SSH Configuration

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


...
```

Enabling Google 2FA

Overview

Two factor authentication is easy to configure and helps further secure your server. It requires a few extra packages -

```
sudo apt update && sude apt upgrade
sudo apt install libpam-google-authenticator
```

Along with these packages, we will need to make some changes to our `/etc/pam.d/sshd` and `/etc/ssh/sshd_config` files. See below for the complete steps.

Setup Google Authenticator

Run `google-authenticator` and respond to the prompts appropriately. Below is an example of the prompts output along with my responses - you can change or modify your responses to these prompts as you see fit. I did exclude some information for security reasons, but nothing more than the output between the prompts setting things up specific to my system.

```
Do you want authentication tokens to be time-based (y/n) y

...

Do you want me to update your "/home/host/.google_authenticator" file (y/n) y

Do you want to disallow multiple uses of the same authentication
token? This restricts you to one login about every 30s, but it increases
your chances to notice or even prevent man-in-the-middle attacks (y/n) y

By default, tokens are good for 30 seconds and in order to compensate for
possible time-skew between the client and the server, we allow an extra
token before and after the current time. If you experience problems with poor
time synchronization, you can increase the window from its default
size of 1:30min to about 4min. Do you want to do so (y/n) n

If the computer that you are logging into isn't hardened against brute-force
login attempts, you can enable rate-limiting for the authentication module.
By default, this limits attackers to no more than 3 login attempts every 30s
Do you want to enable rate-limiting (y/n) y
```

The `...` within the code block above is a placeholder for information similar to the below -

```
Your new secret key is: XXXXXXXXXXXXXXXX
Your verification code is 123456
Your emergency scratch codes are:
  12345678
  23456789
  34567890
  45678901
  56789012
```

Yes, you *should* save those scratch codes. They act as static keys that can be used for 2FA in the event that you are unable to use the linked device for any reason. Along with this, a QR code will be output to your terminal, which can be easily scanned using the Google Authenticator application from any device. Alternatively, you could input your secret key into the application when creating a new token. This will give you an auto-regenerating token that has strong security features, such as the rate-limiting feature I enabled during the final prompt of the `google-authenticator` setup above. This will enable a timeout period between multiple failed login attempts, which makes it more difficult to brute-force.

> Pay attention to the output during the setup process, its important that this process is completed correctly. If it is not, you could face issues when attempting to SSH into your server. **If you are not careful, this could result in a lockout.**
> Always have a secondary login method, until you have verified that these settings work.

SSHD Configuration

> **SSHD** - Secure Shell Daemon
> **Daemon** - A long-running background process that answers requests for services.

In the final steps of the 2FA configuration process, we need to tell SSHD and PAM that we want to use 2FA during the login process. To start, SSHD needs to know that we wish to use custom authentication methods when a connection attempt is made. Add the following to `/etc/sshd/sshd_config`, and keep in mind that comments prefixed with an asterix(*) are custom comments that I've added to explain what we are doing when we change these settings. The rest of the comments found in these files are there by default, and will be included with any installation of SSHD or PAM.

```
# *Default value is 22, change this to whatever you wish and adjust firewall / iptables
accordingly
# *This is not required to be changed for 2FA, but it is recommended for all public-facing
SSHD configurations
```

```
# What ports, IPs and protocols we listen for
Port 1234

# *You should be using keys to authenticate / provision logins
PubkeyAuthentication yes

# *Since we use the above, you should not need to use passwords when logging in
# *If desired, this can still be enabled as an extra-layer
# *Password-only auth is easy to brute-force if not secured well
# Change to no to disable tunnelled clear text passwords
PasswordAuthentication no

# *Required for SSHD to allow the response to Verification code prompt on login attempt
# *This allows you to input and pass your 2FA code to the SSHD when logging in
# Change to yes to enable challenge-response passwords (beware issues with
# some PAM modules and threads)
ChallengeResponseAuthentication yes

# *Since we will be configuring PAM, make sure it is enabled
# Set this to 'yes' to enable PAM authentication, account processing,
# and session processing. If this is enabled, PAM authentication will
# be allowed through the ChallengeResponseAuthentication and
# PasswordAuthentication.  Depending on your PAM configuration,
# PAM authentication via ChallengeResponseAuthentication may bypass
# the setting of "PermitRootLogin yes
# If you just want the PAM account and session checks to run without
# PAM authentication, then enable this but set PasswordAuthentication
# and ChallengeResponseAuthentication to 'no'.
UsePAM yes

# *Specify to SSHD what methods you want to use when a connection attempt is made
# *If this is not configured correctly, our PAM configuration could be ignored.
AuthenticationMethods publickey,keyboard-interactive
```

PAM Configuration

> **PAM** - Pluggable Authentication Modules

PAM allows us to add or configure our custom modules used during the authentication of various systems. For our needs, we will only be adding one line to the `/etc/pam.d/sshd file`. See the below

for an example configuration, our change can be found within the first few lines prefixed by a custom comment that I've added.

