Practical

Writing a Dockerfile for a Somewhat-Secure Docker Container

Dockerfile, robust image and best security practices

Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu

2023–2024

Télécom SudParis
Institut Mines-Télécom & Institut Polytechnique de Paris

Part I.
Introduction

1. Overview

Containers are a lightweight form of virtualization at the level of the operating system (OS). As such, they provide some security guarantees. In particular, they isolate the filesystem: processes inside a container are shown a limited portion of it, that is built from *an image*. In the Docker world, container images are built from a recipe that is called a *Dockerfile*. Writing an unsecured Dockerfile, thus producing a vulnerable image, can undermine all efforts to securely execute a container.

In this practical, you will write a Dockerfile *to containerize a custom service* called PDFMagick, that converts images to PDF. This service is written as a small Python script, but depends on ImageMagick,[1] some software that can be used to "create, edit, compose, or convert digital images"; in your case, it will convert to PDF, JPEG images that are uploaded to your service via a web form served by the Python script.

However, at first the Dockerfile *will be unsecured in the face of a vulnerability of its dependency* ImageMagick, that is called ImageTragick[2] (see section 2). You will understand how the resulting Docker image is vulnerable, and implement mitigations and defences against it.

Information

This practical is not an exhaustive overview of all possible threat vectors that you would try to cover in a real deployment. Rather, it means to show that containerizing an application does not magically make your service secure, and to give a few best security practices. More precisely, you will be demonstrated why:

1. a container image must be immutable;
2. a running container must have as few capabilities as possible;
3. a containerized process should not run as root.

[1] `https://imagemagick.org./`
[2] `https://imagetragick.com/`

2. ImageTragick

CVE-2016-3714, dubbed ImageTragick, is a vulnerability in ImageMagick that exploits flaws in the image decoding code. They lead to reading files, and other issues up to remote code execution. The vulnerability is easy to exploit by crafting images in the MVG format[3] that include a delegation to a filename.

This is an old vulnerability, that has been fixed for a long time.[4] In this practical, you will deliberately use an outdated, vulnerable version of ImageMagick.[5] It may seem artificial, but *such vulnerability is bound to emerge again* in the real life. This constitutes a good example of flaws that you "import" into your system, and that you need to defend against: one of your dependencies is vulnerable, and you are powerless to fix it until it gets updated, so mitigations and defences must be put in place.

Information

While you do not need to understand the vulnerability in itself (a script is distributed with the practical to build payloads to trigger the vulnerability) it is an interesting example of insufficient filtering in arbitrary user input.

3. Prerequisite

You will be deploying a container image, so a working Docker installation on Linux is needed. In addition, you will use a small Python script to build payloads in order to exploit a vulnerability in a dependency of your containerized service, so you need Python 3.

4. Acknowledgements

Thanks to Redhpm for the multi-stage version of the Dockerfile.

[3] A SVG-like, ImageMagick specific file format, see `https://imagemagick.org/script/magick-vector-graphics.php`.
[4] Starting with ImageMagick version 6.9.3-10.
[5] Version 6.9.2-0.

Part II.
Setup

In this part, you fetch the required base files to containerize your service.

Jump into your work directory and execute the command in listing 1.

Listing 1: Download base files for the practical.

```
curl https://www-inf.telecom-sudparis.eu/COURS/CSC5004/practicals/\
    secure-dockerfile.tgz | tar -xz
```

Warning

It is mandatory to work on this practical in a new, empty directory. You will be explained why in section 8.

You should obtain three files:

1. `Dockerfile`: a template of a Dockerfile for your service;
2. `itragick_builder.py`: a Python script to build payloads to trigger the ImageTragick vulnerability;
3. `pdfmagick.py`: the Python script of your service.

You will work on the Dockerfile, filling in blanks, and you will execute `itragick_builder.py` a few times to build payloads to exploit ImageMagick's vulnerability.

Part III.
A Dockerfile for PDFMagick

First, you will write the Dockerfile to build PDFMagick's Docker image.

Information

The Dockerfile is `written` as a multi-stage build. As explained at `https://docs.docker.com/build/building/multi-stage/`:

> With multi-stage builds, you use multiple FROM statements in your Dockerfile. Each FROM instruction can use a different

base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.

