



Improving Recon Scans and Optimizing Task Handling

for Google Summer of Code 25'

Achintya Jai

IIT Madras (2023-2027)

Possible project mentors:

[Sam Stepanyan](#), [Arkidii Yakovets](#), [Ali Razmjoo](#)

Abstract

This project aims to improve the **reconnaissance scan** features of [Nettacker](#) while also addressing the issues caused by **high parallelization** and the ideas mentioned in the [idea list for Nettacker](#). My goal is to improve Nettacker's features so as to facilitate its mass adoption in the industry and for personal projects alike.

Table of contents

Abstract	2
Table of contents	2
Personal Information	3
Programming background	4
General:	4
Cybersecurity experience:	4
Python experience	5
Git and GitHub experience	5
Platform details	5
Motivation behind participating in GSoC 25'	6
Contributions to Nettacker	6
Merged	7
Closed (without merge)	7
Open	7
Issues opened	8
Proposal	8
Improving Reconnaissance Scans	9
Service Logging	10
Improving Regexes	11
Problems	13
Version Scanning and Better Probes	14
Why commit @86923c is an improvement	18
Remaining work	19
Differentiating between TCP and UDP	20
Remaining Work	23
Porting from SQLite to APSW	24

Remaining Work	25
Porting to Huey	25
Remaining Work	27
Adding New Output types	28
Integrating Nettare with DefectDojo	28
SARIF output format	30
Web Server Improvements	31
Creating input fields for custom directory enumeration wordlist	31
Creating input fields for username and password files	32
Integrating Huey with the Web Server	32
Remaining Work	33
Improving Test coverage	34
Timeline	34
Application Review Period	34
Community Bonding Period	35
GSoC Period	35
Phase 1 (week 0 - 3)	35
Phase 2 (week 3 - 6)	36
Phase 3 (week 6 - 9)	36
Phase 4 (week 9 - 12)	37
Post GSoC Period	38
Time commitments	39
Acknowledgements	39

Personal Information

- **Name:** Achintya Jai
- **University:** Indian Institute of Technology, Madras
- **Degree:** B.tech
- **Residence:** Madras/Chennai, India
- **Time Zone:** Indian Standard Time (UTC + 5:30)
- **GitHub Account:** [GitHub](#)
- **Email:**
 1. ch23b002@smail.iitm.ac.in (Institute mail)
 2. helloitsme0218@gmail.com (GitHub linked + personal mail)

- **Language spoken:** English
- **Personal website:** <https://purge12.github.io>
- **Slack username:** purge12

I am **Achintya**, a 2nd year undergraduate student pursuing a B-tech degree at the Indian Institute of Technology, Madras (IITM), which is currently ranked the best engineering institute in the country (a country producing a million or so engineers every year!).

I have a lot of hobbies, which I engage in other than writing code. For example, I like making sketches (you can find some [here](#)), playing my guitar and debating. I spend most of my time with my [solar car racing team](#) where I work as an electrical engineer.

Programming background

General:

I started programming back in 11th grade when my school made it compulsory for all of us to learn python and mysql. During the covid years I used to sit every morning and watch [Mosh Hamedani's lecture](#) on programming with python. Hence when I came to IITM, I was already versed with python to some extent.

There are two fields that I actively take part in now, **AI** (building applications) and **Cybersec**. The journey to each of them has been rough but totally worth it.

Cybersecurity experience:

I started giving CTFs in my first semester, and documented the writeups in [this GitHub](#) repo. I also worked in the CS lab here implementing EDR bypass techniques and demonstrating those to the professors via POCs for the [BEEP](#) and [BloodyAlchemy](#) malwares that I made during my time there. I came 5th in a state level bug-bounty program + CTF in 2025 which was my introduction to **bug bounties**.

I made the reverse engineering [questions and write ups](#) for my college CTF, hosted a lecture on the [basics of reverse engineering](#) for juniors and took a certification course on [malware analysis from coursera](#).

Currently, I am working towards a Wi-Fi deauth and evil-twin attack project at IITM which is under the proposal phase. This network security project will involve me leading a team of junior students in its development. This [GitHub repo](#) explains it better.

