thus avoiding to redo work you've already done.

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 around 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 spell checkers 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` or `qazwsxedc`, 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-94 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

- Measure of uncertainty
- 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

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.

It's important to understand that your password does not contain the entropy itself. Strictly speaking, any password contains zero entropy, because it has already been

chosen. This is difficult for must to understand. The entropy comes from the system from which the password was chosen. This is why in computer science, we typically refer to them as "entropy pools".

Further, entropy is becomes quite complex when human language is involved. There is a famous estimate by Claude E. Shannon that English text contains between 1.0 and 1.5 bits of entropy per letter. This is known as "Shannon Entropy". Because people are predictable, applying this estimate to a password is questionable. In general, it's very hard to give a good estimate on entropy, when human judgement is involved.

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 94 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 a 72-bit entropy key
- $1,000 for the winner
- 313,868,467,392 keys per second
- ~500 years max to completion

To put entropy into perspective, we can look at Distributed.net. On 19 October 1997, they found the correct solution for the RSA Labs 56-bit secret-key challenge. The key was `0x532B744CC20999`, and it took them only 250 days to locate.

Then, on 14 July 2002 they found the winning key for the RSA Labs 64-bit secret-key challenge. That key was `0x63DE7DC154F4D039` and took 1,757 days to locate.

As of 03 December 2002, they are now working on the 72-bit RSA Labs secret-key challenge. They are currently moving at a pace of about 300 billion keys per second. At that rate, it could take a maximum of roughly 450-500 years. $1,000 USD will go to the finder of the key.

Of course, the laws of probability state that the likelihood of finding the key tomorrow is the same as finding it in 500 years. But this should give you an idea of the exhausting search that must be maintained in order to crack a key of this size.

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 a 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 `ffffffffffffffff`, 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.

Other Implementations

- Secure storage:
  - KeePass- Platform independent, open source
  - Password Safe- Windows, GNU/Linux, open source
  - LastPass- Platform independent, proprietary
- Secure generation:
  - pwgen(1)- Platform independent, open source
  - Diceware- Generated using rolls of a fair "D6" die
  - Utilities using /dev/random and /dev/urandom

There are a number of various password managers and secure password generators that can be used when selecting a password that is high in entropy. Some of the more common implementations are covered here:

KeePass- Keepass is a Free and Open Source password manager that is available for Windows, Mac OS X, GNU/Linux, Android, iOS, Windows Mobile, Palm and Blackberry. It stores passwords, as well as usernames, URLs, and other information about the password in an AES-encrypted database. The program can use two-factor authentication, requiring bot a key and a passphrase to decrypt the database.

Password Safe- Password Safe is a utility written by cryptographer Bruce Schneier. Initially released for Windows, there is a beta version for GNU/Linux in the works. The program is Free and Open Source Software, and it is praised for its simplicity in design. All passwords are stored in a Twofish-encrypted database.

LastPass- LastPass is a popular secure password storage utility that has emerged from the web. It is not Free and Open Source Software, but instead is maintained under a proprietary license. LastPass is available for Windows, Mac OS X, GNU/Linux, Android, iOS, Windows Phone 7, WebOS, Symbian and Blackberry. There are also browser plugins available. Passwords are stored in an AES-encrypted database.

For password generators, the following secure methods are popuplar:

pwgen(1)- Free and Open Source Software designed initially for UNIX and unix-like systems, pwgen(1) is also now available for Windows. It can generate passwords of any length, and is designed to generate passwords are that easily memorized. It can also generate hard to remember, "secure" passwords.

Diceware- Diceware is a physical way of generating secure, truly random passwords. A list, known as the "Diceware list" is a pregenerated, cryptographically signed, list of 7,776 words no longer than 6 characters. Each word is assigned a 5-digit number. A fair "D6" (six-sided die) is rolled 5 times, and the numbers are recorded. The five digit number is looked up to find the matching word. This word is recorded, and the fair D6 is rolled again. This method is repeated as many times as necessary, usually 5 or 6 times, creating that number of words. The words are joined together to create the strong passphrase.

Every word in the Diceware list is lowercase. The entropy of each word in Diceware is roughly log2(7776), or 12.9 bits. So, a passphrase of 5 words would have roughly 64.5 bits. See http://world.std.com/~reinhold/dicewarefaq.html for more information.

`/dev/random` and `/dev/urandom`- These are special character devices that exist on most UNIX and unix-like operating systems. `/dev/random` collects data from exterior environmental noise, such as typing on a keyboard or moving a mouse. The data is stored in an entropy pool. Use of this device will take the data from the entropy pool as its source of random data. When the entropy pool is exhausted, `/dev/random` will block the program from continuing, until enough enivronmental noise has been added to the pool. `/dev/urandom` is similar to `/dev/random`, except that it will not block the program, but instead provide "pseudorandom data" as needed.