```
# PAM configuration for the Secure Shell service

# Standard Un*x authentication.

# *Comment this line out to stop PAM from prompting for password on a connection attempt to
SSHD
# *This should be configured according to your AllowPasswordAuthentication setting within
/etc/ssh/sshd_config
#@include common-auth

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

# *Add this line to require authetication via the google-authenticator module for PAM
auth    required        pam_google_authenticator.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.
@include common-password
```

The majority of the above file should be left alone, and **it is a sequential configuration** - this means that the order in which these settings are defined is important to how they are interpreted by PAM. If you wish to rearrange things you can, but be sure that you know what you are doing. The above configuration is verified working on several servers, in my experience at the time of this writing.

Be sure when you are changing these settings, you are running `sudo systemctl restart sshd.service ssh.service` to apply your changes, then try to login *from a new session*. There is no need to terminate your active session, or reload it. If you disconnect your session and you are unable to authenticate due to your changed settings, you could be in for a bad time.

Notes

>

> The `/etc/pam.d/common-auth` file does not need to be changed, but it is an interesting file to read if you have the time. I'll throw a snippet below since it is so short, but this file basically defines the authentication process used in `common-auth`, seen in the above `/etc/pam.d/sshd` configuration where we commented out the `@include common-auth` line.

Basically, this file defines how authentication is handled, and if you read below you can see that the `common-auth` module defaults to `pam_deny.so`, where the connection attempt is blocked by PAM. On a success, PAM simply sets `success=1`, which sequentially skips the `pam_deny.so` step and moves on to `pam_permit.so`, allowing the connection to take place.

```
# /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
```

User Administration

Managing passwords

Change current user password, prompt for current passwd - `passwd`
If you can sudo, run `sudo passwd <user>` to change a user password without prompt for current password, and with no security restrictions (min length, difficulty, etc)

Removing users

To remove a user, run `sudo userdel username`. To remove a user *and* their files within their `/home/username/` directory, run `sudo userdel -r username`

Adding users

> For a useful script to speed up this process when adding multiple users, skip to the end of this guide.

Run the following commands to create a new user on Linux -

These commands assume you are root, on a new host, so you do not need to prefix them with `sudo`, if you are not root you will need to run `sudo adduser <username>`, etc.

```
adduser username
Adding user `username' ...
Adding new group `username' (1000) ...
Adding new user `username' (1000) with group `username' ...
Creating home directory `/home/username' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for username
Enter the new value, or press ENTER for the default
        Full Name []: # You can leave all of this blank, or not
        Room Number []:# Your choice, really
        Work Phone []:
        Home Phone []:
        Other []:
```

```
Is the information correct? [Y/n] y
```

Configuring Sudo

Now, we need to configure the user for sudo access, so we set our preferred text editor and use `sudo -E` to preserve our user's environment settings while running commands as sudo.

```
# Set vim as our preferred editor
export EDITOR=/bin/vim && export VISUAL=/bin/vim
# Edit the sudoers file, preserving our current user's environment settings
sudo -E visudo
```

Find the section within `/etc/sudoers` called user privilege specification

```
# User privilege specification
root ALL=(ALL:ALL) ALL
```

Modify the file by adding the user to the section as it appears below, granting all permissions -

```
# User privilege specification
root ALL=(ALL:ALL) ALL
username ALL=(ALL:ALL) ALL
```

It's considered better practice to override the `/etc/sudoers` file by running `sudo visudo -f /etc/sudoers.d/mySudoers` - This command will allow us to store our changes in a file independent from the default sudoer configuration, and also complies with the idea that `/etc/sudoer` is a *sequential configuration,* which means the order in which settings are applied is crucial to how they are interpreted by our system.

If you feel your sudoer settings are being ignored, consider moving their location in `/etc/sudoers` to the end of the file, or use the command above to create a separate configuration, securing the default settings in the even that a mistake is made, we will still be able to authenticate using sudo. Save `/etc/sudoers` and quit, but note that you will need to logout and login again for your changes to take effect.

If you configured sudo access for your user, make sure you follow the next section to ensure they are added to the relevant `sudo` group

Configure Group Access

> Looking to check current group members? `sudo groupmems -l -g groupname`

>

> Want to add a single user to a single group? `sudo usermod -aG groupname username` will `-a` append the user to the given group. The `-G` option alone will remove the user from all groups other than the one provided.

Run `vigr` in the terminal and add your new username created to the sudo group, and any other groups you may want. This is the same thing as modifying the configuration file `/etc/group` with your preferred editor and saving it. (Docker is a common group that users will need added to - Don't run your containers as root by running `sudo docker`)

```
...
tape:x:26:
sudo:x:27:USERNAME,USERNAME2,USER3
audio:x:29:
docker:x:30:USERNAME,USER3
...
```

When saving `/etc/group`, you'll get some output warning you about consistency between a shadow configuration file. Go ahead and edit it to mirror your changes, and ignore the final warning about the `/etc/group` file consistency since we just came from modifying that file.

```
vigr
You have modified /etc/group.
You may need to modify /etc/gshadow for consistency.
Please use the command 'vigr -s' to do so.

vigr -s
You have modified /etc/gshadow.
You may need to modify /etc/group for consistency.
Please use the command 'vigr' to do so.
```

Securing User / Group IDs

You should change your user and group IDs from the default sequential values we can assume Linux has distributed for us. To do this, choose and valid ID and edit the following commands to suit your needs -

```
# Change user and group IDs
sudo usermod -u 1234 user
sudo groupmod -g 4321 usergroup

