Software Security - Exercise 2

TDT4237 Group 80
Dusan Jovanovic, Guan-Lin Cheng
NTNU, Norway

Abstract

This document represents a report on TDT4237 Software security exercise number 2. Vulnerabilities previously found in the application should be corrected and fixed, the document explains this procedure. These vulnerabilities cover a large spectrum of security issues listed in "OWASP Top 10" list brought by Open Source Foundation for Application Security.

Keywords: software security source code owasp

Contents

1. Introduction

Scope of this report is focused on finding, fixing and documenting security issues. Every issue will be thoroughly documented with additional screenshot images or source code if applicable.

2. Mitigation of vulnerabilities

This section contains descriptions of each vulnerability that has been fixed.

2.1. A1:2017-Injection

"Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization."

In the previous report it was discovered that the application is vulnerable to injection on multiple pages such as */projects* or */login*. We had a look at the code behind exploited vulnerabilities and made essential refactoring and fixes.

In total three files were changed in the *models* directory - *register.py*, *project.py* and *user.py*. As an example we will have a look at one of those files since the logic behind fixes is identical. The example file is *project.py*.

```
def update_task_status(taskid, status):
...
query = ("UPDATE tasks SET task_status = \"" + status +
"\" WHERE taskid = \"" + taskid + "\"")
try:
cursor.execute(query)
db.commit()

def update_task_status(taskid, status):
...
sql_cmd = """UPDATE tasks SET task_status = %s WHERE taskid = %s"""
sql_value = (status, taskid)
try:
cursor.execute(sql_cmd, sql_value)
db.commit()
```

The crucial difference is in **building and executing queries** in source code. In the first source code example we can see the old code where a query is build in-place allowing an attacker to inject malicious SQL code through variables.

In the second code excerpt we can see another approach where a query has two parts - command and values. Query is being built out of these parts and then executed after. With this refactor the attacker cannot inject malicious code. SQL parameters are passed as tuples and **Database API does proper escaping and quoting of variables** thereby eliminating injection vulnerability.

There are multiple requests, both POST and GET, which were found as vulnerable in the previous report. All of these have been corrected using this solution described here and can be inspected in source code if needed. The report covers only one example in a file *project.py* since the same refactor has been done on all other places.

2.2. A10:2017-Insufficient Logging and Monitoring

"Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring."

In the previous report it was discovered that the application is leaking important information once an error occurs, especially database-related information. We had a look at the code behind exploited vulnerabilities and made essential refactoring and fixes.

Logging is composed of printing crucial stack traces and showing them to the user, mostly found in the *models* directory for example.

```
except mysql.connector.Error as err:
print("Failed executing query: {}".format(err))

except mysql.connector.Error as err:
log_error_msg(err)

...

def log_error_msg (msg):
try:
handler = logging.handlers.WatchedFileHandler(
os.environ.get("LOGFILE", "/var/log/project_logs.log"))
handler.setFormatter(logging.Formatter(logging.BASIC_FORMAT))
root = logging.getLogger()
root.setLevel(os.environ.get("LOGELVEL", "INFO"))
root.addHandler(handler)
except Exception:
logging.exception("Exception in main()")
exit(1)
return
```

As we can see, instead of printing the error message to the user (if and when it happens), the error is **being logged at /var/tmp directory** which can be observed in the container. In this case, the user never gets to see internal system information being printed - instead it is logged locally on the server for administrative purposes. This refactor has been done in many database-related operations and can be observed in the source code.

2.3. A9:2017-Using Components with Known Vulnerabilities

"Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts."

Two files are used to specify package versions used in this project.

/src/app/requirements.txt

```
web.py==0.40
mysql-connector==2.2.9
python-dotenv
```

```
web.py
mysql-connector
python-dotenv
```

Python packages have old versions listed, versions with patched security holes and these should be omitted from the build and changed to newest available. This can be achieved by removing the version number from the specification text file.

/mysql/Dockerfile

```
FROM mysql:5.7.15

MAINTAINER me

ENV MYSQL_DATABASE=db \
MYSQL_ROOT_PASSWORD=root

ADD /sql/init.sql /docker-entrypoint-initdb.d

EXPOSE 3306
```

```
FROM mysql:latest

MAINTAINER me

ENV MYSQL_DATABASE=db \
MYSQL_ROOT_PASSWORD=root

ADD /sql/init.sql /docker-entrypoint-initdb.d

EXPOSE 3306
```

On the other hand, the Dockerfile specifies which version of needed runtime components will be used. As we can see, **mysql:latest** dictates using the newest available version of mysql in this setup. Using this version allows a more secure way of password-policy handling.

2.4. A6:2017-Security Misconfiguration

"Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion."

Session timeout

Session timeout is too high by default for this application. File */src/app/views/app.py* is used to set this configuration.

```
# Set session timeout
#edited in March.18
web.config.session_parameters['timeout'] = 15000000

# Set session timeout
#edited in March.18
web.config.session_parameters['timeout'] = 1500
```

Timeout has been greatly reduced and limiting the attacker with an expandable time-frame of cookie validity.