For this lab, you only write the runtime stage starting at line 83.

Question 1

Using the online documentation for Dockerfiles at `https://docs.docker.com/engine/reference/builder/`, take the time to understand the initial content of `Dockerfile` in the runtime stage.

- What does the `FROM` instruction do?
- What does it mean to `COPY --from=build`?
- What is the `RUN` instruction used for?
  - Why are the Shell commands in the `RUN` instructions grouped the way they are, and why are they chained together with `&&` operators? Think of image layers and caching.
  - In a related way, why do `RUN` instructions that install dependencies come in the Dockerfile *before* the blank where you will include your Python script?

There are three places where you are expected to add your own Dockerfile build instructions:

1. installation of Python dependencies;
2. import of your application Python script;
3. setup of the container interface: network ports and default command.

5. Installation of Python dependencies

The first blank space to fill in the Dockerfile is about installing the remaining dependencies for your application.

Question 2

Write a Dockerfile instruction to install the Debian packages `python3` and `python3-pip`. Then, using pip, install the Python dependency `web.py`.

Information

The image is built upon the Debian distribution[a]. To install packages, you first need to update the list of packages on mirrors using the command `apt-get update`, and then you can install packages with the command `apt-get -y install PKG`; the flag `-y` makes it accept any choice proposed to the user, which is necessary for an automated installation such as in a Dockerfile.

Additionally, installing the package `python3-pip` provides the command `pip3`. To install a Python package using pip, you use the command `pip3 install PKG`.

[a] `https://www.debian.org/`

6. Import of your application Python script

The second blank space to fill is about importing your application.

Question 3

Write a Dockerfile instruction to include the Python script `pdfmagick.py`, that represents your application, in the Docker image. It should put the file at the root `/` of the image.

Information

Look at the documentation for Dockerfiles at `https://docs.docker.com/engine/reference/builder/` to find the Dockerfile instruction.

You will find there are actually two possible instructions to achieve this goal: `ADD` and `COPY`. `COPY` offers less functionality: it only copies the files into the image, whereas `ADD` can download from a URL and expand archives. *This is a feature of* `COPY`, and it is recommended to use `COPY` in any case (downloading and expanding an archive is preferably achieved with a `RUN` instruction).

7. Setup of the container interface

Finally, the third space to fill is about declaring the network interface of your containerized application (i.e. the network ports it will use), and defining the command to run.

Question 4

Write a Dockerfile instruction to declare the network port in use by your application, which is `8080`.

Warning

By reading the documentation for Dockerfiles at `https://docs.docker.com/engine/reference/builder/`, you will quickly find out about the `EXPOSE` instruction.

This instruction actually does nothing by itself. It only informs Docker that the container listens on the specified network ports at runtime. It is when actually running a container from this image that you must explicitly *publish* the ports; an `EXPOSE` instruction is actually useful to *automatically expose* all declared ports. Quoting the online documentation for Dockerfiles:

> It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published.

Question 5

In the same last blank space, write an instruction to define the default command to run when creating a container from the PDFMagick image. The command is shown in listing 2.

Listing 2: Command of the PDFMagick Docker image.

```
python3 /pdfmagick.py
```

Information

Again, two instructions may fit the need here: `CMD` and `ENTRYPOINT`. Take the time to understand the differences and interactions between them by reading their documentation at `https://docs.docker.com/engine/reference/builder/`.

8. Building the image

To build your image, run the command in listing 3.

Listing 3: Command to build PDFMagick Docker image.

```
docker build -t 'net5039/pdfmagick:v1' .
```

Information

With the command given in listing 3, you build your image under the label `net5039/pdfmagick:v1` in your local repository.

- `net5039` is the namespace, it can be your name, a project name, etc.;
- `pdfmagick` is the image name;
- `v1` is the tag, it is used to point at a specific variation or version of the image.

Wherever an image is named in a Docker command and a tag is needed, Docker defaults to the tag `latest`.

Warning

When running the `docker build` command, *Docker makes a recursive copy of the target directory* (here, the current directory `.`) to an isolated space. To avoid unnecessary copying, the parent folder of the Dockerfile should only contain files that would end up in the container image, or are useful for the build process.

You can tell Docker to ignore files and directories by listing them in a `.dockerignore` file.

After some time, the command ends.

