**Passwords**

So I know this is a networking course but authentication is definitely a networking issue and as of right now the only real way to authenticate yourself on a number of different sites is to have a password.

Now if I was to ask a good portion of the population how computer store their passwords most people would not know how this occurs, but the inter-workings of how passwords are stored is extremely fascinating.

When you put your password into a computer it is not stored in plain text somewhere, it is put through a mathematical formula called a hash. Now the inner workings of a hash are complicated and are not really important for this course, but one key feature that you need to know is that no matter what input is given to a hash, the output is the same size. This is great for a number of reasons, the computer knows ahead of time how much space to set aside for password storage, the mathematical process used is 'one-way' meaning that it is easy to do in one direction but not easy to undo the process, and all passwords, long and short, now look the same. For instance here is an example of an MD5 hash of the password 'hunter2':

2ab96390c7dbe3439de74d0c9b0b1767

As you can see there is no discernible way to get the password back out of it (there are weaknesses in this mathematical hash and it is no longer used for password protection but is a hash most people know about so we'll use it here for our purposes).

So now we have a username and a hashed password combination and we would like to store it, how would that look in the computer? Well it all depends on your OS. If you are in the Unix environment back in the day the password file would be stored in etc/passwd and entries would look like this

bob:101:2ab96390c7dbe3439de74d0c9b0b1767

Where the username is bob, next would be his user ID, and then the hash to his password and now everyone seems happy. Well here's the problem with that, this password file was world readable meaning that anyone on the system could open up that file and look at it. This was by design because how else would normal users be able to authenticate themselves if they couldn't read the password file. So of course this was the point of access for attackers, they knew the username and password hash of anyone who logged into that system. Now what a hacker wants to do is to be able compute a large amount of hashes and check them against the known hash to see if they can guess your password. The problem is storing a lot of hashes takes up a lot of space very quickly. How can the attacker overcome this? The answer is rainbow tables, and we'll take a look at how they work.

**Rainbow Tables**

The first thing you will need is two functions a hash function (H) and a reduction function (R). The hash function does the same kind of hash as we described above, the reduction function is something a little different. What it does is to take a hash function and return a valid password back, keep in mind it is not an inverse function for the hash (since the hash was designed so it could not easily be reversed). So it looks like this:

Starting string (P0) -> Hash-> R1-> Random String(P1) -> Hash-> R2->Random String(P2) …

An example from Wikipedia looks like this:

wikipedia -> H1 -> ao4kd -> R1-> secret ->  H1 -> 9kpmw -> R2 -> jimbo -> H1 -> v0d$x -> R3 -> rootroot

So now we take this idea and apply it to a number of different rows (called chains), with each column having a different reduction function, theoretically this tries the largest password space you can at a time. So instead of storing each hash individually you would just store the starting plain text and the last hash of your chain. In this way you can extremely cut back on the amount of storage needed to store the hashes.

The way that the rainbow table works is by going down the rows and doing the hashing and reducing on the fly while looking for a hash that matches the hash stored in the password file. Once it finds a match for the known password hash it looks to the plaintext that the hash came from and returns it to the user.

**How to protect from rainbow tables?**

So how do operating systems protect their users from attacks like these, the answer is something called a salt. A salt is a random string of characters that get added to a user's password. Let's look how this is done through an example:

A user wants to log into a computer, this is his first time logging into the machine so it asks him to create a password for his account. At this point the operating system picks a salt for the user; each user's salt is different from other user's salts. So the user picks the password 'hunter2', the operating system then picks a salt, let's say in this case it's 'AzBy'. So the password that gets stored is:

Salt + User Password -> 'AzByhunter2'

This is the value that gets hashed and stored in the password file. The question now is, how does this stop rainbow tables? The salt does two things for the password, it increases the password space (fancy term for increase password length and complexity) that can be used and makes it nearly impossible for an attacker to use a rainbow table against the hash because the attacker would have to know the salt for each user before he is able to run the rainbow table against the hashed password.

**Back to Unix:**

So we saw how the older versions of Unix store passwords but what about newer versions? They have a different approach which makes it much more difficult to get the password hash as hackers did before. There is still an etc\passwd file but it looks a little different:

```
Testhost# more /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:Daemon:/sbin:/bin/bash
lp:x:4:7:Printing daemon:/var/spool/lpd:/bin/bash
nobody:x:65534:65533:nobody:/var/lib/nobody:/bin/bash
ftp:x:40:49:FTP account:/srv/ftp:/bin/bash
at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash
irc:x:39:65534:IRC daemon:/usr/lib/ircd:/bin/bash
snort:x:73:68:Snort network monitor:/var/lib/snort:/bin/bash
jim:x:500:100:Jim Smith:/home/jim:/bin/bash
sally:x:501:100:Sally Hart:/home/sally:/bin/bash
```

As you can see no longer is the password stored in the etc\passwd file, instead there is just an ‘x’ in its place. This ‘x’ signifies that there is an entry for that particular username in another file that is only readable by root users; this file is called the shadow file. This marks a big change in the way Unix operating systems handled user’s passwords, as there was no longer and easy way to get to the user’s password hash (and it is still used today).

**Onto Windows:**

So how does Windows handle passwords? Up until Windows XP they used what they call the LM Hash or Lanman hash. The hashing technique was very similar to Unix but had a couple key differences. The allowed characters in Windows were much larger but the total length was held to 14 characters. This hashing method had some key vulnerabilities that were exercised by some attackers, the first being that instead of hashing the entire password, the LM Hash would first null pad the password to make it 14 characters, then it would take the first 7 characters and hash them and then the second 7 characters and hash them and store both of hashes. It also did not use any sort of salting on the password and would uppercase any letter effectively cutting the alphabet password space in half.

Using our example from above (hunter2) it would look like this:

LM Password 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | H |

LM Password 2

| | | | | | | |
|---|---|---|---|---|---|---|
| NULL | U | N | T | E | R | 2 |

This made it so that any attacker only had to either brute force or use rainbow tables on 7 characters at a time instead of having an unknown length password to guess. One might wonder why we only use 7 characters in the enciphering process, it was by design. In the specifications, the enciphering process used was DES which had a normal key size of 64 bits, but the specification for Lanman called for only using 56 bits in its algorithm. This again made the process less secure.

As you can probably guess this was very relatively easy to break and programs like Ophcrack were specifically designed to break the Lanman password set. (http://ophcrack.sourceforge.net/) This tool now includes the ability to go after the next version of Windows password storage which is the Security Accounts Manager or SAM database.

**Overview:**

When doing penetration testing I will first look to see what OS certain machines are using, if it's legacy then there are tried and true methods to get into those machines via weak password storage techniques with relative ease. To help prove my point I have written a small java program that will show you the importance of choosing longer passwords, not just more complex passwords. (Obligatory plug for XKCD, http://xkcd.com/936/) I have included the runtime .jar file and the .java source file for any of you to pick apart.

You can find links to these programs in the references below or on the class website at: http://ureddit.weebly.com/course-documents.html

**References:**

Reduction Function and Rainbow Tables -> http://stichintime.wordpress.com/2009/04/09/rainbow-tables-part-4-reduction-functions/

LM Hash -> http://en.wikipedia.org/wiki/LM_hash

SAM -> http://en.wikipedia.org/wiki/Security_Accounts_Manager

Link to my program: https://www.box.com/s/6b19a2142cdcf31bd16a

> Heads Up: You will need the Java JRE or JDK installed on your machine to run this program. Also please allow up to 10 seconds once you try to run the program for it to give you back results.

Link to source code to program: https://www.box.com/s/069d4834053c0ed03e95