Hardening Guide
Filestash

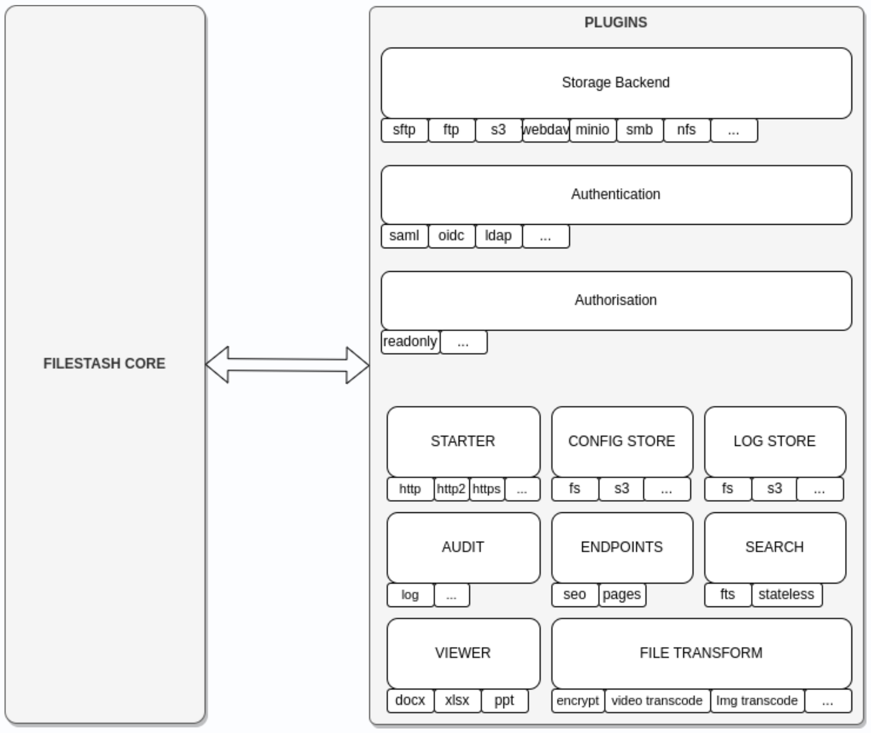
Contents

1 Introduction

Filestash is a self-contained server software that runs on Linux with an amd64 or arm architecture. It can be made available as a docker container or a static binary. This hardening guide will only cover the static build method running in complete isolation as a self-contained standalone software without any runtime dependency.

2 Architectural Overview

Filestash is made of a core around which are associated modules we refer to as "plugins".



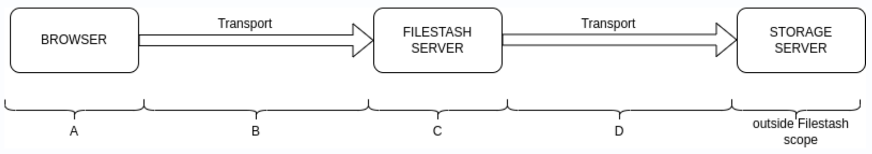
Plugins help change many aspects of Filestash. The scope of plugins encompasses many areas, the most important of which is the triptych storage/authentication/authorisation. The storage plugin implements most file transfer protocols, the authentication plugin connects to an IDP, and the authorisation layer ensure nobody can read/write onto something they aren't supposed to.

The plugins installed in your instance are set at compile time (unless you use the dynamic loader plugin `plg_dlopen` [A.1]) and are visible from the /about page. We can add and remove plugins to only use what's required by your use case, effectively limiting the capabilities of the software and reducing the attack surface. Concretely the list of plugins is located under `/server/plugin/index.go` with a minimal example looking like tihs:

```
package plugin

import (
  . "github.com/mickael-kerjean/filestash/server/common"

  // this is how the server starts, by default using HTTP on
  // port 8334 but a range of other options are available
  // we will discuss some of those options later on in this guide
  _ "github.com/mickael-kerjean/filestash/server/plugin/plg_starter_http"

  // this is the file transfer protocol our instance can use,
  // in this case sftp but a range of options are available
  _ "github.com/mickael-kerjean/filestash/server/plugin/plg_backend_sftp"
)

func init() {
  Log.Debug("Plugin loader")
}
```

3 Hardening of a typical Filestash setup

A typical Filestash implementation has different components:



Filestash is storage agnostic and doesn't have an opinion on the storage server you want to use. This part is out of scope for this document so we will only focus on the hardening of components A, B, C and D.