Encrypted Filesystems

- LUKS, TrueCrypt, etc
- Logical partition/drive/device
- Passphrase storage

Encrypted filesystems give the added security and comfort that your data cannot be compromised. Encrypted filesystems are becoming very popular both in home use, corporate use and government use. Tools exist for encrypting partitions, whole drives, removable media and files, and these tools exist for all the major operating systems: Unix, Linux, Mac OS X, Windows, BSD, etc.

Encrypted filesystems generally store and setup information in the partition header pointing to a separate physical device before gaining access to the soon-to-be decrypted data. Passwords/passphrases or keyfiles are stored in databases on the physical device. As with standard password databases, this password is salted and hashed as well. This adds the extra benefit of security should the drive fall into the wrong hands.

Cold Boot Attack

- RAM data longevity
- Freezing RAM
- Cold Boot
- Live booting methods
- http://citp.princeton.edu/memory/

If the drive is successfully decrypted with the correct passphrase or keyfile, then the passphrase/keyfile is stored in RAM by the kernel for the duration of the uptime of the operating system. Unfortunately, many of these encrypted filesystems keep the passphrase stored in plaintext in the RAM registers.

Why is this unfortunate? Because RAM does not zero out immediately when the power is cut. Rather, the contents of RAM are degraded slowly back to their magnetic equilibrium. This can take up to 5 minutes on some makes and models. Five minutes

is more than enough time to read the contents of the RAM after being pulled from the system and grab the passphrase.

RAM also has another interesting characteristic. By dropping the tempurature of the hardware chips, the duration it takes to zero out the ram can be extended greatly-up to 15 minutes by just using a can of compressed air.

So, how do you get at the contents of the RAM? Doesn't the operating system wipe most of the registers when shutting down the system? Yes, most definitely. As a result, we need to keep this from happening. We can do so by using the cold boot method. This is done by holding down the power button for a few seconds until the system powers off. At this point, the contents of RAM are identical to the last time the kernel assigned registers. We can now read its contents.

Reading the contents of RAM can be done with two methods:

1 **Physically remove the RAM chips from the computer and insert into** a hardware module for reading the contents directly.

2 **Boot off another medium, like a light LiveCD or LiveUSB, and** execute some applications designed for reading ram contents, and saving the results to disk.

After examination of the RAM contents, the passphrase should be able to be extracted from the file created by reading the chips.

It's important to note that this attack is only successful against current running systems, or those placed in standby or hibernate, where the RAM contents are being retained. If the computer has been turned off for even 2-3 minutes, chances are not in your favor that you'll be able to retrieve the key.

Rather than demonstrating this myself, take a moment to check out the site at: http://citp.princeton.edu/memory/. Not only is there a video presentation of the attack, but software for performing the attack as well as papers and documenatation can also be found at that site.

Evil Maid Attack

- Platform, encrypted filesystem & hardware independent
- Unprotected laptop physically accessed
- Maid installs keylogged boot looader
- Passphrase unknowingly given, and logged
- Maid returns, imaging the drive with passphrase
- http://pthree.org/?p=1175

This attack is a platform, encrypted filesystem and hardware independent attack on encrypted filesystems. The idea stems from the fact that if you leave your laptop unattended, a "hotel room maid" enters your room, and gets physical access to the computer. The maid boots your computer from an external medium, and installs a hacked bootloader with a keylogger.

When you return to your room, and boot the computer, you unknowingly give the encrypted filesystem to the key logger. At this point, the keylogger could install malware on your unencrypted disk and send the passphrase over the network. It could store

the passphrase on an unencrypted part of the disk, or whatever. When the maid returns, she has your passphrase and can decrypt and/or image the drive while you are away.

The beauty of this attack, is there is no bullet-proof protection against this sort of attack. Once you leave you computer unattended, and someone else has physical access, all bets are off. No software, hardware, or security device can protect you against this sort of attack. There are steps you could take, every time you boot your computer, but the amount of work and effort would be high on your part.

Conclusion

- Gain root access through single user mode
- Use John the Ripper to break weak passwords
- Use rainbow tables to break hashed, unsalted passwords
- Defeat encrypted filesystems with cold boot & evil maid attacks
- Responsibility to safeguard these attacks

As mentioned at the start of this presentation, it is our responsibility as system administrators and end users to safe guard our systems from single user mode attacks, weak passwords susceptible to John the Ripper and powering our computer down fully to prevent cold boot attacks.

Again, this presentation was not given to teach you how to crack other people's machines. I am not interested in illegal cracking, harming others computers or accessing their data.

Fin

- Questions, comments, rude remarks?