Lecture 24: The Dictionary Attack and the Rainbow-Table Attack on Password Protected Systems

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 14, 2022

3:11pm

Goals:

- The Dictionary Attack
- Thwarting a dictionary attack with log scanning
- Cracking passwords with direct table lookup
- Cracking passwords with hash chains
- Cracking password with rainbow tables
- Password hashing schemes

CONTENTS

	Section Title	*Page*
24.1	**The Dictionary Attack**	3
24.2	**The Password File Embedded in the Conficker Worm**	12
24.3	**Thwarting the Dictionary Attack with Log Scanning**	14
24.4	**Cracking Passwords with Hash Chains and Rainbow Tables**	28
24.5	**Password Hashing Schemes**	41
24.6	**Federated Identity Management**	52
24.7	**Homework Problems**	58

Back to TOC

24.1 THE DICTIONARY ATTACK

- Scanning blocks of IP addresses for vulnerabilities at the ports that are open is in many cases the starting point for breaking into a network.

- If you are not behind a firewall, it is easy to see such ongoing scans. All you have to do is to look at the access or the authorization logs of the services offered by a host in your network. **You will notice that the machines in your network are being constantly scanned for open ports and possible vulnerabilities at those ports.**

- In this lecture I will focus on how people try to break into port 22 that is used for the SSH service. This is a **critical** service since its use goes way beyond just remote login for terminal sessions. It is also used for secure pickup of email from a mail-drop machine and a variety of other applications.

- The most commonly used ploy to break into port 22 is to mount what is referred as a **dictionary attack** on the port. In a dictionary attack, the bad guys try a large number of commonly used names as possible account names on the target machine and, should they succeed in stumbling into a name for which there is actually an account on the target machine, they then proceed to try a large number of commonly used passwords for that account. [An attack closely related to the dictionary attack is known as the **brute-force attack** in which a hostile agent systematically tries **all** possibilities for usernames and passwords. Since the size of the search space in a brute-force attack increases exponentially with the lengths of the usernames and passwords used in the attack, it is not generally feasible to mount such attacks through the internet.]

- If you are logged into a Ubuntu machine, you can see these attempts on an ongoing basis by running the following command line in a separate window

```
tail -f /var/log/auth.log  |  sed G
```

- I will now show just a **two minute segment** of this log produced not too long ago on the host `moonshine.ecn.purdue.edu`. To make it easier to see the usernames being tried by the attacker, I have made a manual entry in a separate line for just the username that the attacker tries in the next break-in attempt. Note that the third line shown for each break-in attempt is truncated because it is much too long. Nonetheless, you can see all of the relevant information in what is displayed. This scan was mounted from the IP address `61.163.228.117`. If you enter this IP address in the query window of `http://www.ip2location.com/` or `http://geoiptool.com`, you will see that the attacker is logged into a network that belongs to the The Postal

Information Technology Office in the city of Henan in China.

```
username tried: staff

Apr 10 13:59:59 moonshine sshd[32057]: Invalid user staff from 61.163.228.117
Apr 10 13:59:59 moonshine sshd[32057]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 13:59:59 moonshine sshd[32057]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:01 moonshine sshd[32057]: Failed password for invalid user staff from 61.163.228.117 port 40805 ssh2


username tried: sales

Apr 10 14:00:08 moonshine sshd[32059]: Invalid user sales from 61.163.228.117
Apr 10 14:00:08 moonshine sshd[32059]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:08 moonshine sshd[32059]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:10 moonshine sshd[32059]: Failed password for invalid user sales from 61.163.228.117 port 41066 ssh2


username tried: recruit

Apr 10 14:00:17 moonshine sshd[32061]: Invalid user recruit from 61.163.228.117
Apr 10 14:00:17 moonshine sshd[32061]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:17 moonshine sshd[32061]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:19 moonshine sshd[32061]: Failed password for invalid user recruit from 61.163.228.117 port 41303 ssh2


username tried: alias

Apr 10 14:00:26 moonshine sshd[32063]: Invalid user alias from 61.163.228.117
Apr 10 14:00:26 moonshine sshd[32063]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:26 moonshine sshd[32063]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:29 moonshine sshd[32063]: Failed password for invalid user alias from 61.163.228.117 port 41539 ssh2


username tried: office

Apr 10 14:00:36 moonshine sshd[32065]: Invalid user office from 61.163.228.117
Apr 10 14:00:36 moonshine sshd[32065]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:36 moonshine sshd[32065]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:38 moonshine sshd[32065]: Failed password for invalid user office from 61.163.228.117 port 41783 ssh2


username tried: samba

Apr 10 14:00:46 moonshine sshd[32067]: Invalid user samba from 61.163.228.117
Apr 10 14:00:46 moonshine sshd[32067]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:46 moonshine sshd[32067]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:47 moonshine sshd[32067]: Failed password for invalid user samba from 61.163.228.117 port 42027 ssh2


username tried: tomcat

Apr 10 14:00:55 moonshine sshd[32069]: Invalid user tomcat from 61.163.228.117
Apr 10 14:00:55 moonshine sshd[32069]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:55 moonshine sshd[32069]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:57 moonshine sshd[32069]: Failed password for invalid user tomcat from 61.163.228.117 port 42247 ssh2


username tried: webadmin

Apr 10 14:01:05 moonshine sshd[32071]: Invalid user webadmin from 61.163.228.117
Apr 10 14:01:05 moonshine sshd[32071]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:05 moonshine sshd[32071]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:07 moonshine sshd[32071]: Failed password for invalid user webadmin from 61.163.228.117 port 42488 ssh2


username tried: spam

Apr 10 14:01:14 moonshine sshd[32073]: Invalid user spam from 61.163.228.117
Apr 10 14:01:14 moonshine sshd[32073]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:14 moonshine sshd[32073]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:16 moonshine sshd[32073]: Failed password for invalid user spam from 61.163.228.117 port 42693 ssh2


username tried: virus

Apr 10 14:01:23 moonshine sshd[32075]: Invalid user virus from 61.163.228.117
Apr 10 14:01:23 moonshine sshd[32075]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:23 moonshine sshd[32075]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:25 moonshine sshd[32075]: Failed password for invalid user virus from 61.163.228.117 port 42917 ssh2


username tried: cyrus

Apr 10 14:01:32 moonshine sshd[32077]: Invalid user cyrus from 61.163.228.117
Apr 10 14:01:32 moonshine sshd[32077]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:32 moonshine sshd[32077]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:35 moonshine sshd[32077]: Failed password for invalid user cyrus from 61.163.228.117 port 43144 ssh2


username tried: oracle

Apr 10 14:01:42 moonshine sshd[32079]: Invalid user oracle from 61.163.228.117
Apr 10 14:01:42 moonshine sshd[32079]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:42 moonshine sshd[32079]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:45 moonshine sshd[32079]: Failed password for invalid user oracle from 61.163.228.117 port 43384 ssh2


username tried: mechael

Apr 10 14:01:52 moonshine sshd[32081]: Invalid user michael from 61.163.228.117
Apr 10 14:01:52 moonshine sshd[32081]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:52 moonshine sshd[32081]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:54 moonshine sshd[32081]: Failed password for invalid user michael from 61.163.228.117 port 43634 ssh2
....
....
....
```

- In mounting a dictionary attack, the bad guys focus particularly on account names that a target machine could be expect to have with high probability. These include:

```
    root
    webmaster
    webadmin
```

```
linux
admin
ftp
mysql
oracle
guest
postgres
test
sales
staff
user

and several others
```

- All of the log entries I showed earlier were for accounts that do not exist on `moonshine.ecn.purdu.edu`. What I show next is a concerted attempt to break into the machine through the `root` account that does exist on the machine. This attack is from the IP address `202.99.32.53`. As before, if you enter this IP address in the query window of `http://www.ip2location.com/` or `http://www.geoiptool.com/`, you will see that the attacker is logged into a network that belongs to the CNCGroup Beijing Province Network in Beijing, China. Note that this is just a three minute segment of the log file.

```
Apr 10 16:23:20 moonshine sshd[32301]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rh
Apr 10 16:23:22 moonshine sshd[32301]: Failed password for root from 202.99.32.53 port 42273 ssh2

Apr 10 16:23:29 moonshine sshd[32303]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rh
Apr 10 16:23:32 moonshine sshd[32303]: Failed password for root from 202.99.32.53 port 42499 ssh2

Apr 10 16:23:39 moonshine sshd[32305]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:23:41 moonshine sshd[32305]: Failed password for root from 202.99.32.53 port 42732 ssh2

Apr 10 16:23:48 moonshine sshd[32307]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:23:50 moonshine sshd[32307]: Failed password for root from 202.99.32.53 port 42976 ssh2

Apr 10 16:23:58 moonshine sshd[32309]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:23:59 moonshine sshd[32309]: Failed password for root from 202.99.32.53 port 43208 ssh2

Apr 10 16:24:06 moonshine sshd[32311]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:08 moonshine sshd[32311]: Failed password for root from 202.99.32.53 port 43439 ssh2

Apr 10 16:24:15 moonshine sshd[32313]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:17 moonshine sshd[32313]: Failed password for root from 202.99.32.53 port 43659 ssh2

Apr 10 16:24:24 moonshine sshd[32315]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:26 moonshine sshd[32315]: Failed password for root from 202.99.32.53 port 43901 ssh2

Apr 10 16:24:33 moonshine sshd[32317]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:35 moonshine sshd[32317]: Failed password for root from 202.99.32.53 port 44128 ssh2

Apr 10 16:24:42 moonshine sshd[32319]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:44 moonshine sshd[32319]: Failed password for root from 202.99.32.53 port 44352 ssh2

Apr 10 16:24:51 moonshine sshd[32321]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:53 moonshine sshd[32321]: Failed password for root from 202.99.32.53 port 44577 ssh2

Apr 10 16:25:00 moonshine sshd[32323]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:01 moonshine sshd[32323]: Failed password for root from 202.99.32.53 port 44803 ssh2

Apr 10 16:25:09 moonshine sshd[32325]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:11 moonshine sshd[32325]: Failed password for root from 202.99.32.53 port 45024 ssh2

Apr 10 16:25:18 moonshine sshd[32327]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:20 moonshine sshd[32327]: Failed password for root from 202.99.32.53 port 45269 ssh2

Apr 10 16:25:27 moonshine sshd[32329]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:29 moonshine sshd[32329]: Failed password for root from 202.99.32.53 port 45496 ssh2

Apr 10 16:25:36 moonshine sshd[32331]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:38 moonshine sshd[32331]: Failed password for root from 202.99.32.53 port 45725 ssh2

Apr 10 16:25:45 moonshine sshd[32333]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:47 moonshine sshd[32333]: Failed password for root from 202.99.32.53 port 45951 ssh2

Apr 10 16:25:54 moonshine sshd[32335]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:56 moonshine sshd[32335]: Failed password for root from 202.99.32.53 port 46186 ssh2

Apr 10 16:26:03 moonshine sshd[32337]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:26:05 moonshine sshd[32337]: Failed password for root from 202.99.32.53 port 46402 ssh2

Apr 10 16:26:12 moonshine sshd[32339]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:26:14 moonshine sshd[32339]: Failed password for root from 202.99.32.53 port 46637 ssh2

Apr 10 16:26:21 moonshine sshd[32341]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:26:23 moonshine sshd[32341]: Failed password for root from 202.99.32.53 port 46859 ssh2
....
....
....
```