Information

What you saw unrolling in your terminal are two things:

1. the normal output of the commands that you run in your Dockerfile with `RUN` instructions (`apt-get`, `pip`...);
2. the Docker-specific output of building the container image, enumerating steps and building *image layers* stored under hashes.

Running the build command again will *reuse cached image layers* (see question 1), such that the build process terminates instantly. Modifications of the Dockerfile, as they will happen later in this practical, will only

trigger the rebuilding of the image layers that you modified, or the ones happening after the modified layers.

You can confirm the existence of an image labelled `net5039/pdfmagick`, with the tag `v1`, by running the command in listing 4.

Listing 4: Check the built image.

```
docker images "*/pdfmagick*"
```

9. Testing the image

Finally, run a container from your image, as shown in listing 5.

Listing 5: Command to run PDFMagick from your image.

```
docker run --name pdfmagick --rm --detach --publish 8080:8080 net5039/pdfmagick:v1
```

Information

In the command shown in listing 5:

- `--name pdfmagick` gives a name to the container that will be created (this is independent from the image's name);
- `--rm` tells Docker to destroy the container when it exits (i.e. when its initial process terminates);
- `--detach` tells Docker to daemonize the container, i.e. its main process is blocking and should run as a background process;
- `--publish 8080:8080` publishes ports to reach your container: the first `8080` is the port on the host, the second `8080` is the port inside the container (which corresponds to the Dockerfile instruction written in section 7).

No command is given to run in the container: *it will execute the default command* defined in section 7. Concerning port publishing, Docker defaults to binding to the `localhost` address.

As can be guessed from the command, your service is now up and running, listening for new connections on `http://localhost:8080`. Fire a web browser to this URL: you will find a form asking to upload an image. Submit such an image, and the service responds with the download of a PDF file, resulting from the conversion of the image.

Congratulations, you successfully containerized an application with its dependencies!

10. Summary: writing a Dockerfile

As a reference, listing 6 gives examples of Dockerfile instructions you should have written in the blank spaces.

Listing 6: Initial version of PDFMagick's Dockerfile (excerpt of blanks).

```
##########
# STUDENT CODE GOES HERE
# so that above layers remain in cache
#
# Instructions:
# * in this Debian distribution, you want the packages python3 and python3-pip
# to  install Python 3 and pip for this version of Python
# * then, you want to install the dependency web.py using pip
##########
RUN apt-get update && \
    apt-get install -y python3 python3-pip && \
    rm -rf /var/lib/apt/lists/* && \
    pip3 install web.py
##########

##########
# STUDENT CODE GOES HERE
#
# Instructions:
# * add the Python script to the image
##########
COPY [ "pdfmagick.py", "/" ]
##########

##########
# STUDENT CODE GOES HERE
#
# Instructions:
# * indicate that the image exposes port 8080
# * set the entrypoint to run pdfmagick.py using python3
##########
EXPOSE 8080

ENTRYPOINT [ "python3", "/pdfmagick.py" ]
##########
```

Now, let's check the security of your container!

Part IV. Security testing PDFMagick

In this part, you will break your PDFMagick service by exploiting:

- its mutability;

- its unnecessary capabilities;
- its unnecessary privileges from running as root.

And you will also fix those vulnerabilities.

As explained in section 2, exploits go through a vulnerability in ImageMagick when processing an image. In the following sections, you will craft payloads for this vulnerability using the script `itragick_builder.py`.

Information

Running `./itragick_builder.py` will print a help message that is reproduced below.

```
Usage: ./itragick_builder.py OUTFILE EXPLOIT ARG
Prepare a payload to exploit the ImageTragick vulnerability
with a MVG file.
OUTFILE is the path to the output file (a MVG file).
EXPLOIT is either:
* rce, for remote code execution
* read, to read a local file into the image
ARG is the argument for the chosen exploit:
* for rce: the command to embed in the payload
* for read: the path of the file to read
```

For example, to build a remote code execution (RCE) exploit for the command `touch /pwned`, you would run the command in listing 7.

Listing 7: Example usage of `itragick_builder.py`.

```
./itragick_builder.py exploit-touch.mvg rce "touch /pwned"
```

Do not put simple quotes `'` inside the RCE command, or it will break the structure of the built image that contains the payload.