3.1 Component A: Browser

By default, Filestash instructs the browser to:

- not download or execute untrusted code and or third-party untrusted resources
- not make the authentication cookie available to javascript [A.3] to reduce the risk of cookie theft if an XSS was discovered
- block the application from loading in an iframe [A.4] to protect your instance against click-jacking attacks. To run from an iframe, you need to whitelist the domain with the configuration key `"features.protection.iframe"`
- disable API calls from third-party domains unless an API key is created and configured to enable access from a specific origin [**??**]
- prevent guessing the effective MIME type of a resource by examining the content of the response through the MIME sniffing mechanism [A.5]
- stop pages from loading when they detect reflected cross-site scripting attacks [A.6]

Other features relevant to the browser:

- HTTP GET calls can't alter the state of Filestash nor the underlying storage, preventing attackers from carving links that would trick users onto doing unwanted actions [A.7]
- CSRF protection

3.2 Component B: Transport from browser to Filestash

- STS can be enabled to instruct the browser to disallow HTTP access altogether [A.8]. To enable this option, `"general.force_ssl"` should be set to `true`
- Filestash can verify the origin of the request it received and deny the ones that are from unknown origins [A.9]. This option can be enabled either by setting the "general.host" config key or the `APPLICATION_URL` environment variable to your origin
- Filestash by default is deployed as a standalone HTTP server that listens on port 8334. That behaviour is wired in the `plg_starter_http` plugin. For hardening, various other `starter` plugins are available:

- `plg_starter_httpsfs`: enables HTTPS on the server using your existing SSL certificates
- `plg_starter_web`: enables HTTPS on the server using Letsencrypt to generate the SSL certificates

3.3 Component C: Filestash server

3.3.1 Application security features

The /about page provides all the information that relates to your build, including:

1. the list of plugins installed which can be OSS plugins, enterprise plugins or custom plugins [A.10]
2. the commit hash that allows you to audit the actual code which went on a build [A.11]
3. the hash of the binary to enable automated systems to detect whenever something has changed like during an upgrade or if an intrusion happens [A.12]
4. the hash of the configuration makes it possible for an automated system to detect any change [A.13]

Other features which are relevant to the server:

1. The currently deployed version is visible from the headers in the HTTP responses [A.14]
2. The healthcheck endpoint ensures the server is running correctly [A.15]
3. The presence of an endpoint instructs security researchers on how to report a security vulnerability [A.16]

Companies that need to maintain 24/7 uptime for their services can consider using high-availability clusters. Various standard architectures are possible based on load balancing in pools or at the DNS level, but the details of such architectures are outside the scope of this hardening guide.

3.3.2 Authentication and authorisation

Filestash has two types of users:

- Admin user who have the ability to see/change the configuration and perform other administrative tasks. It's possible to disable the entire admin console entirely which is normally available under `/admin` by using the `plg_admin_nil` plugin [A.17]
- Normal user. In a hardened mode, we will only keep a single authentication middleware (typically either the SAML plugin or the openID plugin) depending of what your IDP supports [A.18]

By default the endpoints enabling users to authenticate are rate limited [A.19] and it's possible to provide additional restrictions based on custom rules using custom plugins.

3.3.3 Audit logging and treat detection

Application logs are written in both a log file located in `data/state/log/access.log` and `stdout`. The structure of the logs is detailed in the documentation and looks like this:

```
2022/12/08 09:33:55 SYST INFO Filestash v0.5 starting
2022/12/08 09:33:55 SYST INFO [http] starting ...
2022/12/08 09:33:55 SYST INFO [http] listening on :8334
2022/12/08 12:20:58 HTTP 200 GET    0.8ms /files/mickael/
2022/12/08 12:20:58 HTTP 200 GET    0.2ms /custom.css
2022/12/08 12:20:58 HTTP 200 GET    4.6ms /assets/js/app_be2ce54bbd2cfc50e943.js
2022/12/08 12:20:59 HTTP 200 GET    0.2ms /assets/locales/fr.json
2022/12/08 12:20:59 HTTP 200 GET    1.6ms /api/config
2022/12/08 12:20:59 HTTP 200 GET   11.2ms /api/session
2022/12/08 12:20:59 HTTP 200 GET    0.3ms /favicon.ico
2022/12/08 12:20:59 HTTP 302 GET    0.1ms /manifest.json
2022/12/08 12:20:59 HTTP 500 GET    2.9ms /api/files/ls?path=%2Fmickael%2F
2022/12/08 12:20:59 HTTP 200 GET    0.3ms /assets/logo/android-chrome-192x192.png
```

If you use a logging service like Splunk, log ingestion can be done directly through the API of your vendor via a plugin. Auditing is also the responsibility of a plugin. By default `plg_audit_log` will log the actions made by users in plain text and provide a graphical way to query that audit data from the admin interface [A.20].

