# Implementation

The code 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.

The program will execute in three distinct stages. It will first begin by gathering all information it will need to work, such as what IPs and ports to scan. The second stage will be the actual vulnerability scan per se, taking in the information from the previous step as the input and sending the output to the next and last stage, the automated generation of the report.

The code will make use of diverse third-party tools such as Nmap or OWASP ZAP, which will be described in the following sections as they are introduced. Most of these tools have specific modules that allow them to easily interact with Python, and grant almost full control of their functions through scripting.

## Information Gathering

Before the scan can start it is necessary to know what web applications will be scanned, the IP address of their hosts and which port do they reside in. The information gathering phase will be divided in three sequential stages: a first stage of interface scanning to detect what interfaces the device hosting the script is connected to and convert them to CIDR format, a second stage of network scanning to discover live hosts, and a third and final stage of port scanning. Nmap will be used in the two later stages.

Nmap[[1]](#footnote-1) (“Network Mapper”) is a free, open source utility used for diverse tasks such as network discovery and security auditing. It was conceived as a tool to assist in the scanning of large networks, but also works against single hosts. Some of the multiple techniques it implements include port scanning, OS and version detection or ping sweeps, among others. A module for Python exists[[2]](#footnote-2), which makes it easy to implement Nmap functions directly through Python.

### Interface Scan

Before a network can be scanned, it is imperative to first determine *which* network is to actually be scanned. So, the first step will be to determine what network the device is connected to.

The device, by virtue of being connected to a network, will be granted an IP address. These IP addresses are used to identify any particular device inside a given network. They usually take the form of a set of four numbers, each ranging from 0 to 255 (such as 127.0.0.1). Since the device will know its own IP address at any point, it can be used as a starting point to determine the network.

Other than an IP address, a network is identified by its netmask, which determines what other IP addresses belong to the same subnet and should also be scanned in the following stages. It can be represented as a set of four numbers much like an IP address, or as a number denoting how many bits in an address are part of the network, out of a total 32 bits. For instance, a 20 would denote that the first 20 bits are the network part of the address, and so the other 12 bits are used for specific host addresses under the same subnet.

Thus, a network can be fully characterized by an IP address and a netmask. Knowing this, it’s common to represent a network in a human-readable format that includes this information. The most widely accepted representation is CIDR[[3]](#footnote-3) (Classless Inter-Domain Routing) notation. Under this notation, networks are written as an address followed by a slash and then the number of bits of the netmask (such as 192.168.0.0/16).

The *netifaces* module for Python makes it trivial to fetch information about all interfaces the device is connected to. Then, the *socket* module included by default in Python can help find information related to their IPs, such as their address and netmask. Finally, the *netaddr* module can be used to turn this information into a usable CIDR form.

### Network Scan

Trying to scan every single port of every potential IP address would be so time-consuming as to be impractical. It is a slow procedure that is rarely necessary. More often than not, the only IP addresses of interest are those which hold live hosts.

Host discovery works by sending specific request packets (“pings”) to a wide range of IP addresses and then detecting what particular IP addresses answer back[[4]](#footnote-4). Nmap offers different ways to scan a network for live hosts, but the “no port scan” (“-sn” switch) will be chosen for being relatively less intrusive and faster than other alternatives, while still detailing what hosts are up in a given network.

The way it works is by sending an ICMP echo request (“Internet Control Message Protocol”) to each target and then waiting for a response. By default, Nmap will send a TCP ACK packet to port 80 and a TCP SYN packet to port 443. Ports 80 and 443 are considered the default ports for http and https, respectively.

The result of the network scan will be a list of IP addresses that responded to the requests sent by Nmap. This list will be saved so that Nmap can later scan the ports on each of these IP addresses one by one.

### Port Scan

Any given IP address will be associated with 65536 TCP ports and another 65536 UDP ports, where TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are two of the protocols that make up the TCP/IP protocol suite, universally used for data communication across the Internet[9]. The first 1024 TCP ports are commonly associated with specific services such as HTTP (80), HTTPS (443), DNS (53) or FTP (20). Note that this is merely a convention, and is not enforced (so there could be HTTP services in any port other than 80).