# Make sure you edit all the old permissions to reflect the above changes
# Use the old user and group IDs here
```

```
sudo find / -group 1000 -exec chgrp -h username {} \;
sudo find / -user 1000 -exec chown -h username {} \;
```

Not sure what UID and GID to choose? See the table below and choose a value that suits your needs - probably a value within an unused range. **UID and GID do not need to be the same** - This is only the case by default when adding a user via Linux Distributions such as Ubuntu, which is the one referenced / used in this guide. Feel free to specify unique values, and research more into sharing user groups for permissions in scenarios such as granting a list of employees or developers similar access.

> | UID/GID | Purpose | Defined By | Listed in |
> |---|---|---|---|
> | 0 | `root` user | Linux | `/etc/passwd` + `nss-systemd` |
> | 1 ... 4 | System users | Distributions | `/etc/passwd` |
> | 5 | `tty` group | `systemd` | `/etc/passwd` |
> | 6 ... 999 | System users | Distributions | `/etc/passwd` |
> | 1000 ... 60000 | Regular users | Distributions | `/etc/passwd` + LDAP/NIS/... |
> | 60001 ... 61183 | Unused | | |
> | 61184 ... 65519 | Dynamic service users | `systemd` | `nss-systemd` |
> | 65520 ... 65533 | Unused | | |
> | 65534 | `nobody` user | Linux | `/etc/passwd` + `nss-systemd` |
> | 65535 | 16bit `(uid_t) -1` | Linux | |
> | 65536 ... 524287 | Unused | | |
> | 524288 ... 1879048191 | Container UID ranges | `systemd` | `nss-mymachines` |
> | 1879048191 ... 2147483647 | Unused | | |
> | 2147483648 ... 4294967294 | HIC SVNT LEONES | | |
> | 4294967295 | 32bit `(uid_t) -1` | Linux | |

> Table Source - Systemd.io

You should validate all the configuration done to secure your server - for example, this could be validated by running the following commands to check UID / GID after setting them and logging into our user.

Check UID / GID

id -u username

id -g username

If you plan to stop here, be sure to login to your new user before making further changes to your system.

```
sudo su username
# Or
sudo -iu username
```

Bash Add User Script

Using the information on this page, we can create a simple bash script to handle this process for us. If you plan to add a fair amount of users to a system, automating at least the general portion of that process might be valuable to you. See the script below to automate up to this point in these instructions. Simply save it into `addusers.sh` for example, and run `sudo chmod a+x addusers.sh` followed by `sudo ./addusers.sh username 1005` where 1005 is the userID you wish to assign to your new user. Sudo is required here if you wish to assign sudo privileges to the new user.

> Want to call this from the commandline as any other command? Assuming you have the script marked as an executable placed within your `/opt/` directory, run `echo "export PATH=$PATH:/opt/" >> ~/.bash_aliases && source ~/.bashrc` You should now be able to run the script by its current name from any directory on the system - `adduser.sh` Feel free to rename it

```
#!/bin/bash
## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
## A custom bash script for creating new linux users.                        ##
## Syntax: ./adduser.sh <username> <userID>                                  ##
################################################################################

if [ "$#" -ne 2 ]; then
  printf "Illegal number of parameters."
  printf "\nUsage: sudo ./adduser.sh <username> <groupid>"
  printf "\n\nAvailable groupd IDs:"
```

```
  printf "\n60001.....61183 Unused | 65520..............65533  Unused"
  printf "\n65536.....524287 Unused | 1879048191.....2147483647  Unused\n"
  exit
fi

sudo adduser $1 --gecos "First Last,RoomNumber,WorkPhone,HomePhone" --disabled-password --uid
$2

printf "\nEnter 1 if $1 should have sudo privileges. Any other value will continue and make no
changes\n"
read choice
if [ $choice -eq 1 ] ; then
printf "\nConfiguring sudo for $1...\n"
sudo usermod -G sudo $1
fi

printf "\nEnter 1 to set a password for $1, any other value will exit with no password set\n"
read choice

if [ $choice -eq 1 ] ; then
printf "\nChanging password for $1...\n"
sudo passwd $1
fi
```

> The script pasted above is not updated frequently, and only exists here so the code remains relevant to the information on this page. This script can be found at gitlab/shaunrd0/klips, but the version there may have changed slightly since writing the content on this page.

Now after creating this user and following the prompts in the script above, all you'll need to do is configure the user-specific settings you wish to apply in your case.

Creating SSH Keys

The steps in the section below are for generating a SSH key for the remote user you want to use to login to your server. After completing these steps, the next section will cover adding the public key we generate to the server's `authorize_keys` file, and logging into the box remotely.

To make things clear, I will refer to the machines we configure as `A` and `B`. The goal is to provide the necessary configurations on both `A` and `B` so that a user on `A` can use SSH to login to machine `B`. Presumably, machine `B` could be a VPS hosted by DigitalOcean or some other provider, and machine `A` could be your personal laptop that you plan to use to admin this server.

Remote User Configuration

SSH should never be authenticated using passwords alone, using public keys generated by `ssh-keygen` we can authenticate based on a key we generate and distribute manually to the remote server configuratioon files, allowing our user to login to the box. This should be done with care, as a combination of sloppy `authorized_keys` files and lost or stolen keys can lead to a compromised web server!

To generate an `ed25519` key for our new user, first we should navigate to their `~/.ssh/` directory - **on machine** `A`.

```
sudo su username
cd ~/.ssh/
ssh-keygen -t ed25519
```

If you run the last above command as sudo, it will create a key for `root@host`, not the user you are logged in as.
If you are getting privelege errors, you are not in your home directory. If the `~/.ssh` directory does not exist, create it and navigate within the new directory before running the `ssh-keygen` command.

You will be asked to answer a series of questions about the key you want to generate. The general format for filename is `user_<keytype>` so if our user is called `username` the file could be named `username_ed25519`. Once answering the questions this will create a public and private key and output them into your current directory (`/home/username/.ssh`), you should keep your private key safe and never share it with anyone. Your public key is what we give to the remote server so they can verify our identity when logging in.

Once the files are generated, ensure permissions are set approprately for `.ssh/` and `authorized_keys` file (if it exists)
`sudo chmod -R 700 ~/.ssh && sudo chmod 600 ~/.ssh/authorized_keys`

Login Server Configuration

Now, **on machine** `B`, create a new user following the steps in the sections above, or feel free to use the `adduser.sh` script to handle this in one step. Login to this user, just as we did on machine `A`, and navigate to their `~/.ssh` directory. Again, is this `~/.ssh` directory does not exist, just create it and then navigate within.