There are a couple more standard plugins that help in threat detection and remediation:

- `plg_security_scanner`: this plugin contains heuristics to detect the use of a scanner [A.21]
- `plg_security_killswitch`: this plugin makes it possible to remotely stop an instance if we were to discover a vulnerability in the like of log4j in the future [A.22]

3.3.4 Configuration management

By default, the configuration data is stored on the filesystem. Various other plugins can override this default:

- `plg_config_env`: stores the config in the `CONFIG_JSON` environment variable as a base64 string
- `plg_config_s3`: stores and retrieves the config from an s3 bucket
- `plg_config_vault`: stores the config in an hashicorp vault

3.4 Component D: Transport from Filestash to the storage component

The security of the transport layer from the Filestash server to the storage component depends heavily on which file transfer protocols you want to use. As such, the hardening of this component can only be done on a protocol-per-protocol basis:

- SFTP: the base `plg_backend_sftp` plugin allows for an empty host key which does shortcut the host verification based on the public key fingerprint of the SSH server. The hardened version of the SFTP backend plugin `plg_backend_sftp_hardened` makes the host key verification mandatory. `plg_backend_sftp_hardened` supports 2 fingerprinting mechanisms based on md5 and sha256, and the authentication can be done either via a password or using a private key. To provide additional restrictions, a custom plugin is required
- FTP: the base `plg_backend_ftp` supports both FTP and FTPS connection in implicit and explicit mode. The hardened configuration can be done to block FTP altogether and only enabled FTPS in explicit mode (`plg_backend_ftps_explicit`) or in implicit mode (`plg_backend_ftps_implicit`) depending on what is supported by your server

- `plg_backend_webdav_hardened`: same as `plg_backend_webdav` except it won't establish a connection over HTTP but only HTTPS. It's possible to create additional customisations to either restrict the domain and other TLS configurations to provide a hardening based on your specific use case
- `plg_backend_s3_hardened`: same as `plg_backend_s3` but enforces usage of an encryption key in which case Filestash will be relying on the AWS SDK to encrypt the incoming/outgoing data using AES256 (refer to SSECustomerKey in the AWS SDK documentation). More variations of this plugin can be offered to restrict the capabilities of the s3 backend plugin (eg: restrict region, role, ...)
- `plg_backend_git_hardened`: same as `plg_backend_git` but removes support for git over HTTP and only allows git over ssh using a private key and a valid host key

In case your company is using their own root certificate and requires changes in how TLS is handled, we can setup a plugin that implements those particular rules.

A Appendix

A.1

```
curl -X GET -s "https://demo.filestash.app/about" | \
  grep "plg_dlopen"
```

A.2

```
curl -X GET -o /dev/null -s -D - "https://demo.filestash.app/" | \
  grep "Content-Security-Policy: "
```

A.3

```
curl -sD - "https://demo.filestash.app/api/session" \
  --header "X-Requested-With: XmlHttpRequest" \
  --data '{"type":"webdav","url":"https://webdav.filestash.app/"}' | \
  grep -e "Set-Cookie" -e "HttpOnly"
```

A.4

```
curl -X GET -o /dev/null -s -D - "https://demo.filestash.app/" | \
  grep -e "Content-Security-Policy: " -e "frame-src 'self'"
```

A.5

```
curl -X GET -o /dev/null -s -D - "https://demo.filestash.app/" | \
  grep "X-Content-Type-Options: nosniff"
```

A.6

```
curl -X GET -o /dev/null -s -D - "https://demo.filestash.app/" | \
  grep "X-Xss-Protection: 1;"
```

A.7

see server/main.go

A.8

```
# precondition: "general.force_ssl" is set to "true"
curl -o /dev/null -s -D - "http://127.0.0.1:8334/" | \
  grep "Strict-Transport-Security: "
```

A.9

```
# precondition: "general.host" is set to "localhost:8334"
curl -s 'http://127.0.0.1:8334/api/session' \
  -H 'Content-Type: application/json' -H 'X-Requested-With: XmlHttpRequest' \
  --data-raw '{"type":"blackhole"}' | \
  grep '{"status":"error"' # wrong origin yield errors

curl -s 'http://localhost:8334/api/session' \
  -H 'Content-Type: application/json' -H 'X-Requested-With: XmlHttpRequest' \
  --data-raw '{"type":"blackhole"}' | \
  grep '{"status":"ok"' # correct origin does passthrough
```

A.10

```
curl -s "https://demo.filestash.app/about" | \
  grep -e "STANDARD" -e "EXTENDED" -e "CUSTOM" | \
  sed 's|<[^>]*>||g' | sed 's|^[[:space:]]*||g'
```

A.11

```
curl -s "https://demo.filestash.app/about" | \
  grep "Commit hash" | \
  sed 's|<[^>]*>| |g' | sed 's|^\t\s*||g'
```