- As long as we are on the subject of looking at the `/var/log/auth.log` log file, in the same file you will also see numerous break-in entries that look like those shown below. These entries contain the special entry "**failed - POSSIBLE BREAK-IN ATTEMPT!**". Although such entries look alarming at first sight, they are no more sinister than the examples I showed earlier. What triggers this particular form of log entry is when the local `sshd` daemon cannot reconcile the domain name from where SSH connection request is coming from with the IP address contained in the connection request. Shown below is a small segment of such an attack on `moonshine.ecn.purdue.edu` from the IP address 78.153.210.68. As before, if you enter this address in the query window of `http://www.ip2location.com/`, you will discover that the attacker is logged into the network that belongs to PEM VPS Hosting Servers in the city of Carlow, Ireland. The attack represents a concerted attempt to break into the `root` account by guessing the password. I have abbreviated the first line of each attempt as indicated by the sequence of dots in such lines. An actual first line of each attempt looks like the following:

  ```
  Apr 10 21:42:45 moonshine sshd[787]: reverse mapping checking \
       getaddrinfo for 210-68.colo.sta.blacknight.ie [78.153.210.68] \
       failed - POSSIBLE BREAK-IN ATTEMPT!
  ```

  Here is just a **two minute segment** of such an attack:

  ```
  Apr 10 21:41:58 moonshine sshd[757]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
  Apr 10 21:41:58 moonshine sshd[757]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
  Apr 10 21:41:59 moonshine sshd[757]: Failed password for root from 78.153.210.68 port 43828 ssh2


  Apr 10 21:42:01 moonshine sshd[759]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
  ```

```
Apr 10 21:42:01 moonshine sshd[759]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:02 moonshine sshd[759]: Failed password for root from 78.153.210.68 port 43948 ssh2


Apr 10 21:42:03 moonshine sshd[761]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:04 moonshine sshd[761]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:06 moonshine sshd[761]: Failed password for root from 78.153.210.68 port 44058 ssh2


Apr 10 21:42:08 moonshine sshd[763]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:08 moonshine sshd[763]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:09 moonshine sshd[763]: Failed password for root from 78.153.210.68 port 44210 ssh2


Apr 10 21:42:11 moonshine sshd[765]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:11 moonshine sshd[765]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:12 moonshine sshd[765]: Failed password for root from 78.153.210.68 port 44330 ssh2


Apr 10 21:42:14 moonshine sshd[767]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:14 moonshine sshd[767]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:16 moonshine sshd[767]: Failed password for root from 78.153.210.68 port 44440 ssh2


Apr 10 21:42:17 moonshine sshd[769]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:17 moonshine sshd[769]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:19 moonshine sshd[769]: Failed password for root from 78.153.210.68 port 44568 ssh2


Apr 10 21:42:20 moonshine sshd[771]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:20 moonshine sshd[771]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:22 moonshine sshd[771]: Failed password for root from 78.153.210.68 port 44698 ssh2


Apr 10 21:42:23 moonshine sshd[773]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:23 moonshine sshd[773]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:25 moonshine sshd[773]: Failed password for root from 78.153.210.68 port 44818 ssh2


Apr 10 21:42:27 moonshine sshd[775]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:27 moonshine sshd[775]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:29 moonshine sshd[775]: Failed password for root from 78.153.210.68 port 44928 ssh2


Apr 10 21:42:30 moonshine sshd[777]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:30 moonshine sshd[777]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:32 moonshine sshd[777]: Failed password for root from 78.153.210.68 port 45089 ssh2


Apr 10 21:42:33 moonshine sshd[779]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:33 moonshine sshd[779]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:34 moonshine sshd[779]: Failed password for root from 78.153.210.68 port 45186 ssh2


Apr 10 21:42:36 moonshine sshd[781]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:36 moonshine sshd[781]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:37 moonshine sshd[781]: Failed password for root from 78.153.210.68 port 45299 ssh2


Apr 10 21:42:38 moonshine sshd[783]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:38 moonshine sshd[783]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:40 moonshine sshd[783]: Failed password for root from 78.153.210.68 port 45405 ssh2
Apr 10 21:42:41 moonshine sshd[785]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:41 moonshine sshd[785]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:43 moonshine sshd[785]: Failed password for root from 78.153.210.68 port 45521 ssh2


Apr 10 21:42:45 moonshine sshd[787]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:45 moonshine sshd[787]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:47 moonshine sshd[787]: Failed password for root from 78.153.210.68 port 45663 ssh2


Apr 10 21:42:48 moonshine sshd[789]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:48 moonshine sshd[789]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:49 moonshine sshd[789]: Failed password for root from 78.153.210.68 port 45778 ssh2


Apr 10 21:42:51 moonshine sshd[791]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:51 moonshine sshd[791]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:53 moonshine sshd[791]: Failed password for root from 78.153.210.68 port 45882 ssh2


Apr 10 21:42:54 moonshine sshd[793]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:54 moonshine sshd[793]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:55 moonshine sshd[793]: Failed password for root from 78.153.210.68 port 46011 ssh2


Apr 10 21:42:57 moonshine sshd[795]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:57 moonshine sshd[795]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:58 moonshine sshd[795]: Failed password for root from 78.153.210.68 port 46123 ssh2
....
....
....
```

Back to TOC

24.2 THE PASSWORD FILE EMBEDDED IN THE CONFICKER WORM

- When an attacker who has mounted a dictionary attack does find an installed account on the victim machine, the next challenge for the attacker is to gain entry into the account by making guesses at the password for the account. For example, the last two segments of the `auth.log` file shown in the previous section are for two concerted attempts by two different attackers to guess the password for the `root` account on `moonshine.ecn.purdue.edu`.

- In the context of guessing the passwords, it is interesting to examine the guesses that are embedded in the binary for the Conficker worm that we discussed in Lecture 22. Here are the 240 guesses that were taken from

  `http://onecare.live.com/standard/en-us/virusenc/virusencinfo.htm?VirusName=Worm:Win32/Conficker.B`

```
123             1234            12345           123456
1234567         12345678        123456789       1234567890
123123          12321           123321          123abc
123qwe          123asd          1234abcd        1234qwer
1q2w3e          a1b2c3          admin           Admin
administrator   nimda           qwewq           qweewq
qwerty          qweasd          asdsa           asddsa
asdzxc          asdfgh          qweasdzxc       q1w2e3
qazwsx          qazwsxedc       zxcxz           zxccxz
zxcvb           zxcvbn          passwd          password
Password        login           Login           pass
mypass          mypassword      adminadmin      root
```

rootroot	test	testtest	temp
temptemp	foofoo	foobar	default
password1	password12	password123	admin1
admin12	admin123	pass1	pass12
pass123	root123	pw123	abc123
qwe123	test123	temp123	mypc123
home123	work123	boss123	love123
sample	example	internet	Internet
nopass	nopassword	nothing	ihavenopass
temporary	manager	business	oracle
lotus	database	backup	owner
computer	server	secret	super
share	superuser	supervisor	office
shadow	system	public	secure
security	desktop	changeme	codename
codeword	nobody	cluster	customer
exchange	explorer	campus	money
access	domain	letmein	letitbe
anything	unknown	monitor	windows
files	academia	account	student
freedom	forever	cookie	coffee
market	private	games	killer
controller	intranet	work	home
job	foo	web	file
sql	aaa	aaaa	aaaaa
qqq	qqqq	qqqqq	xxx
xxxx	xxxxx	zzz	zzzz
zzzzz	fuck	12	21
321	4321	54321	654321
7654321	87654321	987654321	0987654321
0	00	000	0000
00000	00000	0000000	00000000
1	11	111	1111
11111	111111	1111111	11111111
2	22	222	2222
22222	222222	2222222	22222222
3	33	333	3333
33333	333333	3333333	33333333
4	44	444	4444
44444	444444	4444444	44444444
5	55	555	5555
55555	555555	5555555	55555555
6	66	666	6666
66666	666666	6666666	66666666
7	77	777	7777
77777	777777	7777777	77777777
8	88	888	8888
88888	888888	8888888	88888888
9	99	999	9999
99999	999999	9999999	99999999

Back to TOC

24.3 THWARTING THE DICTIONARY ATTACK WITH LOG SCANNING

- Before getting to the subject of log scanning for protecting a computer/network against a dictionary attack, I should say quickly that if, say, the computer you want to protect is at your home and you want to be able to SSH into it from work without allowing others to be able to do the same, just a couple of entries in the `/etc/hosts.allow` and the `/etc/hosts.deny` files would keep all intruders at bay.

```
/etc/hosts.allow   :    sshd: xxx.xxx.xxx.xxx

/etc/hosts.deny    :    ALL: ALL
```

  where `xxx.xxx.xxx.xxx` is the IP address from where you wish to connect to your home machine. Since `/etc/hosts.allow` takes precedence over `/etc/hosts.deny`, the above two entries will ensure that only you will be allowed SSH access into the machine.