Python experience

My major tech stack is python, and I have learnt to use it majorly through competitions and personal projects. I have used python and flask for the [GenAI hackathon](#) organised by Google that I **won**.

I have also used python for the international [LLM Agents hackathon](#) organised by Berkeley RDI, to develop TerminAI, and I **won** that too.

I maintain these on my GitHub which are linked below:

1. **TerminAI**: https://github.com/pUrGe12/TerminAI_V2
2. **Cromax**: https://github.com/AadiSrivastava05/genaihack_croma

Git and GitHub experience

I have been using git for over a year now and I have become familiar with the workflows involved in git and GitHub. I have learnt about rebasing, **atomic pull requests** etc. by making a lot of mistakes in my PRs while contributing to OWASP. I have also learnt about the best practices in GitHub (like creating a new branch for every contribution). Prior to my contributions to OWASP I worked mostly on solo projects which helped me get familiar with the command line interface.

Platform details

Operating system (primary): Ubuntu 22.04

Operating system (secondary): Windows 11 dual booted

Development setup: Sublime text and gedit

Virtual machines: Kali, REMnux, Win11 on virtualbox

Computer hardware: intel i5, 10th Gen with 8GB RAM

Motivation behind participating in GSoC 25'

My primary motivation behind applying to OWASP is based on my interest in cybersecurity. I have followed OWASP's top 10 since 2023 (that's when I joined college and developed a heavy interest in the subject).

I have used Nettacker for CTFs and I enjoyed working with this tool because it matched my tech stack and hence, I would try to tinker with it to match my use-cases.

I had realised that there are some things that would **improve Nettacker**, that I can help develop. These include:

- Service logging functionality
- Version scanning functionality
- TCP/UDP service detection

I tried to implement some of them through my PRs (will be mentioned below) but there remains a lot to be done. I wish to utilize the opportunity that GSoC presents to bring these **ideas to life in Nettacker** while also working on the ones **mentioned by the organisation**.

Beyond technical exposure, the experience I gained in the past few months has **deepened my appreciation** for the open-source community. I find it astonishing and beautiful that all these projects were built for the sole purpose of development, love for the project and to help the masses.

This is something I want to be a part of.

Contributions to Nettacker

I started contributing to the OWASP community and Nettacker in February 2025. Below is a list of my contributions that are relevant to my current project.

Merged

- [Nettacker PR#1008](#): Fixed the admin_scan output to include the hit URLs
- [Nettacker PR#1012](#): Updated SSH regex for more matches
- [Nettacker PR#1019](#): Added base path for directory enumeration
- [Nettacker PR#1026](#): Added functionality to allow users to enter their own wordlists for scan modules
- [Nettacker PR#1020](#): Setting default event loop policy for asyncio (suggested by [@securestep9](#))
- [Nettacker PR#0136](#): A small patch for en.yaml for clean exit
- [Nettacker PR#1046](#): Added regex for **AMQP** protocol detection
- [Nettacker PR#1039](#): Creating test cases for wordlists and cleaned the wordlist files (removed duplicates)
- [Nettacker PR#1047](#): Bug Fixing my implemented code for **PR#1026**
- [Nettacker PR#1056](#): Fixing database issues, changing drivers for MySQL and PostgreSQL.
- [DefectDojo PR#11761](#): Small documentation fix and link correction

Closed (without merge)

- [Nettacker PR#1016](#): My fix for parallel execution was to change the default delay value, but [@securestep9](#) gave a much better idea (implemented through **PR#1020**)
- [Nettacker PR#1001](#): Tried to fix MySQL functionality, there were some more issues I had overlooked.
- [Nettacker PR#1010](#): Initial fix for admin_scan output logging but it was interfering with the core functionality, hence implemented **PR#1008** above.
- [DefectDojo PR#11805](#): Tried to fix the [anchore engine](#) parser in DefectDojo. It was running into some CI issues that I wasn't able to resolve. Maintainers implemented this themselves.