```
./adduser otherusername 2000
sudo su otherusername
cd ~/.ssh
```

Note that the name of the user on machine `B` does not need to match the name of the user on machine `A`, since we can specify a username with `ssh otherusername@0.0.0.0`.

Now that we have the user created on machine `B`, create an `/home/otherusername/.ssh/authorized_keys` text file and open it for editing. Paste in the public key we generated on machine `A` found at `/home/username/.ssh/username_ed25519.pub`. This `authorized_keys` file is what will be checked for approved keys when logging into machine `B` with a certain username. If the user requesting to login uses any key within it's `/home/otherusername/.ssh/authorized_keys` file, login access is granted.

Once the files are generated, ensure permissions are set approprately for `.ssh/` and `authorized_keys` file
`sudo chmod -R 700 ~/.ssh && sudo chmod 600 ~/.ssh/authorized_keys`

Using Putty with OpenSSH Keys

> This section is outdated, as I no longer use Putty for SSH on Windows. When working on Windows, I tend to run a Linux VM on a seperate monitor, and I just use the VM to ssh around to boxes I own. I just find this to be easier for me personally. As an alternative, you could probably just download and use the Ubuntu application on the microsoft store, and configure SSH as you would on Linux. This would save system resources required to run the VM, if all you need is a terminal.

At some point when a password is used in key generation, ssh-keygen generates openssh private key which doesn't use cipher supported by puttygen.

ssh-keygen doesn't provide option to specify cipher name to encrypt the resulting openssh private key.

There is a workaround: remove the passphrase from the key before importing into puttygen.

Create a copy of the key to temporarily remove the password
`cp ~/.ssh/id_ed25519 ~/.ssh/id_ed25519-for-putty`

import the copied key, using the -p argument to specify a request to set a new password, and -f to specify the import keyfile.

```
ssh-keygen -p -f ~/.ssh/id_ed25519-for-putty
Enter old passphrase: <your passphrase>
Enter new passphrase (empty for no passphrase): <press Enter>
Enter same passphrase again: <press Enter>
```

using some command, view the text contents of the private key generated.

```
cat id_ed25519-for-putty
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZWQyNTUxOQ
```