- Let's now consider a more general situation of detecting repeated break-in attempts and temporarily (or, sometimes, permanently) blacklisting IP addresses from where the attacks are emanating.

- Until recently, DenyHosts was the most popular tool used for keeping an eye on the **sshd** server access logs (in `/var/log/auth.log` on Linux machines). DenyHosts, however, was removed from Ubuntu distributions of Linux sometime in 2014 for "unaddressed security issues" and other reasons.

- As far as the Linux platforms are concerned, **Fail2Ban** is now the most commonly used tool for intrusion prevention through log scanning. [According to the Wikipedia page on Fail2Ban, the development of Fail2Ban has been led by Cyril Jaquier, Yaroslav Halchenko, Daniel Black, Steven Hiscocks, and Arturo 'Buanzo' Busleiman as an opensource project. DenyHosts was created by Phil Schwartz.]

- While both Fail2Ban and DenyHosts detect intrusion attempts by keeping track of the number of login attempts (during a time interval whose length in set is the config file), there is a fundamental difference in how the two tools keep the blacklisted IP addresses at bay. With Fail2Ban, a blacklisted IP address is kept out by adding a new rule to the iptables firewall. [See Lecture 18 on iptables.] On the other hand, DenyHosts places a blacklisted IP address in the `/etc/hosts.deny` file. Subsequently, with both tools, no further SSH connections from the same IP address would be honored — at least until the expiration of a certain pre-set time interval. [Depending on the config options you set, Fail2Ban would be happy to just send you a notification (that is, without banning the IP address) when it sees too many unsuccessful attempts at entry. As you will soon see, by using regex based filters, Fail2Ban can also try to detect malicious behaviors by the connections made by IP addresses (say, for downloading web pages) and subsequently it can take any action you wish vis-a-vis those IP addresses.]

- You may think there is a bit of irony involved in making future intrusion prevention decisions on the basis of *unsuccessful* attempts in the past. Let's say an intruder has successfully managed to break into a machine as root the very first time. It is safe to assume that such an intruder would immediately eliminate all signs of his/her entry into the system. So, one might say, with log scanning of the sort used in Fail2Ban and DenyHosts, your security decision is based more on the actions of a clumsy thief who is unsuccessful and not on the actions of those who may have caused you serious harm in the past.

- However, since it is reasonable to assume that even a successful thief may need to make a few attempts before hitting the jackpot, it makes sense to use tools like Fail2Ban and DenyHosts.

- DenyHost was created exclusively for monitoring the SSHD access log files.

- **On the other hand, one of the best things about Fail2Ban is its versatility.** It can block network access to just about any application that creates a log file for incoming connection requests. It's worth your while to spend a few minutes poring over its config file `/etc/fail2ban/jail.conf` and to see its different sections, as delineated by '`[application name]`', in order to get a sense of the range of applications for which you can trap misbehaving IP addresses. By the way, you can also specify additional server applications — applications that are of your own making and that are not currently mentioned in the config file — if you want to monitor and control network access to them with Fail2Ban. All you have to do is to enter a few lines of text in the config files. [Fail2Ban is so versatile that, even for the same server application running in your computer, it can identity IP addresses that are engaged in different malicious activities and, depending on what activity is involved, it can take different actions. If you examine the file `jail.conf`, you will see entries for an application that is named `[apache-badbots]` that monitors accesses to HTTP and HTTPS in order to catch intruders that make seemingly ordinary web accesses but for the sole purpose of mining email addresses from the web pages being doled out. Fail2Ban detects activities with the help of filters based on regular expressions. A certain number of these filters are predefined in the `/etc/fail2ban/` directory. However, you can create your own filters to supersede those that come predefined or that are new for new kinds of behaviors by malicious hosts.]

- You can install Fail2Ban with `apt-get` or through your Synaptic Package Manager. By default, it will only monitor the log entries in the `/var/log/auth.log` file. However, as mentioned in the previous bullet, you can monitor network attacks on just about any server application running in your computer as long as it spits out a log file for the incoming requests for connections. [You enable log monitoring for an application by inserting the line '`enabled = true`' in the relevant section of the file `/etc/fail2ban/jail.local`. By default, `enabled` is set to `true` for SSHD.]

- Fail2Ban is written in Python and all of its files are in the directory `/etc/fail2ban`. That directory and its subdirectories contain a number of config files that can be used to specify different criteria for trapping IP addresses that make intrusion attempts (and that engage in malicious behaviors) and for specifying the actions to be taken for the blacklisted addresses. Execute '`man jail.conf`' to see the man page regarding the different configuration options.

- The act of installing Fail2Ban also enables it on your machine. You must however customize its behavior for your specific host. To verify that Fail2Ban is up and running, you can execute

  ```
  sudo fail2ban-client status
  ```

  It should return:

  ```
  Status
  |- Number of jail:      1
  `- Jail list:   sshd
  ```

- Another way to see that you have successfully installed Fail2Ban is by checking your iptables firewall rules. For example, assuming that the chains in your firewall were empty to begin with, if you execute the command '`sudo iptables -L`' after installing Fail2Ban, you should see

  ```
  Chain INPUT (policy ACCEPT)
  target     prot opt source       destination
  f2b-sshd   tcp  --  anywhere     anywhere          multiport dports ssh

  Chain FORWARD (policy ACCEPT)
  target     prot opt source       destination
  ```

```
Chain OUTPUT (policy ACCEPT)
target     prot opt source       destination

Chain f2b-sshd (1 references)
target     prot opt source       destination
RETURN     all  --  anywhere     anywhere
```

Note, in particular, the jump to the 'user-defined' chain `f2b-sshd` action inserted by Fail2Ban in the predefined INPUT chain of the filter table of the firewall. [You may wish to review Lecture 18 at this point if you do not remember that 'filter' is one of the four tables in an iptables based firewall and that this table has three predefined chains: `INPUT`, `OUTPUT` and `FORWARD`.] In this manner, all incoming packets would be first subject to the rules in the `f2b-sshd` chain and those that are not trapped by any of the rules in that chain would be sent back to be processed by the rest of the rules in the `INPUT` chain. That we can say on account of the definition of the `f2b-sshd` chain at the bottom of the output At the moment there are no restrictions on any IP addresses in the `f2b-sshd` chain.

- If all you want from Fail2Ban is for it to monitor SSH access (and to ban offending IP addresses) on port 22, you need to make only a very small number of changes — six or fewer — to just one config file. However, as mentioned in the config file `/etc/fail2ban/jail.conf`, you must first create its copy with the name `/etc/fail2ban/jail.local`. All of your customizations must be in the ".local" version of the config file. [Fail2Ban is programmed to first parse the ".conf" files and, subsequently, the ".local" files. In this manner, any customizations in the ".local" files override the corresponding entries in the ".conf" files. This ploy allows the ".conf" files to be changed with upgrades to the software without losing the user-specified customization information.] The small number of changes you'd need to make in `/etc/fail2ban/jail.local` are likely to be in the following lines (I have shown the entries in my install of Fail2Ban):

```
bantime = 3600

findtime  = 3600

maxretry = 5

mta = sendmail

destemail = root@localhost

action = %(action_mwl)s
```

Here is a description of what these parameters mean: The config parameter `bantime` specifies in seconds the duration of time for which a blacklisted IP address is denied further access. The config parameters `findtime` and `maxtry` are used together to decide when to blacklist an IP address. If the intruder makes more than `maxtry` attempts during a `findtime` period of time, the IP address is quarantined for the duration set by `bantime`. The parameter `mta` specifies the mail transport agent to use for sending an email notification to a designated person/admin when an IP address is blacklisted. This notification is sent to the account specified by the parameter `destemail`. Finally, the parameter `action`, as you would guess, tells Fail2Ban what to with an IP address that meets the repeat access conditions as set by the `findtime` and the `maxtry` parameters. In most cases, you'd want those addresses to be banned for the duration set by `bantime`. This action corresponds to the choice "`action_`" inside the curly brackets for the `action` entry shown above. However, if you want that a notification be also sent to the account set by `destemail`, you would need to choose "`action_mw`" for what goes inside the curly brackets. Yet another option for the same is "`action_mwl`". With the "`action_mw`" choice, the email notification will include a "whois" report on the intruding host. And, with "`action_mwl`", the email notification will include relevant log lines.

- Since, to the best of what I know, DenyHosts continues to be rather widely deployed, the rest of this section is devoted to that tool.

- With regard to how DenyHosts works, in addition to entering a blacklisted IP address in in the `/etc/hosts.deny` file, the blacklisted IP addresses are also recorded in in a few more files elsewhere in your directory system for the purpose of synchronizing your blacklisted IP addresses with similar such addresses collected by other hosts in the internet if you have the synchronization option turned on in the config files — see the end of this section for the names of these files. As to how may attempts at breaking in should qualify for blacklisting an IP address can be set by you in the configuration file of DenyHosts.

- The main config file for DenyHosts is `/etc/denyhosts.conf`. [Ordinarily, you would only need to make a small number of changes in the config file for its customization to your needs. For example, when I used to use DenyHosts on my Linux laptop, I changed the `ADMIN_EMAIL` to kak@localhost, uncommented the `SMTP_FROM` and `SYNC_SERVER` lines, set `PURGE_DENY` to 1w, `BLOCK_SERVICE` to `ALL`, `DENY_THRESHOLD_INVALID` to 3, `DENY_THRESHOLD_VALID` to 5, `SYNC_INTERVAL` to 1h, `SYNC_UPLOAD` to `YES`, and `SYNC_DOWNLOAD` to `YES`.] DenyHosts makes its log entries in the `/var/log/denyhosts` file. You can also do "`man denyhosts`" to get more information on the tool. DenyHosts comes with a synchronization feature that allows it to download the IP addresses that have been blacklisted elsewhere. In that sense, the tool has the ability to give you advance protection.