Clickjacking

Clickjacking protection is based on adding **mandatory headers** to every request transmitted from the server and limiting attacker when it comes to adding X-frames and X-content elements. Needed headers are:

```
X-Frame-Options SAMEORIGIN;
X-Content-Type-Options nosniff;
```

A file *entrypoint.sh* is used to specify ngnix server configuration and thereby needed lines are found in the code excerpt below, previously these headers were missing from the configuration.

```
content_server=$content_server'     add_header X-Frame-Options \"sameorigin\" always;\n'
content_server=$content_server'     add_header X-Content-Type-Options \"nosniff\";\n'
```

Cross-site request forgery

CSRF protection is needed whenever a form is present with a **POST method** thereby preventing cross-site requests taking place.

Forms in multiple files have been corrected and injected with a CSRF token. For example, we will explain the registration form. Firstly, *utils.py* file contains two now methods.

```
def csrf():
session = web.ctx.session
if not session.has_key('csrf_token'):
from uuid import uuid4
session.csrf_token = uuid4().hex
return session.csrf_token

def csrf_decorate(f):
def csrf_base(*a, **k):
session = web.ctx.session
input = web.input()
if not (session.has_key('csrf_token') and input.csrf_token == session.pop('csrf_token', Nor
raise web.HTTPError("400 Bad request", {'content-type':'text/html'}, 'Request forgery!')
return f(*a,**k)
return csrf_base
```

First method should generate a token, and the second one is a decorator function used to raise an error if token is missing. If we then look at the register logic file *register.py* we can see that the token generation is made global. Also, the self-method of POST should be decorated with handling of a missing token.

```
# Set global token
web.template.Template.globals['csrf_token'] = csrf

@csrf_decorate
def POST(self):
"""
Handle input data and register new user in database

:return: Main page
"""
```

Later on, the template file *register.html* should use this token and inject it within a form.

```
<form method="POST">
<input type="hidden" name="csrf_token" value="$csrf_token()" />

$:register_form.render()
</form>
```

Application error disclosure

As it was previously mentioned in Logging subsection, many error stack traces were just printed out to the user - especially frequent in sql exception handling. Let's look at the example again.

```
...
except mysql.connector.Error as err:
print("Failed executing query: {}".format(err))

...
except mysql.connector.Error as err:
logger.log_error_msg("Failed executing query: {}".format(err))
```

This kind of error-disclosure was present in every sql exception handling and corrected in a way as demonstrated in source code excerpt. All files that can be found in the *models* directory such as *register.py*, *project.py* and *user.py* have been affected with corrections such as this one and can be examined.

Insecure deserialization

Deserialization is used when storing/reading **the remember me cookie** because the used python package is *pickle* and can be considered insecure in this process.

```
import os, hmac, base64, pickle
...

username, sign = pickle.loads(decode)
...

return base64.b64encode(pickle.dumps(creds))
```

Changing this package to *json* eliminated this vulnerability and provides the same behavior as previously regarding the cookie.

```
import os, hmac, base64, json
...

username, sign = json.loads(decode)
...

return base64.b64encode(json.dumps(creds))
```

This kind of error-disclosure was present in every sql exception handling and corrected in a way as demonstrated in source code excerpt. All files that can be found in the *models* directory such as *register.py*, *project.py* and *user.py* have been affected with corrections such as this one and can be examined.

File upload

Before any changes were made users could upload files of any format and no validation was present. Most of the code found in this document was not present before the fix and can be observed.

```
#upload filname with basename, to avoid path travsal
fn = os.path.basename(fileitem.filename)

...

#check the filename suffix
fn.lower().endswith(('.txt', '.pdf'))
#extract file name itself
filename_no_extension = os.path.splitext(fn)[0]
#hash file name
hashed_fileanme = hashlib.md5(filename_no_extension.encode())
#create the file path
uploaded_file_path = os.path.join(path, "/", hashed_fileanme.hexdigest())
open(uploaded_file_path, 'wb').write(fileitem.file.read())
models.project.set_task_file(data.taskid, uploaded_file_path)
except:
# Throws exception if no file present
# Get session
session = web.ctx.session
# Get navbar
nav = get_nav_bar(session)
data = web.input(projectid=0)
if data.projectid:
project = models.project.get_project_by_id(data.projectid)
tasks = models.project.get_tasks_by_project_id(data.projectid)
else:
project = [[]]
tasks = [[]]
render = web.template.render('templates/', globals={'get_task_files':models.project.get_t
return render.project(nav, project_form, project, tasks,permissions, categories,)
pass
```

Firstly, *os.path.basename* is used to prevent file traversal where filenames are stripped. Afterwards, files are limited to two formats - *.pdf* and *.txt*. Lastly, files are saved with hashed filenames relying on the *md5* hashing algorithm.

2.5. Optional-Old files

One file was found which was left-over during the development process - */src/app/static/friends.png*. This file has been removed from the server now.

No further explanation is needed since this is not a serious security risk - just a cosmetic refactor.

3. References

In the process of formulating and writing this report, OWASP's resources have been used as a guide and source of quotations in this document.

https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf

https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_Top_10

References