Open

- [Nettacker PR#1052](#) - Logging all matched services in port scans for **issue #1023**
- [Nettacker PR#1043](#) - Refactoring [ip.py](#) to return proper boolean values
- [Nettacker PR#1042](#) - Created tests for [die.py](#)

- [Nettacker PR#1041](#) - Profiles cleanup as mentioned in **issue #1038**

Issues opened

- [Nettacker #1038](#) - Profile cleanup required
- [Nettacker #1023](#) - Service scanning doesn't reveal versions
- [Nettacker #999](#) - MySQL connections fail due to SQLite specific config in DbConfig. Should be closed by [PR#1056](#).
- [Nest #829](#) - Show issue status on the contribute page or remove successfully closed issues

Proposal

Following from the discussions in [#project-nettacker](#), my talks with [@securestep9](#) and the [idea list](#) mentioned in OWASP's official site, below are my proposed improvements for Nettacker **arranged** according to relative priority.

- **Improving host scanning**

- High priority**

- Support for service logging - **Code for reference:** [PR#1052](#)
 - Adding version detection - **Code for reference:** [changelog](#)
 - Differentiating TCP vs UDP - **Code for reference:** [@48864ea](#)
- **Fixing multi-threading issues**
 - Porting to APSW for a SQLite - **Code for reference:** [@261795f](#)
 - Using Huey as a task queue - **Code for reference:** [@7aba750](#)
- **Improving output types**
 - DefectDojo output integration - **Code for reference:** [@81cc8a9](#)
 - SARIF output report format - **Code for reference:** [changelog](#)
- **Web improvements**
 - Integrating Huey with flask - **Code for reference:** [@c305812](#)
 - Adding a field for enumeration, passwords and usernames wordlists to override defaults. - **Code for reference:** [changelog](#) and [@4469c41](#)

I will also work on implementing test cases and a detailed outlook on that is presented in the timeline.

Improving Reconnaissance Scans

The following areas need improvement in Netrunner:

1. The services that are found by probing using **tcp_connect_send_and_receive** are not logged properly. I have attached a screenshot of how the text_table looks after running a port scan.

```
conditions: http: - 'Content-Type: ' - HTTP/1.1 400
[2025-03-31 14:34:22][+++] process-0|port_scan|en.wikipedia.org|module-thread 1/1
success_condition (s):
conditions: http: - 'Content-Type: ' - HTTP/1.1 400
[2025-03-31 14:34:23][+++] process-0|port_scan|en.wikipedia.org|module-thread 1/1
success_condition (s):
conditions: open_port: - '2000'
[2025-03-31 14:34:24][+++] process-0|port_scan|en.wikipedia.org|module-thread 1/1
success_condition (s):
conditions: open_port: - '5060'
[2025-03-31 14:34:27][+] Removing old database record for selected targets and mo
[2025-03-31 14:34:27][+] imported 1 targets in 1 process(es).
[2025-03-31 14:34:27][+] building graph ...
[2025-03-31 14:34:27][+] finish building graph!
```

date	target	module_name	port	logs
2025-03-31 14:34:22.328518	en.wikipedia.org	port_scan	80	Detected
2025-03-31 14:34:22.339771	en.wikipedia.org	port_scan	443	Detected
2025-03-31 14:34:23.532118	en.wikipedia.org	port_scan	2000	Detected
2025-03-31 14:34:23.912935	en.wikipedia.org	port_scan	5060	Detected

2. Netrunner does not provide the **versions** of the services being found on open ports.
 - The regexes present in **port.yaml** are only to match the service and not the versions. This is not ideal because hosts might be running outdated versions of some services that might be exploitable.

- With this additional feature, Netttacker will allow users to identify vulnerable and outdated services, which is good.

3. The detected services are not differentiated between **TCP** and **UDP**.

Differentiating between TCP and UDP is important because they have different attack vectors. By correctly identifying TCP and UDP services, Netttacker will provide a more comprehensive view of potential security risks

The following will cover these in detail.