- In the same manner as Fail2Ban, DenyHosts can silently restore access privileges of a blacklisted IP address after a certain period of time whose duration is set in the configuration file. The homepage for DenyHosts is `http://denyhosts.sourceforge.net/`.

- Shown below is a 45 second segment of the `auth.log` file after DenyHosts was fired up. This represents an illegal attempt to break into `moonshine.ecn.purdue.edu` from someone at 190.12.41.50. If you enter this IP address in the query window of `http://www.ip2location.com`, you will discover that the intruder is logged into a network owned by an outfit called PUNTONET in the country of Ecuador.

```
tried to connect as root:

Apr 25 16:29:03 moonshine sshd[31037]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:03 moonshine sshd[31037]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rh
Apr 25 16:29:04 moonshine sshd[31037]: Failed password for root from 190.12.41.50 port 54042 ssh2


tried to connect as apple:

Apr 25 16:29:08 moonshine sshd[31039]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:08 moonshine sshd[31039]: Invalid user apple from 190.12.41.50
Apr 25 16:29:08 moonshine sshd[31039]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rh
Apr 25 16:29:10 moonshine sshd[31039]: Failed password for invalid user apple from 190.12.41.50 port 54102 ssh2


tried to connect as magazine:

Apr 25 16:29:13 moonshine sshd[31041]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:13 moonshine sshd[31041]: Invalid user magazine from 190.12.41.50
Apr 25 16:29:13 moonshine sshd[31041]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rh
Apr 25 16:29:15 moonshine sshd[31041]: Failed password for invalid user magazine from 190.12.41.50 port 54163 ssh2


tried to connect as sophia:

Apr 25 16:29:18 moonshine sshd[31043]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:18 moonshine sshd[31043]: Invalid user sophia from 190.12.41.50
Apr 25 16:29:18 moonshine sshd[31043]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rh
Apr 25 16:29:20 moonshine sshd[31043]: Failed password for invalid user sophia from 190.12.41.50 port 54227 ssh2


tried to connect as janet:
```

```
Apr 25 16:29:23 moonshine sshd[31045]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:23 moonshine sshd[31045]: Invalid user janet from 190.12.41.50
Apr 25 16:29:23 moonshine sshd[31045]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:25 moonshine sshd[31045]: Failed password for invalid user janet from 190.12.41.50 port 54289 ssh2


tried to connect as taylor:

Apr 25 16:29:28 moonshine sshd[31047]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:28 moonshine sshd[31047]: Invalid user taylor from 190.12.41.50
Apr 25 16:29:28 moonshine sshd[31047]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:30 moonshine sshd[31047]: Failed password for invalid user taylor from 190.12.41.50 port 54351 ssh2


tried to connect as vanessa:

Apr 25 16:29:33 moonshine sshd[31049]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:33 moonshine sshd[31049]: Invalid user vanessa from 190.12.41.50
Apr 25 16:29:33 moonshine sshd[31049]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:34 moonshine sshd[31049]: Failed password for invalid user vanessa from 190.12.41.50 port 54406 ssh2



tried to connect as alyson:

Apr 25 16:29:38 moonshine sshd[31051]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:38 moonshine sshd[31051]: Invalid user alyson from 190.12.41.50
Apr 25 16:29:38 moonshine sshd[31051]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:39 moonshine sshd[31051]: Failed password for invalid user alyson from 190.12.41.50 port 54467 ssh2


tried again to connect as root:

Apr 25 16:29:42 moonshine sshd[31053]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:42 moonshine sshd[31053]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:44 moonshine sshd[31053]: Failed password for root from 190.12.41.50 port 54509 ssh2


tried again to connect as research:

Apr 25 16:29:48 moonshine sshd[31055]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:48 moonshine sshd[31055]: Invalid user research from 190.12.41.50
Apr 25 16:29:48 moonshine sshd[31055]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:50 moonshine sshd[31055]: Failed password for invalid user research from 190.12.41.50 port 54581 ssh2


                        AND FINALLY CAUGHT BY DENYHOSTS:


Apr 25 16:29:50 moonshine sshd[31060]: refused connect from ::ffff:190.12.41.50 (::ffff:190.12.41.50)
```

- From the segment of the log file shown above, you can see that the intruder made 10 attempts before getting trapped by DenyHosts. How many attempts an intruder is allowed to make before any further connection requests are summarily refused depends on the choices you make in the `/etc/denyhosts.conf` configuration file. I had the following setting in the config file for the log file segment shown above:

```
    DENY_THRESHOLD_INVALID = 5
    DENY_THRESHOLD_VALID   = 10
```

where the first number sets the limit on how many times an intruder can try to gain entry with usernames that do NOT exist in the `/etc/passwd` file and the second sets a similar limit on trying to gain entry through usernames that actually do exist. I subsequently changed the former to 3 and the latter to 5.

- Obviously, what values you choose for the two parameters shown above and other similar parameters in the config file depends on how much latitude you want to give the legitimate users of your host with regarding to any accidental mis-entry of user names and passwords.

- What I show next is an attack by a cleverer intruder. What this intruder is attempting is not your classic dictionary attack. The intruder appears to know that he/she will be allowed only a limited number of attempts (probably from a prior manual attempt to break in with a number of different login names from conceivably a different IP address). So the intruder is trying only the login names that form the various substrings in the domain name of "`moonshine.ecn.purdue.edu`". Note that the intruder is making only 4 attempts for each login name, one less than it takes to get disbarred by the config settings shown previously. To see the source of the attack, enter the IP address 66.135.39.212 in the query window of `http://www.ip2location.com` and you will notice that this address belongs to a company called Zartana based in Brazil. *In its description at LinkedIn, this company claims to be able to deliver 2,000,000 email messages per hour.*

```
login tried: ecn   (Attempt 1 as ecn)

May  5 10:11:23 moonshine sshd[27483]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIB
May  5 10:11:23 moonshine sshd[27483]: Invalid user ecn from 66.135.39.212
May  5 10:11:23 moonshine sshd[27483]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:23 moonshine sshd[27483]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:25 moonshine sshd[27483]: Failed password for invalid user ecn from 66.135.39.212 port 33901 ssh2


login tried: ecn   (Attempt 2 as ecn)

May  5 10:11:25 moonshine sshd[27485]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIB
May  5 10:11:25 moonshine sshd[27485]: Invalid user ecn from 66.135.39.212
May  5 10:11:25 moonshine sshd[27485]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:25 moonshine sshd[27485]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:28 moonshine sshd[27485]: Failed password for invalid user ecn from 66.135.39.212 port 34028 ssh2


login tried: ecn   (Attempt 3 as ecn)

May  5 10:11:29 moonshine sshd[27487]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIB
May  5 10:11:29 moonshine sshd[27487]: Invalid user ecn from 66.135.39.212
May  5 10:11:29 moonshine sshd[27487]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:29 moonshine sshd[27487]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:31 moonshine sshd[27487]: Failed password for invalid user ecn from 66.135.39.212 port 34163 ssh2


login tried: ecn   (Attempt 4 as ecn)

May  5 10:11:32 moonshine sshd[27489]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIB
May  5 10:11:32 moonshine sshd[27489]: Invalid user ecn from 66.135.39.212
May  5 10:11:32 moonshine sshd[27489]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:32 moonshine sshd[27489]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:34 moonshine sshd[27489]: Failed password for invalid user ecn from 66.135.39.212 port 34282 ssh2


login tried: moonshine   (Attempt 1 as moonshine)

May  5 10:11:35 moonshine sshd[27491]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIB
May  5 10:11:35 moonshine sshd[27491]: Invalid user moonshine from 66.135.39.212
May  5 10:11:35 moonshine sshd[27491]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:35 moonshine sshd[27491]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:37 moonshine sshd[27491]: Failed password for invalid user moonshine from 66.135.39.212 port 34384 ssh2


login tried: moonshine   (Attempt 2 as moonshine)

May  5 10:11:37 moonshine sshd[27493]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:37 moonshine sshd[27493]: Invalid user moonshine from 66.135.39.212
May  5 10:11:37 moonshine sshd[27493]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:37 moonshine sshd[27493]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:40 moonshine sshd[27493]: Failed password for invalid user moonshine from 66.135.39.212 port 34514 ssh2


login tried: moonshine   (Attempt 3 as moonshine)

May  5 10:11:41 moonshine sshd[27495]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:41 moonshine sshd[27495]: Invalid user moonshine from 66.135.39.212
May  5 10:11:41 moonshine sshd[27495]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:41 moonshine sshd[27495]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:43 moonshine sshd[27495]: Failed password for invalid user moonshine from 66.135.39.212 port 34637 ssh2


login tried: moonshine   (Attempt 4 as moonshine)

May  5 10:11:43 moonshine sshd[27497]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:43 moonshine sshd[27497]: Invalid user moonshine from 66.135.39.212
May  5 10:11:43 moonshine sshd[27497]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:43 moonshine sshd[27497]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:46 moonshine sshd[27497]: Failed password for invalid user moonshine from 66.135.39.212 port 34759 ssh2


login tried: purdue   (Attempt 1 as purdue)

May  5 10:11:47 moonshine sshd[27499]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:47 moonshine sshd[27499]: Invalid user purdue from 66.135.39.212
May  5 10:11:47 moonshine sshd[27499]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:47 moonshine sshd[27499]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:49 moonshine sshd[27499]: Failed password for invalid user purdue from 66.135.39.212 port 34906 ssh2


login tried: purdue   (Attempt 2 as purdue)

May  5 10:11:49 moonshine sshd[27501]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:49 moonshine sshd[27501]: Invalid user purdue from 66.135.39.212
May  5 10:11:49 moonshine sshd[27501]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:49 moonshine sshd[27501]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:52 moonshine sshd[27501]: Failed password for invalid user purdue from 66.135.39.212 port 35030 ssh2


login tried: purdue   (Attempt 3 as purdue)

May  5 10:11:52 moonshine sshd[27503]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:52 moonshine sshd[27503]: Invalid user purdue from 66.135.39.212
May  5 10:11:52 moonshine sshd[27503]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:52 moonshine sshd[27503]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:54 moonshine sshd[27503]: Failed password for invalid user purdue from 66.135.39.212 port 35189 ssh2


login tried: purdue   (Attempt 4 as purdue)

May  5 10:11:55 moonshine sshd[27505]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:55 moonshine sshd[27505]: Invalid user purdue from 66.135.39.212
May  5 10:11:55 moonshine sshd[27505]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:55 moonshine sshd[27505]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:58 moonshine sshd[27505]: Failed password for invalid user purdue from 66.135.39.212 port 35321 ssh2


                      FINALLY TRAPPED BY DENYHOSTS
```

Back to TOC

24.4 Cracking Passwords with Hash Chains and Rainbow Tables

- As you have seen in the earlier sections of this lecture, a dictionary attack means trying out one password at a time to break into a machine. Password cracking, on the other hand, means that you have already broken into a machine and somehow gotten hold of the document where all the password hashes are stored. (This document is usually referred to as the *System Password File.*) Now you want to map the password hashes back to the character strings that are the passwords as entered by the users.