11. Exploiting image mutability

First, let's try to break the service by writing to its main Python script `/pdfmagick.py`.

Question 6

Use the script `itragick_builder.py` to build a payload for a *remote code execution*; the command of the RCE should write something (for example "PWNED") to the file `/pdfmagick.py` in the container. What happens when you upload the payload (the MVG image) to the service?

To confirm you intuition, run the command in listing 8 to display the changes made in the container `pdfmagick` to its image.

Listing 8: Command to check changes in a running container.

```
docker diff pdfmagick
```

Additionally, run the command in listing 9 that uses `docker exec` to run a command inside a running container.[6] Here, it displays the file that was corrupted using `cat` to confirm your exploit.

Listing 9: Command to display the content of a file inside a running container.

```
docker exec pdfmagick cat /pdfmagick.py
```

Information

`docker exec` can be an handy debugging tool: with the code in listing 10, you can run a Bash shell inside the container. Notice the flags, that are mandatory when running an interactive program such as a shell.

Listing 10: Command to run a Bash shell inside a running container.

```
docker exec --interactive --tty pdfmagick bash
```

Warning

Note that `docker exec` executes a command that is available inside the running container (i.e. from its image). It means that `docker exec` is limited to those commands: if your container image does not come with the program Bash, you will not be able to run the command in listing 10.

[6] You are root on the hosting system, so this is different from the exploit where you execute arbitrary code while being an unprivileged user of the containerized service.

Question 7

Now that you have the power to overwrite the code executed by the container, how far could the exploit go?

This particular exploit of the vulnerability can be defended against by making the script `/pdfmagick.py` unwriteable, i.e. by removing the `w` bit in the access rights. In essence, this is about making the image *immutable*.

Information

It is also possible *to have Docker mount the entire root filesystem of a running container as read-only* with `docker run --read-only`. This would defend all files against such an exploit, however this cannot be implemented for every container, such as your PDFMagic service, because it writes its processed files to its filesystem. In this case, a fitting solution is to use a dedicated volume.

Question 8

Implement removing the write bit in the access rights, by writing an instruction in the Dockerfile just after importing the script into the image.

Test your fix:

1. rebuild the Docker image *under a new tag* (for instance `v2`)
2. stop the currently running — and broken — container with the command `docker stop pdfmagick`
3. start a new one from the newly built image.

Refer to listings 3 and 5 for the commands to achieve this.

Information

Another side of the "immutability" of Docker images is *to avoid building a different version of the same image under an existing tag*. By doing so, you guarantee the users of the image that its content will not change unexpectedly, which could lead to breakages that are hard to trace down.[a]

[a] `https://www.whitesourcesoftware.com/free-developer-tools/blog/overcoming-dockers-mutable-image-tags/` is a good read on this subject.

Reload the web page of your service at `http://localhost:8080`, and test it again with the same exploit payload.

Question 9

Confirm that your defence is in place by running `ls -l /pdfmagick.py` inside the container.

With the hint that Docker retains a few *Linux kernel capabilities* on its containers, study the documentation about capabilities of Docker containers at this page: `https://docs.docker.com/engine/security/#linux-kernel-capabilities`. Why does the exploit still work?

Dropping the write right bit on the script is not enough to protect against this exploit: you need *to drop the corresponding Linux kernel capability*.

12. Exploiting unnecessary Linux kernel capability

By following the lead of the Linux kernel capabilities, you would arrive at the conclusion that the root account inside the container retains the `CAP_DAC_OVERRIDE` capability. Simply put, this capability allows to ignore the rights bit of a file, rendering your fix above inefficient.

Stop the currently running and broken `pdfmagick` container.

Question 10

Using the flag `--cap-drop` of the command `docker run`, run the PDFMagick container from the same previous image with the `CAP_DAC_OVERRIDE` capability dropped.

Test the exploit again: what happens?

Congratulations, you defended against this exploit! But more await...

13. Exploiting execution as root

In this section, you will use the "read" exploit of ImageTragick.

Question 11

Using `itragick_builder.py`, build a read exploit to the file `/etc/shadow` and test it: upload it to your service, and display the resulting PDF.

Given that the file must remain readable by root (and is not readable by any other user) how would you defend against such an exploit?

Stop the currently running container and go back to editing the Dockerfile for one last version of it.