Port scanning consists on sending out connection requests to every port corresponding to an IP address sequentially, noting which ports respond. The scan can then follow up with a more in-depth scan of those ports that responded in order to determine what kind of service they are running. Scanning the whole range of 65536 ports is slow and rarely necessary, so it’s common for scanners to limit their reach to a smaller array of commonly used ports. By default, Nmap will only scan the 1000 more frequently used ports.

By default Nmap will run a SYN scan, capable of scanning thousands of ports per second in absence of restrictive firewalls[[5]](#footnote-5). This scan sends a SYN packet to every target port and waits to receive a SYN/ACK packet in response, which would indicate that the target port is open. Since this scan doesn’t open a full TCP connection, it is sometimes referred to as half-open scanning.

The scan will result in a list of open ports associated to each live host found during the previous step. Nmap can also perform service detection to determine with reasonable accuracy what kind of service is associated with each port. This allows the program to detect HTTP or HTTPS services outside of their most commonly use ports.

Once the program has composed a list of hosts and ports that might potentially have web applications, this list is sent to the next stage for an automated scan.

## 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. ZAP has a Python client API that allows it to work directly through Python by use of the ZAPv2 module[[6]](#footnote-6).

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[10]. 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 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.

According to the official OWASP ZAP documentation, the passive scan searches for any of the following vulnerabilities[[7]](#footnote-7):

-Application Errors

Checks server responses in order to find responses that contain known error strings.

-Cache Control

Checks the cache-control header field to ensure it follows the industry’s best practices.

-Content Type Missing

Ensures that the Content-Type header exists and its value isn’t empty.

-Cookie HttpOnly

If a cookie is set with the “HttpOnly” flag, browsers will know that only client side scripts should interact with that cookie. This check ensures that all cookies set are flagged as “HttpOnly”.

-Cookie Secure Flag

If a cookie is set with the secure flag, it will not be sent during a plain HTTP session. This check analyzes all cookies generated during HTTPS sessions to make sure they include this flag.

-Cross Domain Script Inclusion

Checks whether there are scripts that don’t come from the domain hosting the content.

-Header XSS Protection

The X-XSS-Protection header will keep a page from loading if it detects a potential XSS attack. This check confirms the existence of this header and whether it is correctly configured.

-Mixed Content

A HTTPS connection is considered more secure than a HTTP one, but not so if the content is sourced via HTTP, which would defeat the purpose of HTTPS in the first place. This check analyzes content served via HTTPS to ensure none of it is sourced via plain HTTP.

-Password Autocomplete

This is considered an unsafe practice. This check examines all password fields to ensure their ‘autocomplete’ setting is disabled.

-Private Address Disclosure

Checks whether the response contains private IP addresses (RFC 1918).

-Session Id in URL Rewrite

Ensures that the session token is not written to the URL.

-X-Content-Type-Options

MIME sniffing[[8]](#footnote-8) may occur when a document has no MIME type. In this situation, some browsers will try to guess the correct MIME type by looking at the resource. This represents a potential vulnerability, so the X-Content-Type-Options header exists to prevent MIME-sniffing. This check ensures that his header exists and is correctly set to ‘nosniff’.

-X-Frame-Options Header Scanner

The X-Frame-Options header[[9]](#footnote-9) defines how a page will be rendered. It can forbid the page’s content from being embedded into other sites, which can prevent click-jacking attempts. This check ensures that this header exists and is valid.

Once an appropriate number of URLs has been fetched and the passive checks have been applied, 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.

According to the official OWASP ZAP documentation, the attempted attacks are as follows[[10]](#footnote-10):

-Buffer Overflow

In the case of a buffer overflow, the extra data might get written into a sensitive memory location. To check for this flaw, the program sends large strings of input text and checks for anomalous behavior.

-Client Browser Cache

Checks the headers of secure pages to ensure they don’t allow browsers to cache the page.

-CRLF Injection

CRLF refers to the Carriage Return and Line Feed special characters used to denote the end of a line. It might be abused by an attacker to split a HTTP response, which can lead to cross-site scripting or cache poisoning[[11]](#footnote-11). The program tests for this vulnerability by trying to inject a new header into the response by sending a parameter preceded by CRLF characters.