- You might ask that if a specific feature of a hashing function is its one-way property — that it maps a string to a hash but you are not supposed to be able to construct an inverse-map from the hash to the string — how is password cracking possible at all? Note that, strictly speaking, this one-way property applies only to hash functions such as those that belong to the officially sanctioned SHA family. In the past, the hash functions used for password security have not always been the sort of hash functions discussed in Lecture 15, as you will soon see in what follows in this section.

- The following two facts have given much impetus to the development of password cracking methods during the last twenty years: (1) The older versions of the Microsoft Windows platform used an extremely weak method for hashing passwords; and (2) The ubiquity of the Windows machines all around the world.

- The password hashing used in the older versions of the Windows platform is known as the LM Hash where LM stands for LAN Manager. This hashing function is so weak that a password can be cracked — meaning that the ASCII string for the password can be inferred from its hash value — in just a few seconds through the rainbow table attack that I'll describe later in this section. An open-source tool called `Ophcrack`, co-developed by the inventor of the rainbow tables, can crack such a password hash in about 13.6 seconds 99.9% of the time using a rainbow table of size roughly 1 GB. [The developers of Ophcrack claim that they can also crack the hashes generated by the NTLM Hash algorithm used in the more recent Windows machines. Note that the most recent Microsoft applications have moved on to NTLMv2 and Kerberos based protocols for user authentication.]

- Since the LM Hash has served as such a magnet for the development of password cracking algorithms, it is educational to review it. For the LM Hash algorithm, a password is limited to a maximum of 14 ASCII characters and zero-padded to 14 if shorter than that. Any lowercase characters in the password are converted to uppercase. Subsequently, this 14-character string is divided into two 7-character substrings, with the 56 bits of each substring used as a key to the DES algorithm to encrypt the 8-character plaintext string `KGS!@#$%`. Each half produces a 64-bit ciphertext and two ciphertext bit streams are simply concatenated together to create a 128-bit pattern that is stored as the password "hash" by the LM Hash algorithm. [In case you are wondering about the plaintext `KGS!@#$%`, its first three letters, KGS, are believed to stand for "Key of Glen and Steve" and the next five characters are what you get by pressing `Shift 12345` on your keyboard.]

- In addition to the cryptographic weakness inherent to DES, there are several vulnerabilities that are specific to the LM Hash algorithm itself. For one, it is easy to guess if the original password string was no longer than 7 characters since in all such cases the second half the input string is all zeros and it results in the predictable DES encryption given by the hex `0xAAD3B435B51404EE`. Another source of great weakness in LM Hash is that the two halves of the hash value can be attacked separately since there were calculated independently. Additionally, ordinarily each character of the 14 character string would be one of 95 printable characters. However, since LM Hash converts lowercase to uppercase, that means that each character can only be one of 69 values. Therefore, the total number of distinct hash values for each 7-character part of the password is $69^7 \approx 2^{43}$, not a very large number for modern desktops. [In general, if the size of the alphabet is k and you want to construct strings of length n from the alphabet, the total number of distinct strings you'll able to construct is k^n — since you will have k

choices at each of the n positions in a string. In this collection of size k^n, every string is of length n. Now suppose we also accept strings of length $n-1$, then you will get an additional k^{n-1} strings, and so on. What that implies is that the total number of password strings (of all possible printable ASCII characters) of length 7 or less is given by $69^7 + 69^6 + 69^5 + 69^4 + 69^3 + 69^2 + 69$.]

- As mentioned at the beginning of this section, **password cracking means that an adversary has somehow gotten hold of the document where all the password hashes are stored and is now trying to figure out the actual passwords from those hashes**. In a Linux machine, the root-readable-only document where all the hashes are stored is `/etc/shadow`. [In a Windows machine, the passwords, I believe, are stored in the `C:\Windows\System32\config\SAM` document. This file, however, may not be directly readable while your machine is up and running. There is an Offline NT Password Tool available at `http://pogostick.net/~pnh/ntpasswd/` that, ordinarily meant for resetting your password on a Windows machine, can also be used to read the SAM file where the password hashes are stored.]

- That brings us to the question of **how to actually reverse-map a password hash to the actual password entered by a user**. Now that disk storage is so cheap, a straightforward answer to this question is to construct a hash for all possible character combinations and to then store these `<password, hash>` values (in the form of `<hash, password>` pairs) in a giant disk-based hash-table database of the sort that are now made available by all major computing languages. [In Linux/Unix platforms, such disk-based hash tables are accessed through what are known as DBM libraries. The

Perl module `DB_File` and the Python module `bsddb` provide very convenient interfaces to this type of disk storage. See Chapter 16 of my book `Scripting with Objects` for further information on how to use such disk-based storage.] Let's say you want to construct this type of a lookup table for attacking the LM Hash password file. As mentioned earlier, you are likely to attack each of the two halves of the password hash separately and, for each half, you have $69^7 \approx 2^{43}$ different possible strings to search through. Since 2^{43} is roughly 9×10^{12} (which, colloquially speaking, is nine trillion) and, assuming for the sake of a simple argument that we can store the inverse mapping from the password hash values to the passwords in the form of a hashtable with no collisions, we would only need to store the seven bytes for each ASCII string. At runtime, when we seek the password P associated with a password hash C, the hashtable access function would convert C into the memory address where P is stored. [Information in hashtables is stored in buckets. Ideally, each bucket would hold a single <key,value> pair, where the key would be the hash of a password and the value the password itself. For a disk-based hash table for LM password cracking, each key C would require 8 bytes and each P 7 bytes. Therefore, each <key,value> pair would require a total of 15 bytes. This implies the hash table would require $15 \times 9 \times 10^{12}$ bytes of storage — that is 135 terabytes of disk storage. Considering that RAID array storage is now under $50/terabyte at some of the vendors, creating a full lookup table for attacking the LM Hash passwords is not that out of the question any longer.]

- If the size of the disk space mentioned above seems large, you can reduce the space needed considerably if you assume that random juxtapositions of the characters are unlikely to exist in a password. You can construct lookup tables whose sizes are only a few gigabytes by just using concatenations of meaningful word fragments. If the passwords are short enough, such lookup tables can be deadly effective in instantly revealing a user's password string.

- When a password hash is attacked by looking up a table of previously computed hashes, we refer to that as the lookup-table attack (in order to distinguish it from the rainbow table attack I'll address next). Note that an adversary may not even have to compute the hashes for a lookup-table attack. You can acquire such lookup tables either for direct download or on physical media from various vendors on the internet. Ostensibly, this is legitimate business as it allows network administrators to test the strength of the user passwords. But, obviously, nothing prevents bad guys from using these tables to crack password hashes.

- If you still believe that the disk storage needed for a lookup table attack is much too large for the sort of password hashes you want to attack, or if your goal is to attack (or, say, to attempt attacking) longer passwords, you are going to need the rainbow tables.

- The idea of rainbow tables was invented by Phillipe Oecshlin and is described in his paper "Making a Faster Cryptanalytic Time-Memory Trade-Off" that appeared in Lecture Notes in Computer Science in 2003.

- In order to understand how a rainbow table is constructed, you have to first understand what is meant by a hash chain and how such chains allow you to trade time for memory. That is, in comparison with the memory required for constructing a hash for every possible password (and then using it subsequently as a lookup table to determine the password that goes with a hash), hash chains requires reduced memory but at the cost of having to spend more time to get to the password (most of the time).

- Fundamental to the notion of a hash chain is a *reduction function*. A reduction function maps a hash to a character string that *looks* like a password. There is nothing extraordinary about a reduction function. You could, for example, take the last few bytes of the hash and create any sort of a mapping from those bytes into the space of all possible passwords. Any mapping that more or less uniformly samples the space of all possible passwords is a good enough mapping. We can certainly expect that a reduction function may map more than one hash to the same password. **As it turns out, it is a good thing when a reduction function does that.**

- Let p be the plaintext password and c be its hash. Let the hash function that takes us from p to c be denoted $H(.)$. So we have $c = H(p)$. Let's now envision a *reduction function* $R(.)$ that when applied to c yields a string that looks like a plaintext. Let p' be the plaintext that results from applying the reduction function to c. So we can write $p' = R(c)$.