Question 12

Before defining the command of the container image (see question 5), write a Dockerfile instruction to have it run as the user `nobody`. You can find the instruction in the usual online documentation at `https://docs.docker.com/engine/reference/builder/`.

Information

In this case, you make it run as `nobody`, a user that already exists in the base Debian image. To use a custom user, you must create it in the image beforehand, by running a command such as `useradd`.

Build this new version of the Docker image under a new tag (for example `v3`) and test it with the exploit.

Congratulations, you secured your container against a few exploits!

14. Summary: writing a fairly-secure Dockerfile

In this part, you experimented with a few security measures that must be taken when writing a Dockerfile and running a container:

1. run as an unprivileged user;
2. drop unnecessary capabilities;
3. make the container image as immutable as possible.

Warning

Those three security measures are *complementary*! Despite the fact that running as non-root will successfully defend against every exploit shown in this practical, dropping capabilities and making the container immutable are effective mitigation measures in case an exploit such as privilege escalation arises.

Information

A good practice for security and reusability when writing a Dockerfile, is *to start from an image as minimal and/or as focused as possible*, and then to extend it as minimally as possible. For example, you could start `FROM python` when your service is written in Python, or you could make sure caches and build dependencies are cleaned up.

From a usability point-of-view, it keeps container images at a small size; and from a security point-of-view, it minimizes the attack surface: you cannot be attacked via a piece of software that you don't include!

As a reference, listing 11 gives examples of Dockerfile instructions you should have written in the blank spaces.

Listing 11: Fairly secure version of PDFMagick's Dockerfile (excerpt of blanks).

```
##########
# STUDENT CODE GOES HERE
# so that above layers remain in cache
#
# Instructions:
# * in this Debian distribution, you want the packages python3 and python3-pip
# to  install Python 3 and pip for this version of Python
# * then, you want to install the dependency web.py using pip
##########
RUN apt-get update && \
    apt-get install -y python3 python3-pip && \
    rm -rf /var/lib/apt/lists/* && \
    pip3 install web.py
##########

##########
# STUDENT CODE GOES HERE
#
# Instructions:
# * add the Python script to the image
##########
COPY [ "pdfmagick.py", "/" ]
RUN chmod 444 "/pdfmagick.py"
##########

##########
# STUDENT CODE GOES HERE
#
# Instructions:
# * indicate that the image exposes port 8080
# * set the entrypoint to run pdfmagick.py using python3
##########
EXPOSE 8080

USER nobody

ENTRYPOINT [ "python3", "/pdfmagick.py" ]
##########
```

Additionally, listing 12 gives a command to run a container from this image with the additional security of the capability `CAP_DAC_OVERRIDE` dropped.

Listing 12: Command to drop the capability `CAP_DAC_OVERRIDE` before running PDFMagick.

```
docker run --name pdfmagick --rm --detach \
    --publish 8080:8080 --cap-drop=CAP_DAC_OVERRIDE \
    net5039/pdfmagick:v3
```

Part V. Conclusion

In this practical, you wrote a Dockerfile to build a container image for your service, and then secured it against a vulnerability of one of its dependencies. By doing so, you experienced various best practices and you understood the limits of the security guarantees of containers: while you did not escape the isolation of the container, you subverted its service.

To go further, you can try to write the service by starting from a more focused base image, e.g. the official Python image.[7] You can also rewrite the service using a multi-stage build, a rather novel feature of Docker to build cleaner images: `https://docs.docker.com/develop/develop-images/multistage-build/`.

Explore all the images publicly available on the DockerHub[8]; what parts of your own computer system could you containerize, and for what gains?

On the security side, finish reading the Docker security page at `https://docs.docker.com/engine/security/` for more insights than this practical can give you; in addition, a topic that is not covered here is resource management, that you can read more about at `https://docs.docker.com/config/containers/resource_constraints/`. You can also follow Docker's lab on advanced security of container using seccomp at `https://training.play-with-docker.com/security-seccomp/`. Finally, read more about "rootless mode", i.e. running the Docker daemon on the host at `https://docs.docker.com/engine/security/rootless/`, which is useful to protect the hosting system against compromised containers.

[7] `https://hub.docker.com/_/python/`
[8] `https://hub.docker.com`