# Implementation

## Automated Testing

The fastest, most efficient way to perform an automated test is to make use of one of the many pen-test assisting tools that exist. For the purposes of this project, OWASP ZAP (Zed Attack Proxy) will be used to automatically scan a target web application.

ZAP is an integrated penetration testing tool that can be used to find a wide array of vulnerabilities in a web application. The tool is meant both for unexperienced beginners as well as professionals. ZAP provides tools for manual testing as well as automated scanners, but this project will only make use of the second.

The script will be written in Python. Python is a high-level programming language that puts emphasis on code readability. Unlike many other programming languages that break up code blocks with curly braces or specific keywords, Python instead uses whitespace indentation to mark the indentation level of each line of code. OWASP ZAP has a Python client API that allows it to work directly through Python by use of the ZAPv2 module.

In order to perform an automated scan on the target web application, the first step is to open the OWASP ZAP program. This can be achieved with the help of the ‘OS’ and ‘Subprocess’ modules for Python. The program will be started in headless mode (without a graphical interface) in order to minimize loading times. The script will continuously try to connect to the target webpage until it can detect the program. Once the program is open, the actual scan can be divided in two stages: the spidering stage and the active scan stage.

The program is only supplied with the root URL (parent page) of the web application, so it needs some way to find its subdirectories (children pages). A spider or crawler is a bot that does exactly that, fetching any children pages under the parent URL to later scan them. OWASP ZAP features a spider that can be directly called from Python. If the scan takes too long, it is possible to limit the number of children to scan in the code. This would reduce the total scan time, but would also lower the number of vulnerabilities found.

While the program is running the spider it passively scans every received response, searching for easy to find vulnerabilities (similar to a checklist). This is a passive scan done in the background without interrupting the exploration of the web application.

Once an appropriate number of URLs has been fetched, the proper scanning stage begins. Unlike the passive scan performed during the spidering process, this is an active scan that applies attacks or payloads against the target application in order to find vulnerabilities.

Once the scan is done, OWASP ZAP saves all the collected information pertaining to the found vulnerabilities inside a variable that will later be accessed and analyzed in order to get the needed data to generate the report. As a last step before the report generation, the headless instance of OWASP ZAP is closed via Python.

## Report Generation

The next step is to make use of the information gathered during the previous step to generate a human-readable report that summarizes the vulnerabilities found during the scan and their severity. The report will be a pdf file generated using GitBook.

First, it’s necessary to create a template for the report in GitBook. GitBook uses Jinja2, a templating language for Python modelled after Django’s templates. Its syntax makes it possible to not only create templates for variables, but also to iterate through lists and use conditionals.

GitBook will take into account any variables found within the *book.json* file when generating the pdf file. The purpose of the code will then be to dynamically create this *book.json* file using the appropriate variables obtained during the previous scan. This is achieved by collecting the variables of interest inside a list that is later dumped to the json file.