A.12

```
curl -s "https://demo.filestash.app/about" | \
  grep "Binary hash" | \
  sed 's|<[^>]*>| |g' | sed 's|^\t\s*||g'
```

A.13

```
curl -s "https://demo.filestash.app/about" | \
  grep "Config hash" | \
  sed 's|<[^>]*>| |g' | sed 's|^\t\s*||g'
```

A.14

```
curl -X GET -o /dev/null -s -D - "https://demo.filestash.app/" | \
  grep "X-Powered-By: "
```

A.15

```
curl -X GET -s "https://demo.filestash.app/healthz" | \
  grep '{"status": "pass"}'
```

A.16

```
curl -X GET -s "https://demo.filestash.app/.well-known/security.txt"
```

A.17

```
# precondition: install plg_admin_nil
curl -s -X GET -o /dev/null http://localhost:8334/admin | \
  grep "HTTP/1.1 404 Not Found"
```

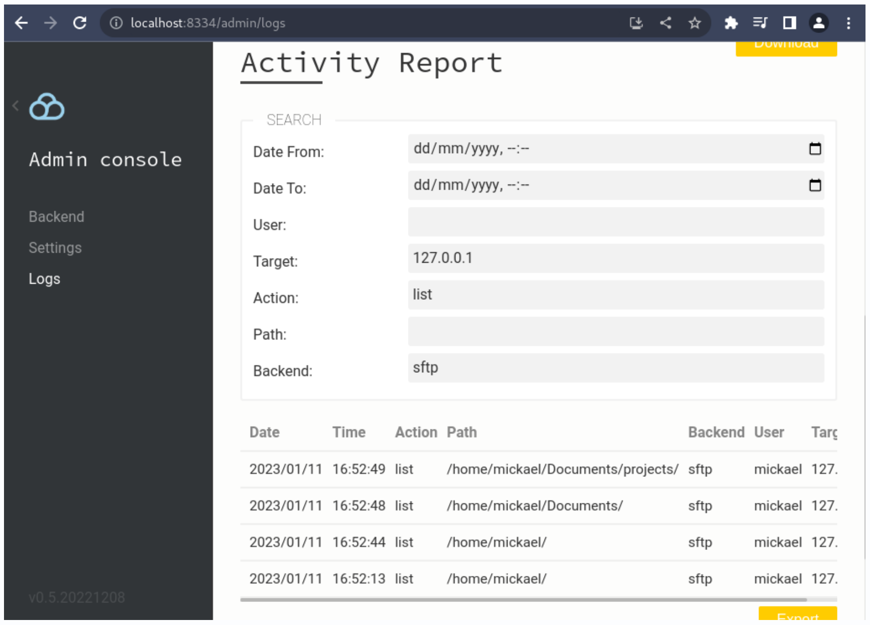
A.18

```
curl -s "https://demo.filestash.app/about" | \
  grep -e "STANDARD" -e "EXTENDED" -e "CUSTOM" | \
  sed 's|<[^>]*>||g' | sed 's|^[[:space:]]*||g' | \
  grep "plg_authenticate_"
```

A.19

```
echo '{"type":"blackhole"}' > /tmp/post.json
ab -n 5000 -c 1 -p /tmp/post.json \
  -H "X-Requested-With: XmlHttpRequest" \
  "http://localhost:8334/api/session" 2> /dev/null | \
  grep -e "Complete requests:" -e "Non-2xx responses:"
```

A.20



Activity Report

SEARCH

Date From:	dd/mm/yyyy, --:--
Date To:	dd/mm/yyyy, --:--
User:	
Target:	127.0.0.1
Action:	list
Path:	
Backend:	sftp

Date	Time	Action	Path	Backend	User	Targ
2023/01/11	16:52:49	list	/home/mickael/Documents/projects/	sftp	mickael	127.
2023/01/11	16:52:48	list	/home/mickael/Documents/	sftp	mickael	127.
2023/01/11	16:52:44	list	/home/mickael/	sftp	mickael	127.
2023/01/11	16:52:13	list	/home/mickael/	sftp	mickael	127.

A.21

```
curl -X GET -s "https://demo.filestash.app/about" | \
  grep -e "STANDARD" -e "EXTENDED" -e "CUSTOM" | \
  sed 's|<[^>]*>||g' | sed 's|^[[:space:]]*||g' | \
  grep "plg_security_scanner"
```

A.22

```
curl -X GET -s "https://demo.filestash.app/about" | \
  grep -e "STANDARD" -e "EXTENDED" -e "CUSTOM" | \
  sed 's|<[^>]*>||g' | sed 's|^[[:space:]]*||g' | \
  grep "plg_security_killswitch"
```