```
AAACCGyjniPP1oVCXqkdCeCKFp+5+5cI7L79rP5RYHJ5Y6fQAAAJh3QGp1d0BqdQAAAAtzc2gtZWQy
NTUxOQAAACCGyjniPP1oVCXqkdCeCKFp+5+5cI7L79rP5RYHJ5Y6fQAAAEBJr8PzmuEN6qNyrY07Lr
LAgZRjo9efYETKqFbS2jVTQobKOeI8/WhUJeqR0J4IoWn7n7lwjsvv2s/lFgcnljp9AAAADmthcHBl
ckBrYXB1bnR1AQIDBAUGBw==
-----END OPENSSH PRIVATE KEY-----
```

copy this output from your ssh session to the machine running Putty

On the windows machine, create a .ssh directory in the users folder who wishes to SSH into the server (C:\Users\Shaun.ssh)

navigate inside the directory, and create a text file - paste the output from your private key into this file, file->saveAs In the dropdown 'save as file type', select 'All Files', be sure to end the keyfile name with the .key extension -> username_ed25519.key click save.

Open puttygen, load convert->import keys.. select the text file we created in C:\Users\Shaun.ssh\ and set the passphrase from puttygen.

Don't forget to shred and remove ~/.ssh_id_ed25519-for-putty afterwards since it is not password protected.

The new password protected key will authorize the user based on the local password set in putty, using the remote PUBLIC key stored on the server.

Yubikey SSH Authentication

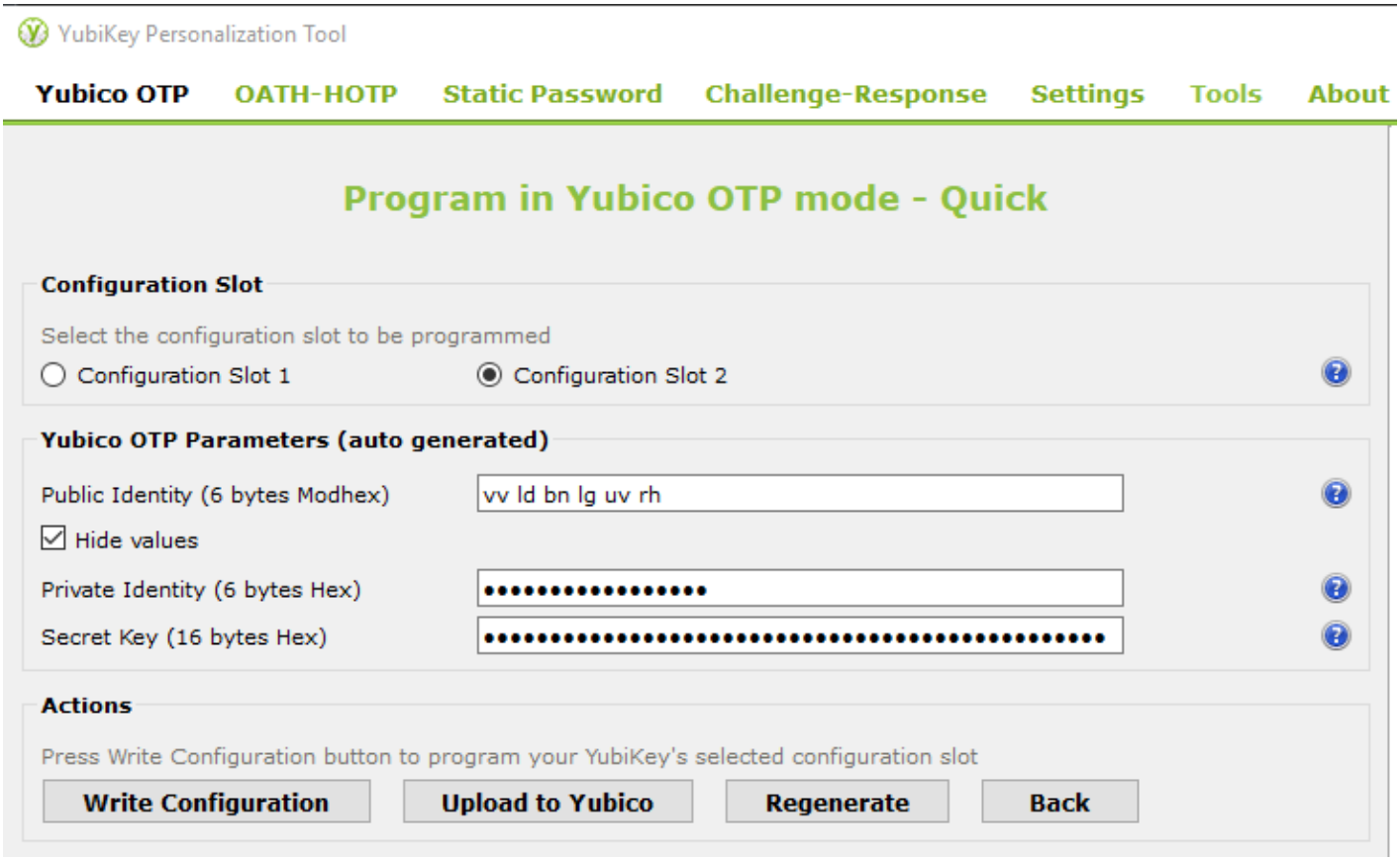
Overview

Yubikeys provide many different forms of secure authentication, for the sake of time this guide will only cover OTP (One Time Password) Authentication over SSH configured on an Ubuntu 19.04 box. This form of authentication allows you to consolidate all your 2FA passwords within one physical USB key. When plugged in and tapped, various configurations can be accessed and passed to be used for 2FA or primary auth, depending on your needs. In this guide, I will configure primary authentication using OTPs validated over Yubico's Authentication API via a collection of information such as time, ID, API Keys, and more. By cross referencing the information at the exact time of authentication / interaction with the associated physical Yubikey, we can login with a quick tap after some configuration on our services.

Yubikey Personalization Tool

To configure OTPs on our Yubikey, We'll need the Yubikey Personalization Tool - this will allow us to create a secondary configuration or overwrite the current running primary. Download the tool and select 'Yubico OTP' along the top bar, and click 'quick' configuration. You'll be greeted with the below window.

Writing OTP Configuration



Write the configuration, and you will be prompted if you would like to create a log file of your configuration. This file can be used to recover your key configuration should you lose it - if you choose to save a log of this configuration you should take care to store it in a very secure place, if someone were to obtain it they could use it maliciously. Save the log file, or click cancel to create no logs and store the configuration only on the Yubikey.

> It's important to note that the Yubikey configurations are **write-only**, this means that if in the future you want to obtain the configuration from the key you will need to overwrite the running config and save the log at the time of configuration. This is a security feature to prevent an on-site attacker from duplicating keys and configurations freely.
>
> The firmware on the key is also burned into the chip so no modifications can be made to the back-end of the key to alter these security settings. This means no updates can be provisioned to the key. For me, this is a fair trade for my security. Should I need a newer version key, I will simply purchase a new one.

Upload Configuration Credentials