Service Logging

I raised [PR #1052](#) to log all services that are detected after regex matching. This is what the output logs look like:

```
2025-03-31 15:05:09][+++] process-0|port_scan|127.0.0.1|module-thread 1/1|request
success_condition (s):
ipp - ['Server: CUPS/2.4 IPP/2.1\r\n']
2025-03-31 15:05:09][+++] process-0|port_scan|127.0.0.1|module-thread 1/1|request
success_condition (s):
mysql - ['caching_sha2_password\x00']
ipp - ['Server: CUPS/2.4 IPP/2.1\r\n']
2025-03-31 15:05:09][+++] process-0|port_scan|127.0.0.1|module-thread 1/1|request
success_condition (s):
open_port - ['5432']
mysql - ['caching_sha2_password\x00']
ipp - ['Server: CUPS/2.4 IPP/2.1\r\n']
2025-03-31 15:05:10][+] Removing old database record for selected targets and mo
2025-03-31 15:05:10][+] imported 1 targets in 1 process(es).
2025-03-31 15:05:10][+] building graph ...
2025-03-31 15:05:10][+] finish building graph!
```

date	target	module_name	port	logs
2025-03-31 15:05:09.430760	127.0.0.1	port_scan	631	ipp - ['Server: CUPS/2.4 IPP/2.1\r\n']
2025-03-31 15:05:09.636973	127.0.0.1	port_scan	3306	mysql - ['caching_sh a2_password\x00']
2025-03-31 15:05:09.707598	127.0.0.1	port_scan	5432	open_port - ['5432']

This PR is still open and once merged I will proceed with improving the regex for matching more services.

Improving Regexes

I have updated the regex for **SSH** through [PR#1012](#) and added **AMQP** detection regex through [PR#1046](#). The following regexes can further be added/updated:

- **MySQL** (update)

The current regex is: “is not allowed to connect to this MySQL server” which can be updated to “**\x08**\\d+\\.\\.\\d+\\.\\.\\d+| **caching_sha2_password**\\x00|is not allowed to connect to this MySQL server”.

This works because **\x08** is the MySQL handshake byte which will always be sent during arbitrary probing and **caching_sha2_password** is the default authentication method in modern MySQL systems.

```
success_condition (s):
mysql - ['caching_sha2_password\x00']
[2025-03-30 12:51:16][+] Removing old database record for selected targets and modules.
[2025-03-30 12:51:16][+] imported 1 targets in 1 process(es).
[2025-03-30 12:51:16][+] building graph ...
[2025-03-30 12:51:16][+] finish building graph!
```

date	target	module_name	port	logs
2025-03-30 12:51:16.263133	127.0.0.1	port_scan	3306	mysql - ['caching_sh a2_password\x00']

- **Elasticsearch** (new)

This is a service that listens on port 9200 for HTTP requests. Since this is a non-standard port, it must be set properly in the **/etc/services** file. Additionally, the responses given are HTTP based so the entry for **elasticsearch** must be done **above that of http** in port.yaml.

This is the regex: “**X-elastic-product**:\\s***Elasticsearch**\\\"reason\\\":\\\"text is empty
\\(possibly HTTP/\\d+\\.\\.\\d+\\)\\”

```

elasticsearch - ['X-elastic-product: Elasticsearch', '"reason":"text is empty (possibly HTTP/0.9)"]'
[2025-03-30 14:48:31][+] Removing old database record for selected targets and modules.
[2025-03-30 14:48:31][+] imported 1 targets in 1 process(es).
[2025-03-30 14:48:31][+] building graph ...
[2025-03-30 14:48:32][+] finish building graph!

```

date	target	module_name	port	logs
2025-03-30 14:48:31.826322	127.0.0.1	port_scan	9200	elasticsearch - ['X-elastic-product: Elasticsearch', '"reason":"text is empty (possibly HTTP/0.9)"]'

- **IPP** (new)

The Common UNIX Printing System (CUPS) relies on the Internet Printing Protocol (IPP) which is HTTP based. With a small modification to the arbitrary probe sent by `tcp_connect_send_and_receive` (replacing “ABC” with “GET / HTTP/1.1\x00”), we can match multiple HTTP based protocols, IPP being one of them.

