Last update; 4/30/2018
Author: Jim Martin, Clemson University (jmarty@clemson.edu)

# Unix/Linux Login Authentication - A Tutorial

This document is a brief tutorial on authentication. The focus is based on a context of a user logging into a system using a secret password.

## Terminology

Let's begin by defining terminology.

- A hash function is a one-way mathematical function applied to input data. Depending on the context the data might be a message or a password.
- Symmetric encryption- requires a shared secret key at both the encryption end and the decryption end.
- Asymmetric encryption- An alternative to symmetric encryption allows users to use public / private keys which addresses the key distribution problem.
- Authentication : ensures aspects of an exchange or communications that might include the data and/or the identify of the sender. We refer to these two aspects as message authentication and user authentication.
- Pseudo random number generator (PRNG) : an approach to generate random numbers in the range [0,1]. A good PRNG for crypto will have a period (the number of unique, random numbers generated before the algorithm begins generating the same sequence of random numbers) that is very large, produces numbers that reflect a uniform distribution between [0,1], and produces numbers that are not correlated.
- Key: assuming an algorithmic cryptography context, a key is typically a fixed number of bits that are input to a encryption/decryption algorithm to produce the ciphertext or cleartext respectively.
- Keystream: a stream of random numbers typically generated by a PRNG that are used in an encryption/decryption process. A keystream is generally used in streaming oriented ciphers.
- Key generation function - producing one or more secret keys from a secret master key.
- Password: shorter than a passphrase – usually 8 - 12 characters with at least 1 or more character that is not a number or letter (e.g., underscore or comma)
- Passphrase: typically longer than a password – 30 -50 characters. Many times we try to make it easier to remember by using a phrase. A good passphrase of 50 characters can be hashed to create a robust 80 bit key
- A salt, similar to a nonce, is a random piece of data that is used as additional input to the hash function that is used to authenticate a user who is logging in. A salt is used to address the limited entropy (roughly the measure of potential randomness) that a typical password contains. A practical translation of this statement is that a salt makes hashed password more robust against dictionary attacks. A salt also ensures that two users that select the same password have a different hashed password.

Last update; 4/30/2018
Author: Jim Martin, Clemson University (jmarty@clemson.edu)

## Message authentication versus User Authentication

*Authentication in the context of message authentication* – the objective is to ensure that when Bob receives a message from Alice, Bob knows that the message is indeed from Alice. The generalized description involves Alice with a message (M) to send to Bob. Alice runs the message through an authentication program that produces a unique fingerprint of M which we will refer to as the authenticator. In practice, an authenticator might be any of the following: hash, message digest, fingerprint, message authentication code, digital signature. There are three requirements for secure message authentication:

- Exhibit robustness with respect to the 'one-way property' - a malicious user, such as Trudy, that has access to the message (which might be sent in cleartext) and the authenticator (referred to as any of these terms: hash, message digest, fingerprint, message authentication code) is prevented from being able to modify message including the sender information without having the recipient detect the change.
- Collision free – it is highly unlikely that there could ever be two (different) messages that lead to the same authenticator.
- Secure authentication – an extension of the first requirement. To ensure Trudy can not modify the message and update the authenticator, the authentication algorithm needs to involve a shared secret key available to Alice and Bob, but not Trudy. Symmetric or public key methods can be used.

A simplified discussion of how secure authentication might work. The example assumes that Alice sends message M to Bob with secure authentication:

  - Let's assume a Message Authentication Code (MAC) technique that requires a shared secret (s), usually called an authentication code.
  - Alice hashes the M+s producing a secure hash (the MAC).
  - Alice sends M+MAC to Bob
  - Bob receives M+MAC, hashes M+s (Bob has the shared secret authentication code, s) producing MAC'
  - Bob compares the MAC attached to M with MAC' – if they are the same, M has not been modified (well….unless Trudy also has s !!)

*User authentication in the context of user logins with a* password – This is sometimes referred to as end user authentication. The objective is to allow user access to system resources based on information the user has (a secret password). The system is required to maintain this password in a secure manner. The assumption is that Trudy might gain access to the password database on a system. The passwords must be stored in a manner that makes it computationally infeasible for Trudy to deduce the users password.

- Context of user Alice logining to our Ubuntu VM.
  - Alice enters her password, M.

    - The login process concatenates Alice's salt (s) from her information in /etc/shadow.
        - When the passwd was first created or last updated, a new salt ( a random number) is added
    - The login process invokes a hash function on M+s, producing a hash, h.
    - The login process compares h with the authenticated/hashed password saved in /etc/shadow.
- The method does not store 'encrypted' passwords in /etc/shadow. It stores the SHA-512 has that is computed with an input message of a password+salt.

We all have experienced the latest forms of 'multi-factor authentication' that attempt to authenticate with at least two of the following methods:

- Based on something the user knows (a password)
- Based on something the user has (a security token, our DUO procedure can be used to provide a token)
- Based on something that physically (and uniquely) identifies the user (a biometric like the latest smartphone fingerprint-based login)

Authentication based on something the user knows requires the user to enter her user name and password. The system must maintain users passwords in a secure manner even in the event that a malicious user gains access to the files containing passwords. Unix stores passwords in hashed form in the file /etc/shadow. The hash algorithm that is used (typically SHA-512 on Ubuntu 16.04 systems) produces a hash that offers a robust 'one-way ' property. This means that that if used correctly, it is computationally infeasible for a malicious user to deduce ('crack') the user's cleartext password from the hash. The 'if used correctly' boils down to the robustness of the password.