By uploading your configuration, you provide Yubico with the information required to authenticate you key with your new configuration when an attempt is made. Click Upload configuration, and you'll be redirected to a web page that will automatically populate some fields in the screenshot below. For the sake of this guide, I have deleted the information as it should not be shared publicly. You could find and fill out this form manually, without the Personalization Tool - though it would take some more effort that I won't cover here. If you want to see the page or access it remotely, the URL is just https://upload.yubico.com/

yubico
trust the net

Enter information about your newly-personalized YubiKey.

Note: It can take up to 15 minutes for an uploaded identity to become valid on our validation servers. 'vv' prefix credentials are not guaranteed to have the same availability as production 'cc' prefix credentials. Yubico reserves the right to revoke any 'vv' prefix credential on the Yubico validation service (YubiCloud) at any time, for any reason, including if abuse is detected or if the credential is loaded onto a counterfeit YubiKey.

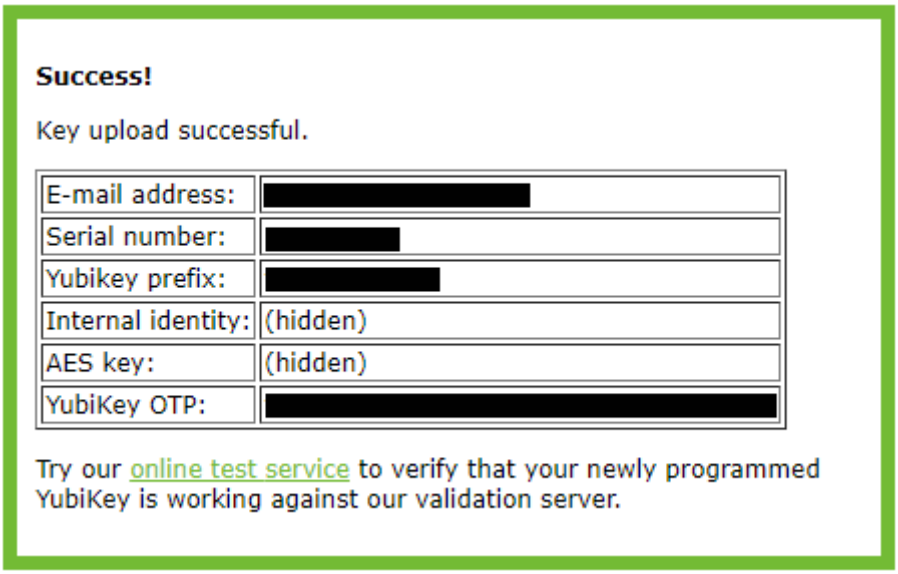
Your email address:
Serial number of the YubiKey:
Public identity:
Private identity:
Secret key:
OTP from the YubiKey:

I'm not a robot
reCAPTCHA
Privacy - Terms

Upload AES key

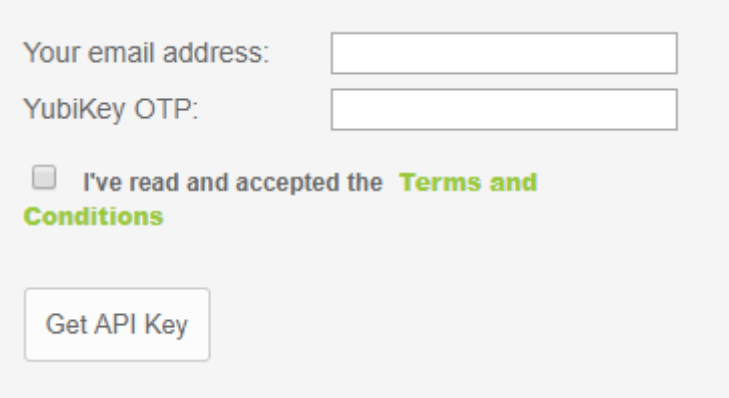
> The 'OTP from the Yubikey' should be passed into the associated field by accessing the configuration you just wrote to your key. So, for this guide, I have used the Yubikey 5 NFC - Which allows for two configurations, selected within the Personalization Tool prior to writing our configuration.
> **This is how you will authenticate when prompted by your services.**
> To use configuration 1, tap the key.
> To use configuration 2, tap and hold the key for 2-3 seconds.

After completing this form, you'll be greeted with the one below - save it if you want to be able to restore your key settings should you lose this one.

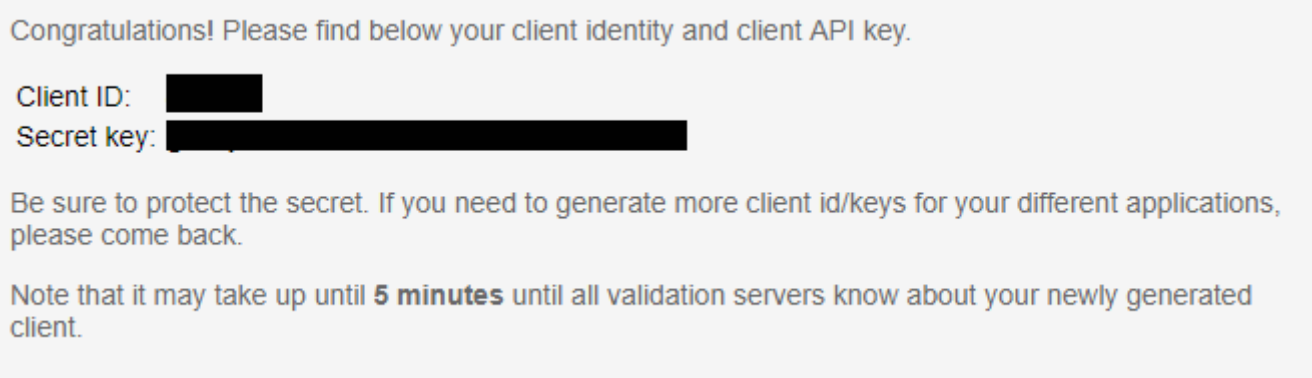


Obtain Yubico API Key

To request an API key, fill out the Get API Key form from Yubico, note that this step must be completed after writing and uploading your configuration, and will be directly associated with the OTP authentication we have configured in the steps above. The form is simple, and provides a good test of our new configuration. Basically, we authenticate with our new OTP and Yubico provides us with an associated API key to use with our configuration files in the future.



Once filled out, Yubico will present you with your new keys -



Ubuntu Server Configuration

The following steps will be performed via command-line within your Ubuntu server. Note that these steps may vary if you are not using Ubuntu, but generally they should be very similar in concept.

> **Any time you are directly modifying SSH access to a remote server**, you should be careful to validate your new settings **before** exiting the session you've configured them in. This ensures that if your settings are not correct, you will still be logged in and therefore can just continue to alter them until they suit your needs.
>
> If you exit your session to validate your settings and are unable to reconnect - you could be locked out. Don't get locked out, just start an entirely new session to test your settings.

SSHD Configuration

Some basic modifications need to be made to the `/etc/ssh/sshd_config` - see that the lines below exist in some form within your configuration. It is possible to mix-and-match these options with many other forms of authentication, should you want the user to be prompted for various things such as Google-2FA, PIN, or a basic password. By gating the second form of authentication behind the Yubikey, you remove the opportunity for brute-forcing or guessing at these PINs or passwords, so the need to update them is far less frequent, but they should still be maintained / reset occasionally.