The regex is: “**Server: CUPS/\d+\.\.\d+ IPP/\d+\.\.\d+
r?\n**”

The response given by IPP is HTTP formatted, and hence this must be **placed above http** in port.yaml otherwise the http match will happen first.

```

ipp - ['Server: CUPS/2.4 IPP/2.1\r\n']
[2025-03-30 15:47:29][+] Removing old database record for selected targets and modules.
[2025-03-30 15:47:29][+] imported 1 targets in 1 process(es).
[2025-03-30 15:47:29][+] building graph ...
[2025-03-30 15:47:29][+] finish building graph!

```

date	target	module_name	port	logs
2025-03-30 15:47:29.003742	127.0.0.1	port_scan	631	ipp - ['Server: CUPS/2.4 IPP/2.1\r\n']

This is port.yaml structure after adding elasticsearch and IPP protocols,

```

elasticsearch:
  regex: "X-elastic-product:\s*Elasticsearch|\"reason\": \"text is empty \\\(possibly HTTP/\\d+\\.\\.\\d+\\)\\\""

```

```

reverse: false

ipp:
  regex: "Server: CUPS/\\d+\\.\\.\\d+ IPP/\\d+\\.\\.\\d+\\r?\\n"
  reverse: false

http:
  regex:
"HTTPStatus.BAD_REQUEST|HTTP\\/[\\d.]+\\s+[\\d]+|Server: |Content-Length:
\\d+|Content-Type: |Access-Control-Request-Headers: |Forwarded:
|Proxy-Authorization: |User-Agent: |X-Forwarded-Host: |Content-MD5:
|Access-Control-Request-Method: |Accept-Language: "
  reverse: false

```

Problems

There are two issues now:

1. Probing using **GET / HTTP/1.1** is not ideal because HTTP services now get **incomplete requests**.

Services like elasticsearch start returning **error packets** because of authentication failures, and give no useful information about themselves. The regex therefore doesn't work.

```

this is the response: b'HTTP/1.1 401 Unauthorized\r\nWWW-Authenticate: Basic realm="security" ch
ent-type: application/json;charset=utf-8\r\ncontent-length: 459\r\n\r\n{"error":{"root_cause":[{"
,"header":{"WWW-Authenticate":["Basic realm=\\\"security\\\" charset=\\\"UTF-8\\\"";"Bearer realm=\\
tials for REST request [/]","header":{"WWW-Authenticate":["Basic realm=\\\"security\\\" charset=\\
[2025-03-30 16:16:30][+++] process-0|port_scan|127.0.0.1|module-thread 1/1|request-thread 0/1|
success_condition (s):
http - ['HTTP/1.1 401']
[2025-03-30 16:16:30][+] Removing old database record for selected targets and modules.
[2025-03-30 16:16:30][+] imported 1 targets in 1 process(es).
[2025-03-30 16:16:30][+] building graph ...
[2025-03-30 16:16:30][+] finish building graph!

```

date	target	module_name	port	logs
2025-03-30 16:16:30.427406	127.0.0.1	port_scan	9200	http - ['HTTP/1.1 401']

2. Some services don't return anything when probed with arbitrary bytes

These services are usually UDP based or they are FTP services like **SMB, RDP** etc. that require authentication before they respond. Additionally there are protocols like **FTP** that leak their banners without needing probes.

These issues will be addressed by creating a different function that will adjust its probes based on the port it's probing.

Version Scanning and Better Probes

A basic execution of version scanning can be found over [this changelog](#) in my local fork.

Commit [@86923c](#) further improves this execution and this is what I will be working with. (It is detailed in [this section](#))

This relies on a YAML file that is formed using the [nmap-service-probes.txt](#) file. The generated YAML file will be structured as follows:

```
service_logger:
- value: 43
  probe:
    - Probe TCP GenericLines q|\r\n\r\n|
  regex: *id001
- value: 79
  probe:
    - Probe TCP GenericLines q|\r\n\r\n|
    - Probe TCP GetRequest q|GET / HTTP/1.0\r\n\r\n|
    - Probe TCP Help q|HELP\r\n|
  regex: *id002
```

Parser - [Gdrive link](#) (used to generate the YAML file)

YAML file - [Gdrive link](#)

(Note, the YAML file has been “patched” after parsing to ensure referencing and anchoring works, and hence running the parser will give a slightly off file).