Users must login to gain access - therefore the system maintains a file with user login information. Unix access control is primitive but it does provide some level of access control on any data that is stored in files. The system provides a hierarchical level of access control – regular users can be forbidden from accessing critical data files. Users are able to access certain critical data files in specific cases – typically through the use of the Unix setuid feature. We explain this in the following paragraphs.

The critical data associated with user accounts is located in the files in /etc :

- /etc/password : contains plain-text information about each user's account (home directory, preferred shell, …)
- /etc/shadow : contains info required to authenticate a user. The user's password in an encrypted format, along with the salt that was randomly selected/used when the password was created. Note that users do not have rw access.

As we point out later in this document, the purpose of using a salt is primarily to address the limited entropy (roughly the measure of potential randomness) that a typical password contains….many users might use portions of the same passwords.

To generate a fixed length key from a short password will not produce a very secure key space and might be readily cracked. Adding a random salt to be used with the password to generate a key (the hash output) makes it MUCH more difficult to crack the password using a dictionary –based attack. Clearly a salt addresses the problem of users using weak passwords OR if two users happen to select the same password.

To generate a new password, a hash is run against the salt concatenated with the password. The salt is concatenated again with the resulting hash from the function and that string is what is stored in the password database /etc/shadow. The reason for the shadow file is because the /etc/password file needs to be world readable on a unix system. They used to store the encrypted password in the 2nd field of the /etc/password file. To improve the security, the crypto information from the password file was moved to the shadow file and only gave root access to that file.

The Unix setUID permission bit on the passwd program is set which causes the user's process running the program to have its UID bit set to that of the owner of the password and shadow files (which is root) allowing the user temporary access to the file.

Note that modern Linux systems likely use more robust authentication methods that the traditional unix method described above. The Pluggable Authentication Modules (PAM) is widely used…and can bypass or augment the existing Unix authentication. What some administrators do is to use PAM for end user authentication and to monitor access of /etc/passwd in a 'honeypot' manner - assuming a hacker that breaks into a system will first attempt to learn about other users by looking at /etc/passwd. The idea of a 'honeypot' has been around for decades and is effective for detecting hackers. However, there are now a set of hackers that have the skills/resources to easily identify traditional honeypot environments. A newer approach to honeypots is emerging that creates a more dynamic honeypot environment that is much more realistic and consequently more difficult for a hacker to detect. Emerging Intrusion Detection Systems (IDSs) are blending virtual reality and even gaming techniques to 'interact' with hackers to gather data and information on the hacker and her capabilities[1].

## Example

[1] Check out the presentations from the most recent RSA conference (see https://www.rsaconference.com/events/us18/presentations )

Last update; 4/30/2018
Author: Jim Martin, Clemson University (jmarty@clemson.edu)

This example shows what happens when a user chooses their password on a Unix system, how it gets stored in /etc/shadow and then how the password is checked when the user logs in. We ignore some details on real Linux systems such as pam to make this a bit simpler.

1. User 'monroe' runs the passwd program and chooses the password 'HorseStapler+5'
2. The passwd program receives the password, checks it (for validity rules), then it determines what algorithm it needs to use for hashing, (DES, MD5, SHA-256, SHA-512) Let's say the system chooses SHA-512 for its hashing algorithm.
3. The SHA-512 algorithm uses a 16 character random string for its salt made up of alphanumeric characters + '.' and '/' . The system generates a random salt string 'BohpaS.aul0Qua/t'.
4. The system now concatenates the salt onto the password like this 'HorseStapler+5BohpaS.aul0Qua/t'. This is the "new" plaintext that will actually be hashed and stored. It makes it more unique so that anyone else using the password 'HorseStapler+5' because of its popularity will not result in the same hashtext in the password database. It also protects against rainbow table attacks.
5. The system runs the hash algorithm on the new string SHA512('HorseStapler+5BohpaS.aul0Qua/t') and gets the hashtext output 'IVUen1dSKZ634jM.KLQ1Am/WPh..DSO2MYI53qffac2IFzESKwIufyVjzQGlxNenOXGehMTCdSoL9DLPe6Zfm1'
6. So that the system can authenticate the user in the future, it stores the salt as well an indicator for the type of hashing used in the /etc/shadow file as the following string in the second field (encrypted field) for that user's entry like this '$6$BohpaS.aul0Qua/t$IVUen1dSKZ634jM.KLQ1Am/WPh..DSO2MYI53qffac2IFzESKwIufyVjzQGlxNenOXGehMTCdSoL9DLPe6Zfm1'

An example entry in /etc/shadow looks like this:

```
monroe:$6$BohpaS.aul0Qua/t$IVUen1dSKZ634jM.KLQ1Am/WPh..DSO2MYI53qffac2IFzESKwIufyVjzQGlxNenOXGehMTCdSoL9DLPe6Zfm1:15196:0:99999:7:::
```

The $6$ part indicates that it was SHA-512 hashed. The part in-between the 2nd and 3rd '$' character store the salt that was used. This field in the shadow file is called the encrypted field. Some people don't like that name because its hashed, not encrypted. Encrypted implies that it can be decrypted, which it cannot.

Last update; 4/30/2018
Author: Jim Martin, Clemson University (jmarty@clemson.edu)

Re-authentication

1. Next time user monroe logs in to the system, he submits the password 'HorseStapler+5'.
2. The system looks up the user in /etc/shadow and finds the full string above. The system takes the salt part of it (the part in-between the 2nd and 3rd '$') and then concatenates it with the password like above. Then it hashes that using the SHA-512 algorithm as indicated by the algorithm designator '6' in-between the 1st and 2nd '$'.
3. The system then compares the resulting hashtext against the string after the 3rd '$' in the encrypted password field of the shadow file.