```
#/etc/ssh/sshd_config


AuthenticationMethods keyboard-interactive
ChallengeResponseAuthentication yes
UsePAM yes
```

Now, we'll need to `vim /etc/ssh/authorized_yubikeys` to populate a list of server-level authorized keys. Note that you can create user-specific keys stored within home directories much like the default `.ssh/authorized_keys` file works.

```
#/etc/ssh/authorized_yubikeys


shaun:vrnfgfebjiji
guests:vrnfgfebjiji:hhrefkikfcgr:dllcfndknkbf
newuser:hhrefkikfcgr:vrnfgfebjiji
```

Yubico PAM Configuration

Yubico Provides a custom PAM which allows them to pass your authentication through their API when connecting to the SSHD. Run the following commands to download it for Ubuntu.

```
sudo add-apt-repository ppa:yubico/stable
sudo apt-get update
sudo apt-get install libpam-yubico
```

> You may need to move `pam_yubico.so` to wherever PAM modules are stored on your system (usually `lib/security`). The Ubuntu package will automatically install the module in the appropriate location, but you can check to see whether it's in the right location with `ls /lib/security`. It may also be stored in `/usr/local/lib/security`, in which case you will need to move it manually.

We'll need to modify the configuration for our new PAM for Yubikeys. Add `auth required pam_yubico.so id=<clientid> id key=<SecretKey> key authfile=/etc/ssh/authorized_yubikeys` to your `/etc/pam.d/sshd` configuration. Be sure to modify the `<values>` appropriately.

```
#/etc/pam.d/sshd

#Add the following line, modifying <values> appropriately.
auth required pam_yubico.so id=<clientid> id key=<SecretKey> key
authfile=/etc/ssh/authorized_yubikeys

# Standard Un*x authentication. Uncomment this to use a password as well.
# @include common-auth
```

Note that the above file is a sequential configuration and the order of the lines added to this file is critical to the way it is read by your system.

> It is possible to mix-and-match these options with many other forms of authentication, should you want the user to be prompted for various things such as Google-2FA, PIN, or a basic password. By gating other forms of authentication behind / after the Yubikey line we added above, you remove the opportunity for brute-forcing or guessing at these PINs or passwords, so the need to update them is far less frequent, but they should still be maintained / reset occasionally.

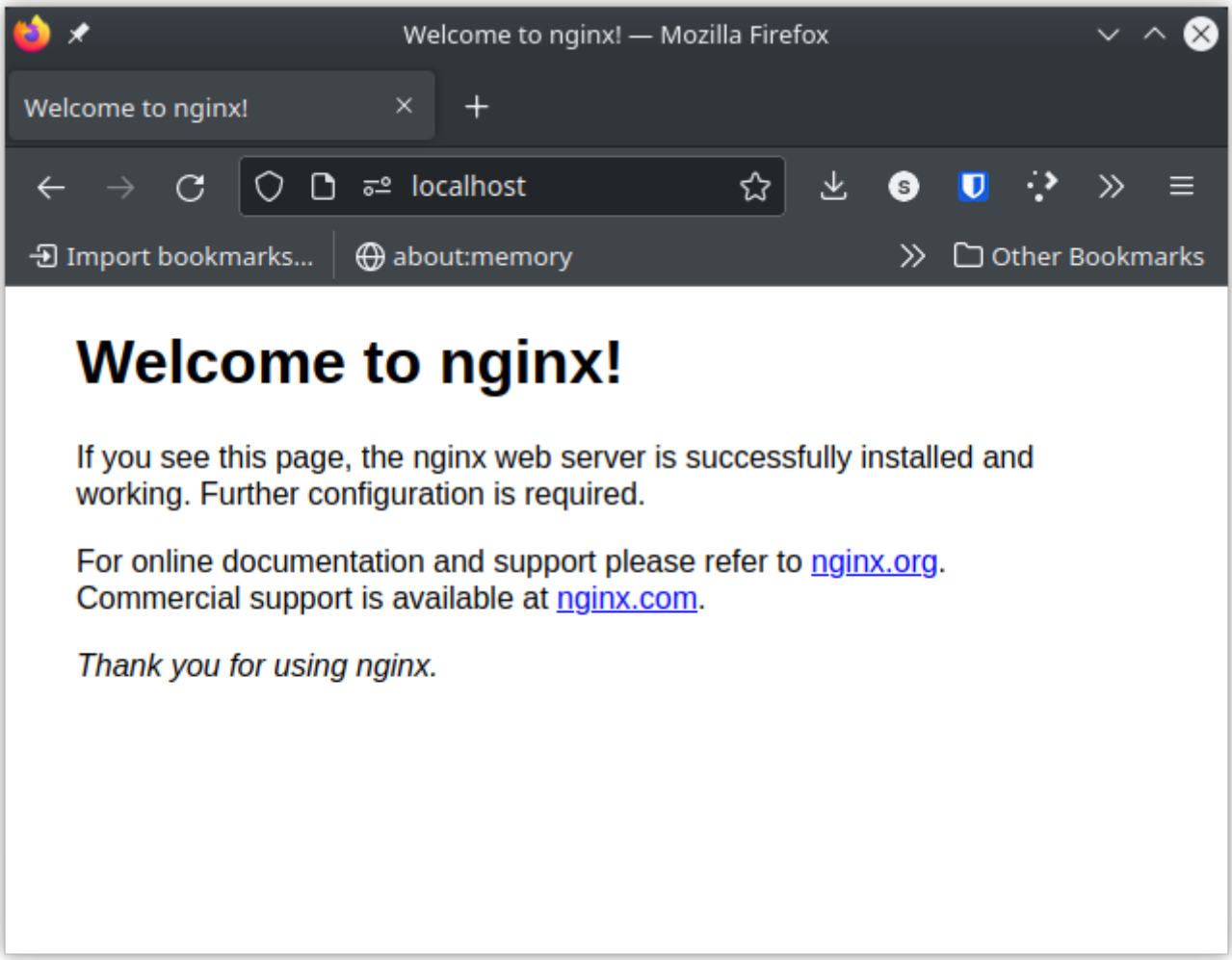
Tunneling

Reverse Tunneling

this is AKA Remote Port Forwarding, where we basically forward a remote server's port to direct requests to a local port on our machine. This is rather fun to play with, and only takes a few minutes to complete a working example if you're familiar with Linux and NGINX.

First we should keep in mind that if we want to forward any ports below `1024` on the remote server, we need to login as the root user. It doesn't matter if your user has sudo or not, it won't work unless you are root. You could maybe reconfigure things to make this not the case, but for the sake of this example we will just use the root user.

Start a local NGINX server and visit `localhost` in your web browser to see that it's working correctly. We will just use the default NGINX template.



Now login to your remote server and make sure the following line is withi `/etc/sshd/sshd_config` to allow public port forwarding.