- Given the pair of functions $H()$ and $R()$ as defined above, starting from some randomly chosen plaintext p_1 from the space of all passwords, we can now construct a hash chain in the following manner:

  $$p_1 \longrightarrow c_1{=}H(p_1) \longrightarrow p_2{=}R(c_1) \longrightarrow c_2{=}H(p_2) \longrightarrow p_3{=}R(c_2) \longrightarrow c_3{=}H(p_3) \longrightarrow p_4{=}R(c_3) \longrightarrow \cdots$$

  We will specify the length of the chain by the parameter k. Each link in this chain would consist of one application of the hash function $H()$ and one application of the reduction function $R()$. We store in a table just the starting plaintext p_1 and the ending plaintext p_k.

starting point plaintext	*endpoint* also plaintext after k steps of $R(H(p_k))$
p_1^1	p_k^1
p_1^2	p_k^2
p_1^3	p_k^3
$\ldots$	$\ldots$

- Let's say that a password cracker wants to use the above table to crack a given hash C. The cracker creates a chain — let's refer to as the test hash chain — by first applying $R()$ to C get $q_1 = R(C)$, and then applying $H()$ to q_1 to get $d_1 = H(q_1)$, and so on. The test chain will now look like:

  $$q_1{=}R(C) \longrightarrow d_1{=}H(q_1) \longrightarrow q_2{=}R(d_1) \longrightarrow d_2{=}H(q_2) \longrightarrow q_3{=}R(d_2) \longrightarrow \cdots$$

  If any of plaintext passwords in this chain — meaning if any of $q_1, q_2, \cdots$ — match any of the endpoints in the second column of the table shown above, then there is a high probability that the password that the cracker is looking for is in the chain corresponding to that row.

- In other words, if the plaintext string q_m for some value of m in the test hash chain generated from the hash C matches, say, the endpoint entry p^i_k in the second column of the table, the cracker can expect with a high probability that the password associated with C is in the chain that corresponds to the i^{th} row of the table. The starting point in this row is given by p^i_1. The cracker will now regenerate the chain for the i^{th} row of the table. The regenerated chain will look like:

$$p^i_1 \longrightarrow c^i_1{=}H(p^i_1) \longrightarrow p^i_2 = R(c^i_1) \longrightarrow c^i_2{=}H(p^i_2) \longrightarrow \cdots\cdots \longrightarrow c^i_{k-1}{=}H(p^i_{k-1}) \longrightarrow p^i_k{=}R(c^i_{k-1})$$

  With a significant probability, the cracker will find that his hash C matches one of the hashes in this chain. [Note that the hash C that the cracker wants to crack can be anywhere in the chain.] Once a match is found, the password that the cracker is looking for is the plaintext that immediately precedes C in the chain.

- That leads to the question of how long to grow the test chain starting with C as we look for plaintext matches with the endpoints in the table. The answer is that if the test hash chain was grown through k steps, which is the same number of steps used in the hash chain table, and if no plaintext matched with any of the endpoints, then the password that the cracker is looking for does NOT exist in any of the chains stored in the table.

- Additionally, let's say that as we grow the test hash chain one step at a time starting with the hash C to be cracked, we run into a q_m that matches one of the endpoints in our table, but we are unable to find C in the chain for that row. In such an event, we continue to grow the test chain and look for another q_n that matches one the endpoints in the table. But, obviously, we do NOT grow the test hash chain beyond the k steps.

- When we run into a q_m that matches one of the endpoints in the table but when the chain for that row does not contain the hash C we are trying to crack, we refer to that as a *false alarm*.

- Ideally, the hash chain table should have the property that the passwords stored implicitly in all the chains should span (to the maximum extent possible) the space of all possible passwords. This is for the obvious reason that if a legitimate password is neither a starting point, nor an endpoint, and nor in the interior of any of the chains, then there would be no way to get to this password from its hash. Said another way, if a password is NOT reduced to during the construction of the hash chain table, then that password cannot be inferred from its hash.

- Whether or not the requirement mentioned above can be met in practice depends much on the reduction function $R()$. Note that any choice for $R()$ will map multiple hashes to the same password string. So it is possible for two chains to contain the same password string. Say Chain 1 contains a specific password at step i and Chain 2 has the same password at step j with $i \neq j$. Now the two chains will traverse the same transitions even though their endpoints will be different. The endpoints will be different because the number of remaining steps in the two chains in the two chains is not the same. Because the endpoints will be different, Chain 1 and Chain 2 will occupy two different rows in the table even though the passwords stored implicitly in the two chains show significant overlap. When two different chains in a table overlap in this manner, we refer to that as a *collision*. This overlap cannot be detected because we only store the starting points and the endpoints for the chains. Nonetheless, such implicit overlaps can significantly reduce the ability of a hash chain table to crack a hash because of the reduced overall sampling of the space of all the passwords.

- It is this overlap between the hash chains — also referred to as the *merging* of the chains — that places an upperbound on the size of a hash chain table. Ordinarily, you would want to construct a hash chain table for a large number of randomly selected starting points in the space of all passwords. But, as the size of the table grows, the table becomes more and more inefficient on account of chain merging. Before the invention of rainbow tables, this problem was taken care of by constructing a number of hash chain tables, each with a different reduction function $R()$.

- With rainbow tables, instead of constructing a number of hash chain tables with different reduction functions to overcome the problem of chain merging, you now construct a single hash chain table, but now you use k different reduction functions, $\{R_1(), R_2(), \cdots, R_k()\}$, for each of the k steps in the construction of a chain. For a collision to now occur, the password that is reduced to must be the output of the same reduction function — an event with much lower probability than was the case with hash-chain tables as presented above. This also takes care of one more problem with the old-style hash-chain tables. You see, in hash-chain tables as explained above, there is always a possibility that you will encounter a loop as you grow a chain. Since a reduction function is intentionally many-to-one, there is always a chance that the password that is reduced to will be the same at two different places in a chain. [Obviously, this can also happen in a test hash chain.] As with chain collisions, such loops reduce the efficiency of a hash chain table. However, when you use different reduction functions for the successive reduction steps in a chain, you are less likely to run into loops.

- Using k different reduction functions in growing a hash chain calls for a change in the lookup procedure. By lookup we mean querying the hash chain table with the hash C that you want to crack. The lookup consists of first applying the last of the reduction functions $R_k()$ to obtain, say, $q_1 = R_k(C)$ and then checking whether q_1 is an endpoint in the rainbow table. If not, we grow the test chain by calculating $q_2 = R_{k-1}(H(q_1))$ and search for q_2 as an endpoint in the table. If a matching endpoint cannot be found for q_2, we grow the test chain by one more step by calculating $q_3 = R_{k-2}(H(q_2))$; and so on.

- There are several websites that provide pre-computed rainbow tables for different hash functions. When the hashing function is MD5 and for password strings that go up to 8 characters, you can obtain the pre-computed rainbow tables from

  `http://www.freerainbowtables.com/en/tables2/`

  And here is a website devoted to GPU accelerated implementation of rainbow table attacks:

  `http://project-rainbowcrack.com/`

Back to TOC

24.5 Password Hashing Schemes

- Now that you know about password cracking, the very first thing you need to become aware of is the fact that there do not yet exist any tools for cracking passwords that are hashed with state-of-the-art password hashing schemes that use variable "salts" and variable "rounds". As to what is meant by "salt" and "round" will become clear from the presentation in this section. An example of such a state-of-the-art password hashing scheme is `sha512_crypt`. I'll have more to say about this scheme later in this section.

- Before launching into how modern password hashing schemes work, I do want to mention the wrong impression created by the following sort of statements one often runs into: "*Passwords are stored as hash values*," "*Hash values for passwords that are not sufficiently long*," etc. Taken at their face value, such statements seem to imply that when a user provides a password, it is straightforwardly supplied to a hashing function, such as those described in Lecture 15, and the result stored somewhere in the system. This may have been true for some of the older methods for creating password hashes, nothing could be farther from the truth for the state-of-the-art schemes for converting user-entered passwords into their hashes.

- The main reason why you cannot just directly apply an algorithm such as SHA-512 to a user-entered password string is because the resulting hash values would still be crackable despite the fact that hash function itself is cryptographically secure and possesses the one-way property defined in Lecture 15. [To explain this issue, let's say there are no constraints placed on the lengths of the passwords chosen by the users. Assume for the sake of argument that the passwords used by some folks have only six characters in them and they all consist of lowercase letters. Total number of such passwords that can be composed with exactly six characters is only $26^6 = 308915776$. Given a hash of such a password, even when that hash is produced by, say, the cryptographically secure SHA-512 algorithm, it would be trivial to construct a lookup table for all such hashes and acquire the password in less time than it takes to blink an eye. Now imagine an intruder who has no desire to crack all the passwords in, say, the `/etc/shadow` file maintained by the network administrator. All that the intruder wants is to break into just a couple of accounts where he/she can install his own software. For such an intruder, just being able to crack short passwords is good enough.]