-Cross Site Scripting (Reflected)

First, it submits a ‘safe’ value and checks the response for any instances of this value. Then, for any locations where this value was found in, the program attempts a series of attacks aimed at that location to confirm whether a XSS attack is possible.

-Cross Site Scripted (Persistent)

It begins by submitting a ‘safe’ value much like the reflected variant, but it then spiders the whole application to find any locations where the value might be found and tries to attack them.

-Code Injection

Submits PHP and ASP attack strings as URL parameter values and then examines the response.

-Command Injection

Submits \*NIX and Windows OS commands as URL parameter values that would produce a specific output should the application be vulnerable. If this output isn’t found in the response, then it tries a more complex approach that involves blind injection by submitting sleep instructions and timing the responses.

-Directory Browsing

Tries to gain access to a directory listing by examining the response for patterns commonly used with web server software such as Apache or IIS.

-External Redirect

Tries to force a redirect by sending specific parameter values to the URL, and then checks the headers and response to check whether a redirect occurred.

-Format String Error

Sends strings of input text that compiled C code would interpret as formatting parameters, and then looks for any anomalies.

-Parameter Tampering

Submits requests with parameter values known to cause errors, and compares the response against patterns found in common errors in Java, Microsoft VBScript, PHP and others.

-Path Traversal

Tries to access files and directories outside of the web document root using combinations of pathname prefixes and names of local files typically found in Windows or \*NIX systems.

-Remote File Include

Passes an external URL as a parameter value for the current URL and checks whether this causes the title of the page to change.

-Server Side Include

Detects the OS the server is running on, and then sends a HTML SSI directive. The response is then analyzed to check whether it matches with a pattern indicating the SSI (Server Side Include) was successful.

-SQL Injection

Attacks URL parameters and form parameters with valid and invalid SQL syntax, using diverse SQL injection techniques such as error, Boolean or Union based injection.

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.

The headless instance of OWASP ZAP will be closed via Python as soon as the report generation ends, or at any point that an error occurs and the execution of the program is interrupted.

## 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 formatted according to the 800-115 NIST guideline[11], using a penetration test report from Offensive Security as a template[12]. The resulting output 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[13].

GitBook will take into account any variables found within the *book.json* file when generating the pdf file[[12]](#footnote-12). 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.

The report will begin with a short introduction mentioning what URLs were scanned and what version of OWASP ZAP was used for the scan. It will follow with a summary of results, listing every vulnerability that was detected at least once, along with a brief description of each vulnerability and a list of all URLS where they were found. The next section is an attack narrative, describing in some level of detail the payloads that were applied during the active scan, followed by a brief risk rating section assigning risk levels to the vulnerabilities. The report will end with a set of recommendations for each vulnerability and a brief description of the methodology used to automate the scan and generate the report.

1. https://nmap.org/ [↑](#footnote-ref-1)
2. https://pypi.python.org/pypi/python-nmap [↑](#footnote-ref-2)
3. http://searchnetworking.techtarget.com/definition/CIDR [↑](#footnote-ref-3)
4. https://nmap.org/book/man-host-discovery.html [↑](#footnote-ref-4)
5. https://nmap.org/book/man-port-scanning-techniques.html [↑](#footnote-ref-5)
6. https://github.com/zaproxy/zaproxy/wiki/ApiPython [↑](#footnote-ref-6)
7. https://github.com/zaproxy/zap-core-help/wiki/HelpAddonsPscanrulesPscanrules [↑](#footnote-ref-7)
8. https://www.keycdn.com/support/what-is-mime-sniffing/ [↑](#footnote-ref-8)
9. https://www.keycdn.com/blog/x-frame-options/ [↑](#footnote-ref-9)
10. https://github.com/zaproxy/zap-core-help/wiki/HelpAddonsAscanrulesAscanrules [↑](#footnote-ref-10)
11. https://www.acunetix.com/websitesecurity/crlf-injection/ [↑](#footnote-ref-11)
12. https://gitbookio.gitbooks.io/documentation/content/format/templating.html [↑](#footnote-ref-12)