Configuring SSHD Authentication

Generating Private Keys

To generate a key with *no password* using the `ed25519` algorithm, we can run the following command. This will output the generated `private_key` and `private_key.pub` withtin the directory specified after `-f`

> If you intend to use a password for your private key, do not pass it as an option through the commandline! Your bash history should not contain passwords or other sensitive information. Use `ssh-keygen -t ed25519 -f /home/username/.ssh/username_ed25519` and follow the secure prompts instead.

```
ssh-keygen -t ed25519 -p "" -f /home/username/.ssh/username_ed25519
```

Now you can cat out your public key with `cat /home/username/.ssh/username_ed25519.pub` and copy the output to the `/home/remoteuser/.ssh/authorized_keys` file on the remote host you want to access with this private key. Take note of which `remoteuser` you use on the host, as logging in with any other username will fail.

SSH Authentication Configuration

SSH will look for configurations passed to the commandline above all other configurations. Using a command like `ssh user@host.com -p 1234 -i /path/to/private_key` will override the `Port` and `IdentityFile` settings in all SSH configurations by using the commandline options `-p` and `-i` respectively.

User Configurations

If there are no relevant commandline options, SSH will then check for user configurations. Each user may define their own configuration within `~/.ssh/config`. You could construct the entire `ssh user@host.com -p 1234 -i /path/to/private_key` command automatically by running `ssh hostname` if you add the following configurations to `~/.ssh/config`

```
Host hostname
    HostName host.com
```

```
    User username
    Port 1234
    IdentityFile /path/to/private_key

Host ip-host
    HostName 127.0.0.1 # Can also use IPs for HostName
    User username
    Port 1234
    IdentityFile ~/.ssh/private_key # Can reference ~/ for user's home directory
```

If you created your `~/.ssh/config` file manually, you may see the following error when attempting to SSH

```
Bad owner or permissions on /home/username/.ssh/config
```

SSH requires that this file is readable and writable only by the user it is relevant to. So to fix this, we run the following command

```
sudo chmod 600 ~/.ssh/config
```

Server Configurations

Finally, if no other configurations are provided either within the `ssh` command's arguments or within the relevant `~/.ssh/config` file, SSH searches for any server configurations at `/etc/ssh/ssh_config`.

Pluggable Authentication Modules

The PAM configuration files are handled sequentially at the time of authentication. This means that the order in which these settings are place is crucial to how they are interpreted by PAM. Be careful to understand what each line does and where is should be placed, or you could end up being locked out from your server due to a configuration error.

Default SSHD PAM

Upon starting an Ubuntu 19.04 server, PAM comes configured for basic password authentication for SSH by using `/etc/pam.d/common-auth` within the `/etc/pam.d/sshd` configuration. Notice below on line 4 of the default `/etc/pam.d/sshd` configuration file packed with Ubuntu 19.04. PAM includes the `/etc/pam.d/common-auth` file and sequentially runs through the steps it requires.

This page will only cover to and including line 14 of /etc/pam.d/sshd - the rest of these files were provided for completeness.

```
# PAM configuration for the Secure Shell service
# Standard Un*x authentication.

@include common-auth

# Disallow non-root logins when /etc/nologin exists.
account    required     pam_nologin.so

# Uncomment and edit /etc/security/access.conf if you need to set complex
# access limits that are hard to express in sshd_config.
# account  required     pam_access.so

# Standard Un*x authorization.
@include common-account

# SELinux needs to be the first session rule.  This ensures that any
# lingering context has been cleared.  Without this it is possible that a
# module could execute code in the wrong domain.
session [success=ok ignore=ignore module_unknown=ignore default=bad]        pam_selinux.so
close

# Set the loginuid process attribute.
session    required     pam_loginuid.so

# Create a new session keyring.
session    optional     pam_keyinit.so force revoke

# Standard Un*x session setup and teardown.
@include common-session

# Print the message of the day upon successful login.
# This includes a dynamically generated part from /run/motd.dynamic
# and a static (admin-editable) part from /etc/motd.
session    optional     pam_motd.so  motd=/run/motd.dynamic
session    optional     pam_motd.so noupdate

# Print the status of the user's mailbox upon successful login.
session    optional     pam_mail.so standard noenv # [1]

# Set up user limits from /etc/security/limits.conf.
```

```
session    required     pam_limits.so

# Read environment variables from /etc/environment and
# /etc/security/pam_env.conf.
session    required     pam_env.so # [1]
# In Debian 4.0 (etch), locale-related environment variables were moved to
# /etc/default/locale, so read that as well.
session    required     pam_env.so user_readenv=1 envfile=/etc/default/locale

# SELinux needs to intervene at login time to ensure that the process starts
# in the proper default security context.  Only sessions which are intended
# to run in the user's context should be run after this.
session [success=ok ignore=ignore module_unknown=ignore default=bad]        pam_selinux.so
open

# Standard Un*x password updating.
```

Now, lets take a look at `/etc/pam.d/common-auth` -

```
#
# /etc/pam.d/common-auth - authentication settings common to all services
#
# This file is included from other service-specific PAM config files,
# and should contain a list of the authentication modules that define
# the central authentication scheme for use on the system
# (e.g., /etc/shadow, LDAP, Kerberos, etc.).  The default is to use the
# traditional Unix authentication mechanisms.
#
# As of pam 1.0.1-6, this file is managed by pam-auth-update by default.
# To take advantage of this, it is recommended that you configure any
# local modules either before or after the default block, and use
# pam-auth-update to manage selection of other modules.  See
# pam-auth-update(8) for details.

# here are the per-package modules (the "Primary" block)
auth    [success=1 default=ignore]      pam_unix.so nullok_secure
# here's the fallback if no module succeeds
auth    requisite                       pam_deny.so
# prime the stack with a positive return value if there isn't one already;
# this avoids us returning an error just because nothing sets a success code
```

```
# since the modules above will each just jump around
auth    required                        pam_permit.so
# and here are more per-package modules (the "Additional" block)
auth    optional                        pam_cap.so
# end of pam-auth-update config
```