- To make it virtually impossible to carry out the sort of attack described in red above, all modern password hashing schemes combine a user's password string with a number of random bits that are known as the salt. Before I explain what salt is and why it makes it virtually impossible to crack a password — even the short ones — let's look at how the hash value of a password is actually stored in `/etc/shadow`: [If you execute '`man shadow`', you will realize that each line in the file `/etc/shadow` consists of 9 colon-separated field. The first field is always the username; the second field is the password hash that is shown below; the third field the date of last password change; the fourth field the number of days the user must wait before he/she is allowed to change the password; the fifth the number of days after which the user will be forced to change the password; and so on. Shown below is what is stored in the second field — the password hash field — for some user.]

```
$6$rounds=40000$ZVzZ72hf$Tf19cHUK0g.nf.I/Bpn5jd3jokKMEAIHssRW20EUGfneuTUzkhNmGv9iDhjfeDpJtq0yGjtSeXSq8
```

- What is shown above, although nominally referred to as a password hash, is in actuality the MCF (Modular Crypt Format) representation of a password hash. With MCF, a password hash looks either like

  ```
  $<identifier>$rounds=<number-of-rounds>$<salt>$<password-hash>
  ```

  or, when the "number of rounds" is set to its default value 5000, like

  ```
  $<identifier>$<salt>$<password-hash>
  ```

  Therefore, in the example shown above, what is stored for the password hash in `/etc/shadow` for a user consists of:

  ```
  identifier:          6

  number of rounds:    40000

  salt:                ZVzZ72hf

  actual hash value:   Tf19cHUK0g.nf.I/Bpn5jd3jokKMEAIHssRW20EUGfneuTUzkhNmGv9iDhjfeDpJtq0yGjtSeXSq8
  ```

- The "identifier" shown above refers to the *Password Hashing Scheme*. Note that there is more to a password hashing scheme than just a hashing algorithm. Of course, as you would guess, all modern password hashing schemes use a hashing algorithm and it is commonly the case that the name of a password hashing scheme includes a mnemonic for the hash algorithm used by scheme. Also, the name of a password hashing scheme typically ends in the substring "crypt," as illustrated by the table shown below that shows the identifiers used for today's more important password hashing schemes:

Password Hashing Scheme	Identifier
md5_crypt	1
bcrypt	2
bcrypt	2a
bcrypt	2x
bcrypt	2y
bsd_nthash	3
sha256_crypt	5
sha512_crypt	6
sun_md5_crypt	md5
sha1_crypt	sha1

Note again that, except for `bsd_nthash`, the names of all the Password Hashing Schemes mentioned above end in the substring "crypt". [The `bcrypt` password hashing scheme is used in Unix/Solaris systems. The underlying hashing algorithm in `bcrypt` is based on the Blowfish cipher I mentioned in Section 3.2 of Lecture 3 as a variant of DES. The password hash output by `bcrypt` omits the separator character '$'.] The table I have shown above is reproduced from `http://packages.python.org/passlib/modular_crypt_format.html`. As mentioned there, MCF is not an official standard, but a commonly used format today for storing password hashes.

- Getting back to the `/etc/shadow` entry for a password shown on page 42, you can now tell that the password hash shown at the bottom of that page was generated by the `sha512_crypt` password hashing scheme.

- Let's now examine the second field of the `/etc/shadow` entry for the password hash shown earlier in this section. This entry says: `rounds=40000`. As you will soon see, modern password hashing schemes hash a password (along with its salt – whose meaning will soon be explained) multiple times. **You might ask: To what purpose?** You are even more likely to raise this question after you realize that an intruder who has stolen the `/etc/shadow` or an equivalent file can see the number of rounds applied by the password hashing scheme. So, in order to crack a password hash, the intruder could use the same number of rounds. Note that the intruder already has access to the password hashing scheme used since they are all in the public domain. For the answer to this very reasonable question, read on.

- By hashing a multiple number of times, you make it that much harder to crack a password through any sort of a table lookup, rainbow or otherwise, especially if the number of rounds is randomly chosen for each user account. Even though some state-of-the-art password hashing schemes can generate a password hash with any number of rounds, most password hashes are computed with a default value for the number of rounds — 5000. The reason for that is that the protection provided by salts is considered to be strong enough to thwart any lookup-table based attacks for several more years to come.

But should computers become even more powerful and should massive disk storage become even more inexpensive, the additional protection made possible a variable number of rounds would certainly be put to greater use. [There is also a minimum and a maximum on the number of rounds. The minimum is 1000 and maximum is 999,999,999. Specifying a value below 1000 would cause 1000 to be used for the number of rounds and specifying a value of 1 billion or greater would cause 999,999,999 to be used for the number of rounds.]

- That takes us to the third part of what is stored for a password hash in its MCF representation in the second field of a file like `/etc/shadow` — the salt. **A salt is simply a randomly chosen bit pattern that is combined with the actual password before it is hashed by a hashing algorithm.** The salt used in the `/etc/shadow` entry shown earlier is `ZVzZ72hf`. These are eight Base64 characters, each standing for six bits. Therefore, this salt consists of a 48-bit word that will be combined with the user's password before hashing.

- Assume that my password is as simple as, say, the ASCII string "avikak". This password consists of only 6 characters. Assuming these to be ASCII characters and using 8-bit encoding for each character from the ASCII table (despite the fact that the MSB for all the printable characters in the ASCII table is 0), my actual password consists of a bit stream that contains 48 bits. Using the same salt as shown above, I may prepend the 48 bits of the salt to the 48 bits of the password "avikak" to form a 96 bit input to the hashing function. In actual practice, a password hashing scheme is likely to create a repetitive concatenation of the salt bits and the password bits to form a bit pattern that is hashed. The precise nature of this concatenation and repetition depends on the password hashing scheme used.

- If, as a system admin, I use a different salt for each different username, it would be impossible for an adversary to use a precomputed table of any sort for inferring the passwords from their hash values. Obviously, the intruder who stole the `/etc/shadow` file knows the salt used for each username. Nonetheless, he/she would not be able to use *precomputed* rainbow tables available on the web for cracking the passwords. And it would simply take much too long (possibly years) for the intruder to create his/her own rainbow tables that accounts for every possible value of the salt.

- In general, if you use an n-bit salt, the size of storage needed for password cracking through table lookup goes up by 2^n. So a 48-bit salt results in the size of this storage for mounting a lookup type attack going up by a factor 2^{48}. Typically, up to 16 Base64 characters are used for salt — that makes for a maximum of 96 bits of salt — with the result 2^{96} variability in the hash value of a given password string.

- **Note that a side benefit of using a random value for salt is**

that it makes less likely that any two usernames will have the same password hash associated with them. In any enterprise level system, there is always a chance that multiple people will use the same mnemonic string as a password. So without salt, one could end up with a number of people with exactly the same password hash for a set of different usernames. Imagine what a bonanza that would be for an intruder who wants to take over as many user accounts as possible with minimal work.

- The password hash shown earlier is in the Base64 representation for the bit patterns for both the salt and for the actual hash. It is important to keep in mind, however, that the Base64 representations as used in a password hash may NOT correspond to the MIME-compatible Base64 encoding you have seen in these lecture notes so far. In the Base64 encoding used in password hashes, all you are guaranteed is that the encoding is being carried out by converting 6-bit binary strings into printable ASCII characters, but that the mapping used in this conversation may differ from one password hashing scheme to another. [The Python library `passlib` provides the MIME-standard Base64 encoding through `passlib.utils.BASE64_CHARS`. For Base64 encodings as used in `sha512_crypt`, `sha256_crypt`, `md5_crypt`, the same library provides the encoding through `passlib.utils.HASH64_CHARS`, etc.] **The Base64 encodings as used by password hashing schemes are also known as Hash64 encodings.**

- Now that you know about the purpose of salts and rounds in

password hashing schemes, it's time to become familiar with the logic of an actual password hashing scheme. Your goal should be to understand how a hashing algorithm is used in a password hashing scheme. Toward that end, I recommend that you read the specification document for the `sha512_crypt` password hashing scheme: "Unix crypt using SHA-256 and SHA-512" by Ulrich Drepper that is available at
`http://www.akkadia.org/drepper/SHA-crypt.txt`.

- The `sha512_crypt` password hashing scheme is a SHA-512 based culmination of a series of password hashing schemes that owe their origin to old Unix `crypt()` function. [**Just for historical interest, do "man crypt" on your Linux machine to find out more about the now ancient `crypt()` function. It creates a password hash by encrypting a constant string of all zeros with the DES algorithm with the key being the user-supplied password. The 56-bit DES key is constructed by taking the lowest 7 bits of the first 8 characters of the password entered by the user. For obvious reasons, `crypt()` is not considered secure any more.**] It is interesting to contrast how password hashing used to be carried out in the old `crypt()` function with how it is carried out in `sha512_crypt`. To give the reader just a flavor of what is done to the user supplied password string for the computation of its hash, a scheme such as `sha512_crypt` first creates multiple replications of a concatenation of the user-supplied password string, the salt, followed again by the password string, the number of such concatenations used being the number 64-byte blocks in the original password string (with provision for the password length modulo 64).

- Python's library for a large number of password hashing schemes is called `passlib`. It can both create password hashes and verify a user-entered password. This is the library you would want to use if you wanted to create a multi-user application with a Python frontend for password based security. The following URLs are useful for accessing `passlib`'s API and other documentation:

  http://pythonhosted.org/passlib/password_hash_api.html

  http://packages.python.org/passlib/contents.html

- The names of all password hashing schemes in `passlib` end in the suffix "_crypt". And all such schemes define the following two methods

```
encrypt()
verify()
```

  the first for generating a password hash and the second for verifying a user-entered password against its hash in the memory. For example, suppose my password was "avikak" (which, by the way, it is not; so don't get any ideas about breaking into my machine). If I call

```
hash = passlib.hash.sha512_crypt.encrypt("avikak")
print hash
```

  I'll get the following output for the password hash:

```
$6$rounds=40000$zJ1zd4BOmLiJCrRA$t96c5xt7cwlXxw7xr3d8ltpHp3sjH.kCJxn2EcHyizt791qtSJyL3cI3bi/jlLeY6VrZMt0.zDzZiN5eohX/J1
```

As you can see, `passlib` uses a default of 40,000 rounds and 16 Base64 characters for the salt. On the other hand, if I want to set the number of rounds to the more universal default of 5000, I can call

```
hash = passlib.hash.sha512_crypt.encrypt(‘‘avikak’’, rounds=5000)
print hash
```

I get the following for the password hash:

```
$6$ABd0TbzfFDtm3gde$ePE12Bl8AFVXP.0H5gPyCT0eXGwX0.zxflR/9U05dQ27ILAbHMiX0EjVLcB3Rio/8wI7mBIVfoKo7ZJKYbILW0
```

Note that this password hash does not explicitly mention the number of rounds because the number 5000 is universally acknowledged to be the default value for this parameter. Here are some additional examples of calls to the `passlib` library for creating password hashes:

```
print passlib.hash.sha512_crypt.encrypt(‘‘avikak’’, rounds=5000, salt_size=8)

print passlib.hash.sha512_crypt.encrypt(‘‘avikak’’, rounds=5000, salt="ZVzZ72hf")

print passlib.hash.sha512_crypt.encrypt(‘‘avikak’’, rounds=40000, salt="ZVzZ72hf")
```

Back to TOC

24.6 Federated Identity Management

- User authentication is becoming increasingly distributed. It is now common for websites to grant you access to some or all of their resources based on your login credentials at Twitter, Facebook, Google, etc.