```
# /etc/sshd/sshd_config
# By default, this is set to `no`; Make sure you change it to `yes`
# GatewayPorts no
GatewayPorts yes
```

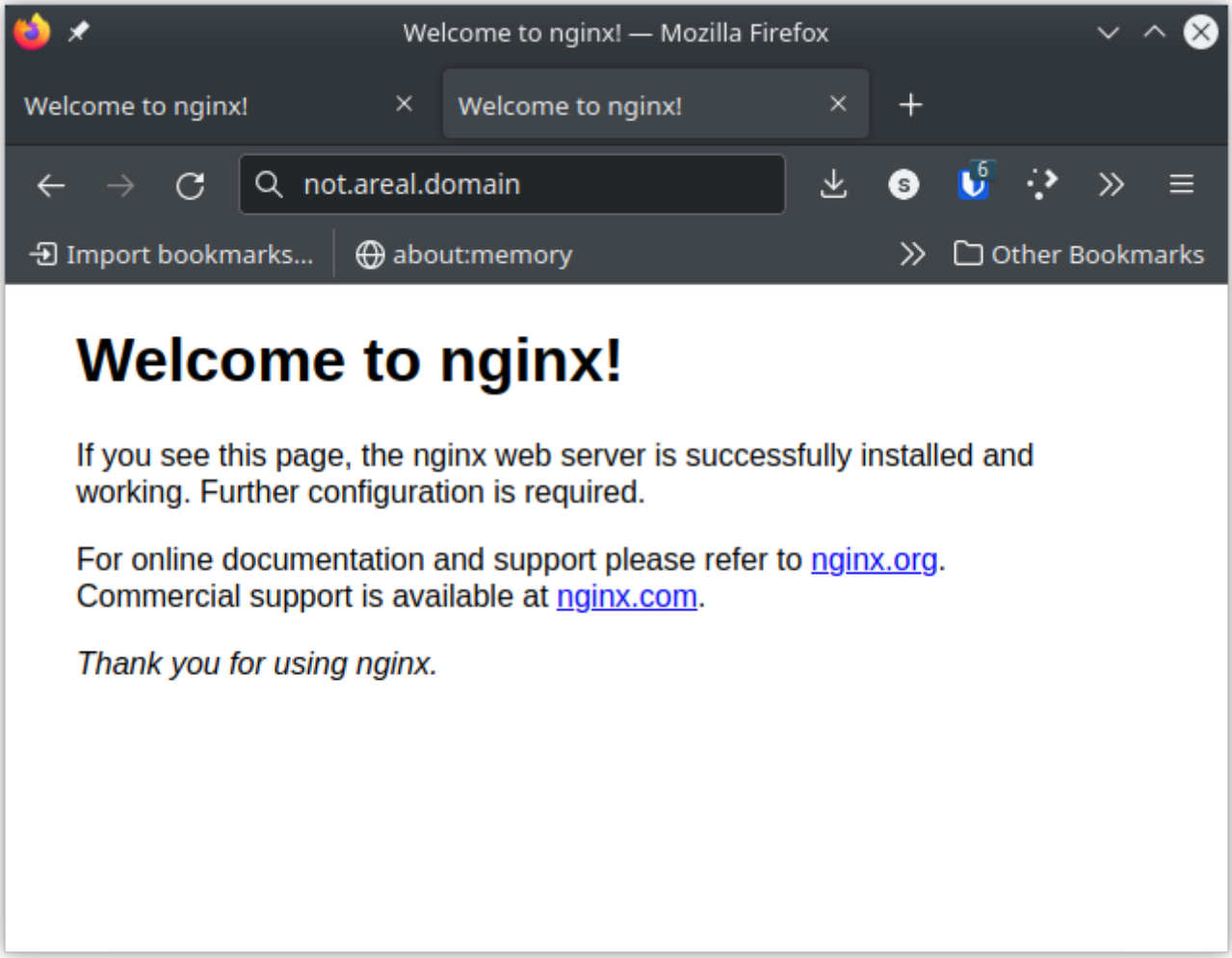
Now, restart the `sshd.service` by running the following command, and make sure to stop the `nginx.service` if it is running on your remote server. Finally we `exit` the ssh session so we can relog as `root` and start our remote SSH tunnel

```
sudo systemctl restart sshd.service
sudo systemctl stop nginx.service
exit
```

To bind the remote server with the ssh command, the syntax is `ssh -R <REMOTE_PORT>:<LOCAL_IP>:<LOCAL_PORT> root@<REMOTE_IP>`. An example of this for my server is the command below. Note the remote IP is fake, since I don't want to share this IP publicly.

```
ssh -R 80:127.0.0.1:80 root@123.456.789.123
```

That's it! Once you've connected to your ssh session, you can visit your remote server's domain name or IP and it will redirect requests to port `80` to your local webserver.



Sources

goteleport - ssh tunneling explained