See on line 17 above, where we define an authentication method, what should be done on success, and what should be done otherwise (The default is a failed attempt, since we assume the user is not who they say they are.) Upon successful authentication, we set the step=1, which simply tells PAM to skip one step in our authentication process. So, instead of the default path sending the user to line 19 where they are denied authentication, we move to the next valid line (23) and permit the attempt.

The example above is the basics of how configuring PAM will be handled. It can be tricky at first, but just mess with things a bit and you will have the hang of it in no time.

Custom SSHD Authentication

Surely, if we mean to secure our server we will need to define a more specific way to authenticate. Below is an example configuration of a custom `/etc/pam.d/sshd` that allows us to do a few things when a user attempts to login -

1. Prompt for a YubiKey
2. Prompt for Local Password
3. Check for /etc/nologin file

```
# PAM configuration for the Secure Shell service



# Prompt for YubiKey first, to gate off all other auth methods
auth required pam_yubico.so id=<IDVALUE> id key=<KEYVALUE> key
authfile=/etc/ssh/authorized_yubikeys

# Prompt for the local password associated with user attempting login
# nullok allows for empty passwords, though it is not recommended.
auth required pam_unix.so nullok

# If /etc/nologin exists, do not allow users to login
# Outputs content of /etc/nologin and denies auth attempt
auth required pam_nologin.so



# We comment this out, because we already handled pam_unix.so authentication above
```

```
# Standard Un*x authentication.
#@include common-auth

...
Excess config clipped off
All below lines remain the same as their corresponding in the default /etc/pam.d/sshd
...
```

This gives us a little more security, and a lot more control over who can access our server when we are doing impacting things that require data to remain untouched. This is a very touch configuration file so there are a few things to note, I'll go over how I implemented each step of authentication and then how to modify the default PAM SSHD settings to handle these changes appropriately.

Prompting For YubiKey

By prompting for a key first, we gate all other methods behind a hard to fake form of authentication utilizing Yubico's OTP API within their Yubicloud service. It is possible to host your own validation services, but for me I would rather leave that kind of security responsibility in the hands of the much more capable and prepared hands of Yubico. See the Page on Configuring YubiKey SSH Authentication for a complete guide on how to setup your key and a more in-depth explanation of the required Yubico PAM and SSHD configuration steps. Upon purchase of a key, we will need to register it with the Yubicloud and gather an ID and KEY. We pass this into a custom PAM within our `/etc/pam.d/sshd` configuration file, and this enables Yubico to generate OTPs for secure authentication.

```
# PAM configuration for the Secure Shell service



# Prompt for YubiKey first, to gate off all other auth methods
auth required pam_yubico.so id=<IDVALUE> id key=<KEYVALUE> key
authfile=/etc/ssh/authorized_yubikeys


...
```

On line 5 above, we create an API request upon authentication using our information from Yubico, and check that the user attempting to login exists within the Authorized Yubikeys File, and that the correct 12-character public key is associated with their account.

If you do not create an Authorized Yubikey file, you will not be able to authenticate. SSH login will fail with errors that don't correspond with the issue - (ex.. Failue - Keyboard-interactive) If you are having issues, be sure that the file exists in the correct place as indicated within `/etc/pam.d/sshd`, and ensure the keys / users are correct as well.

Prompting For Local Password

We then prompt for a password, which provides protection in the event the key falls into the wrong hands. This way, we won't need to be scrambling our passwords every other week since they are gated behind another form of secure authentication.

It would be possible to setup a configuration capable of removing a compromised public key from all associated user accounts.

For example, should a public key be seen providing the incorrect password post-Yubikey authentication, we can assume either the key has been stolen, or the user has forgotten their password and will need to reset it. Send an email to the user notifying them of this activity, give them a chance to reset their password, and upon no response or verification of a stolen key kick off a script to remove the key from all accounts.

```
...

# Prompt for the local password associated with user attempting login
# nullok allows for empty passwords, though it is not recommended.
auth required pam_unix.so nullok

...
```

Above, we simply request basic pam_unix.so (PAM Unix Sign On module Authentication) with the argument `nullok`, which enables empty passwords. This is handled as expected, and just asks us for a password upon authentication, the password being set within the host - see Changing a User's Password for more information.

Nologin Check

The nologin check allows us to have full control over a system should we want to seal it off from any logins, even if they are permitted to be on the host normally. The `/etc/nologin` file simply needs to exist, and PAM will fail any authentication attempt and output the contents of the nologin file. This allows us to create a message indicating why there is no logins permitted and who to contact should there be an issue. This is a useful feature when attempting to protect data consistency in environments where many people are accessing the same servers. Below, we configure the pam_nologin.so module to handle this step in authenticating

```
...

# If /etc/nologin exists, do not allow any user to login
# Outputs content of /etc/nologin and denies auth attempt
```

```
auth required pam_nologin.so


...
```

Revision #2
Created 2019-07-06 11:15:17 UTC by Shaun Reed
Updated 2020-06-04 15:57:07 UTC by Shaun Reed