- Let's say you have a small business that provides some sort of a service to the paying customers. When the customers log in and supply their identity credentials, how should you authenticate them? In the old days, your only option was to run your own password manager. However, there can be significant costs associated with that. Perhaps the biggest issue related to running your authentication server is the security of the user ID data in the server. You can easily imagine the consequences of someone breaking into your system and stealing the user ID data — it could ruin your business. [There is another issue here that is also important: If every organization did its own authentication of the user credentials, just imagine how many different username/password combinations a user would need to keep track of. In general, an informed user would not want all his/her usernames and passwords at the different sites to be the same for security reasons.]

- But now there is an alternative: As a small-business owner, you can use an Identity Provider (IDP) to authenticate the users when they log in. You have surely been to websites where you could log in with your Google or Facebook or Twitter credentials. Those websites were using these popular social media companies as Identity Providers. An IDP typically has a special website for the benefit of small businesses that shows how their identity verification services can be used.

- So if user authentication is to be entrusted to a third party, what should be the rules of interaction between the three parties involved: (1) the user; (2) the service provider; and (3) the identity provider?

- The following three frameworks/protocols provide answers to the question posed above:

OAuth : Focusing on the version 2.0 of OAuth, it is an authorization framework that specifies how a server or a website in the internet can accept a user's login credentials on behalf of another server or website. For example, assuming that the website for a restaurant has a password protected page for some of its more private services, it may ask you to login with your Twitter credentials by clicking on a button. As explained later in this section, clicking on that button causes the user's browser to be redirected to Twitter's login page where you would be asked to enter your ID credentials. The identity credentials you enter in that page would go directly to a Twitter server for their authentication. And, after they are authenticated, the Twitter server would issue an "authorization ticket" to the restaurant web server for accepting you as a verified customer. All these communications would be governed by the OAuth 2.0 framework. An important aspect of this scenario is that the identity provider (in this case, Twitter) does not have to share the user login credentials with the service provider (the restaurant). The OAuth 2.0 standard is described in the document RFC 6749.

OpenID : Whereas the OAuth framework deals primarily with the interaction between two web entities for the purpose of one entity supplying login credentials based authorization to the other for accepting a user, the processing and the verification of the identity credentials supplied by a user and how some of that information would be sent back to the service provider would typically be handled by the OpenID protocol.

SAML : SAML (Security Assertion Markup Language) is the oldest of the three frameworks/protocols listed here. It is used by large enterprises to implementation SSO (Single Sign-On) that allows for a single log-in by a user at a given site to access the other sites and services run by the enterprise.

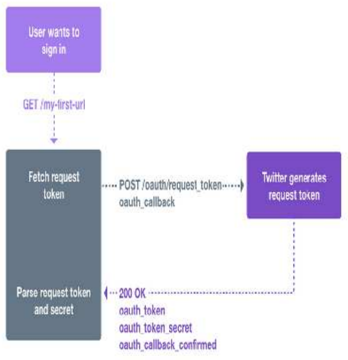
- In what follows, I'll start with an example of how Twitter uses the OAuth framework to allow its identity servers to be used by other service providers for user authentication. This example is from Twitter's webpage at

  `https://developer.twitter.com/en/docs/basics/authentication/guides/log-in-with-twitter`

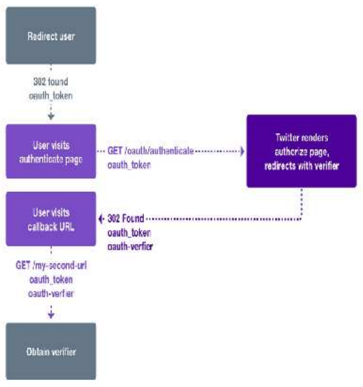
- As shown in the figure that follows, there are three steps involved in how Twitter allows a service provider to use its identity verification server. The first step takes place when you as a user clicks on the login button at the webpage of the service provider (let's say it's a restaurant). That click by you sends an

OAuth `request_token` to the Twitter server. After Twitter verifies that the request is from a business that it has agreed to provide identity services for, it sends back to the restaurant's web server the 200-OK status code (which implies success followed by content creation by the sending party in the HTTP protocol), an `outh_token` along with an `outh_token_secret` using the SSL/TLS protocol for confidentiality.

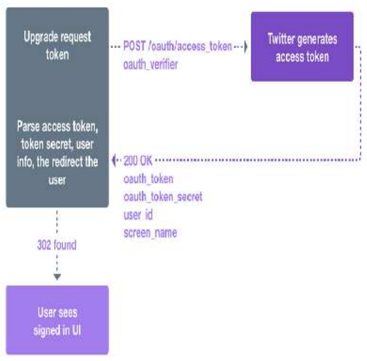
- Step 2 is initiated with the user's browser receiving a URL redirect, which corresponds to the browser receiving the HTTP status code "302 Found". As shown in the Step 2 figure, this takes the user to a Twitter login page for entering the identity credentials. If successfully verified, the user's browser receives a second URL redirect to the login verification page. And that concludes Step 2.



Step 1



Step 2



Step 3

- Subsequently, in Step 3, after the login credentials supplied by the user are authenticated, the user's browser receives one final browser redirect that takes the user to the restaurant's access-controlled webpage that the user wanted to visit in the first place. But the success of that redirection is subject to the service provider (the restaurant's web server) receiving an access token shown in the figure for Step 3. The access token received from Twitter also contains information regarding the user (name, location, etc.) for the benefit of the service provider as shown in the Step 3 figure.

- As explained above, OAuth is about a designated 3rd party e-commerce server (like Twitter) *authorizing* the service provider (like a restaurant) to accept the user as a legitimate client. That's why OAuth is referred to as an *Authorization Framework*. OAuth is more about the interaction between the identity provider's server and the service provider's website than about the identity verification itself. That takes us to the second of the three frameworks/protocols mentioned previously in this section about federated identity management — the OpenID protocol. Version 2 of OAuth uses OpenID as the user authentication layer in the form of "OpenID Connect (OIDC)". You can think of OIDC as a specific implementation of OpenID that provides an `ID_token` to encode the user's identity which is subsequently delivered to the service provider. More generally, though, OIDC is considered to be a "profile" of OpenID.

- That brings us to the third of the federated identity management protocols mentioned previously: SAML. This is the oldest of the three frameworks/protocols and was meant to do together what OAuth and OpenID do separately. While mobile applications that require user authentication to be carried out by a 3rd party server have generally switched over to OAuth and OpenID, larger enterprises are continuing to use SAML for what's known as SSO (Single Sign-On) that requires a user to log in only once for the different e-services within the enterprise. Specific to SAML is the use of what the protocol refers to as an "assertion" that is a digitally signed XML document whose different tags stand for the issuer that authenticated the identity, attributes related to the user who was authenticated, etc.

- Before ending this section, I want to say a few words about a potential security vulnerability in OAuth. Imagine a rogue business masquerading as a restaurant that wants to steal user login credentials. Now recall the URL redirects I mentioned in my explanation of OAuth using the Twitter example. Remember, when you clicked on the login button on the restaurant's webpage, that was supposed to take your browser to a Twitter log-in page through a URL redirect received from Twitter. Now imagine the situation in which the restaurant's web server traps the outgoing call when you click on that button and redirects your browser to a log-in page that looks deceptively like the real Twitter login page. You can easily imagine the rest of such a security exploit.

Back to TOC

24.7 HOMEWORK PROBLEMS

1. As you now know, Fail2Ban protects your computer by updating the iptables based firewall rules. In Section 24.3, when I showed an example of these rules, it was based on the assumption that initially all the chains in at least the filter table of the firewall were empty. I also did not show an example of the rules after an IP address is banned. Install Fail2Ban in your computer and construct a demonstration that illustrates the modification to the firewall rules after one or more IP addresses are banned.

2. As mentioned in Section 24.3, by default the Fail2Ban tool monitors only the `/var/log/auth.log` file for repeated attempts at breaking into a computer through the SSH port 22. It can, however, be made to monitor any of the other log files such as `/var/log/apache/access.log` for access to your HTTPD server, `/var/log/mysqld.log` for access to your database server mysqld, `/var/log/squid/access.log` for access to your Squid proxy server, `/var/log/named/security.log` for access to your bind9 based DNS sever, etc. In order to appreciate the full versatility of Fail2Ban, create your own server application — based on, say, the server scripts you have seen elsewhere in these lecture notes. Make sure that your server application has associated with it an access log in which the server makes different kinds of entries depending on how a client is interacting with the server.

Now create a filter to recognize some particular type of such client interactions. And when a client is found to engage in such an interaction with the server, either trigger a ban on the client IP address or, at the least, get Fail2Ban to send you an email to that effect. Look at the regex based filters in the directory `/etc/fail2ban/filters.d/` to get ideas on how you can set up your filter.

3. A very educational library for learning about the different password hashing schemes is Apache's Common Codec library. Here is a link to the Apache Commons repository for all kinds of functionality in Java: `http://commons.apache.org/` and here is a link `http://commons.apache.org/proper/commons-codec/apidocs/` specifically to the Digest package of the Codec library that contains the Java class `Sha2Crypt` that implements various SHA-2 based password hashing schemes. In particular, you will find it educational if you look at the implementation of the `Sha2Crypt` class. This implementation mirrors on a step-by-step basis the previously mentioned specification of `sha512_crypt` by Ulrich Drepper at `http://www.akkadia.org/drepper/SHA-crypt.txt`. As one might expect, the defaults with respect to the salts, the rounds, etc., in the Python based `passlib` and in the Java based `Sha2Crypt` are not the same. The goal of this homework is to become familiar with the defaults in the two implementations of Ulrich Drepper's specification of `sha512_crypt` so that they produce the same password hashes for a given password string. That is, either by default or by specific mention, you want the two

implementations to use the same number of rounds and the same salts.