I have ignored the probe names in my executions but they can possibly be used for **memoization** and making traversal faster.

The probes are formatted by listing the protocol first followed by the actual bytes to be sent. Something similar is the case for the regexes. [This official documentation](#) for the file format of nmap-service-probes explains the meaning behind **other flags** that are used in the YAML file.

A summary and some description of the changes made in the above changelog are:

- Updating **arg_parser.py** to allow users to set a **-sV flag** to enable version scanning.
- Inside **port.yaml** another method is specified for version scanning. This is called **tcp_version_scan**.

```
payloads:  
- library: socket  
  steps:  
    - method: tcp_connect_send_and_receive  
      method_version: tcp_version_scan
```

- The **base.py** file is modified to implement version scanning for all detected ports. (**getattr** is inside the if condition to avoid unnecessary calls).

This piece of code is written inside the **engine.run()** function. Currently **tcp_version_scan** just prints any logs it finds and returns nothing.

```
version_response = ""  
for _i in range(options["retries"]):  
    try:  
        response = action(**sub_step)  
        if (version_scan) and ("port_scan" in  
options.get("selected_modules")) and (response is not None):  
            version_action = getattr(self.library(),  
backup_method_version)  
            version_action(**response)  
            break  
    except Exception:
```

```
response = []
```

- The **tcp_version_scan** function is defined inside `socket.py`. This function takes the result of the **tcp_connect_send_and_receive** function and returns the versions for the ports.

It does the following:

- a. Tries to send a **custom payload** depending on the port detected and it's already **parallelized** (but not truly, this will be explained in the [remaining work](#) section) since each thread executes this independently.
 - b. If a **response is received** and matched, it returns that.
 - c. If **no response is received** or no response is matched, it sends a **null probe** and tries to match that and return.
 - d. If none matched, it returns an **empty string**.
- It uses **port_to_probes_and_matches** which is defined inside **common.py** to get the probes and regexes. This basically parses the raw YAML file and returns a structured output.

This is the base of that function.

```
def port_to_probes_and_matches(port_number, data):
    results = {"probes": [], "matches": []}

    for entry in data.get("service_logger", []):
        if int(entry["value"]) == port_number:
            results["probes"] = entry.get("probe", [])
            matches_list = entry.get("regex", [])
            break
        else:
            # It doesn't have probes for ports like 3306 (mysql)
            matches_list = []
```

The **matches_list** is further processed such that the final output of the function is of the following structured format.


```
{
  "probes": [probes],
  "matches": [
    {"match_1": {"service": "",
                  "regex": "",
                  "flag_1": "",
                  "flag_2": "" ...}},
    {"match_2": {"service": "",
                  "regex": "",
                  "flag_1": "",
                  "flag_2": "" ...}}]]}

```

There are 6 flags defined in the nmap docs which will be used here. The doc has been [linked here](#).

A sample code for **port_to_probes_and_matches** can be found over [@86923c](#) in my local fork.

- The resulting **[probes]** list is passed to the **extract_probes** function which returns the raw bytes for the probes.

This is a one-liner in python:

```
return [bytes(probe.split("q|")[1][:-1], encoding="utf-8") if
        "q|" in probe
        else bytes(probe.split("q/")[1][1:], encoding="utf-8") for
        probe in probes_list]

```

- The probes are sent over the socket and a response is awaited. If a response is received, it is sent to the **match_regex** function along with the **regex_values_dict_list**.

The implementation for the **match_regex** function can be found over [@17dceaa](#) in my local fork.

```
try:
    for probe in raw_probes:
        socket_connection.send(bytes(probe,
encoding="utf-8"))
        response = socket_connection.recv(1024 *
1024 * 10)

```

```

        matches = match_regex(response,
regex_values_dict_list)
        socket_connection.close()
    except Exception as e:
        print("exception hit: {}".format(e))
        matches = b""

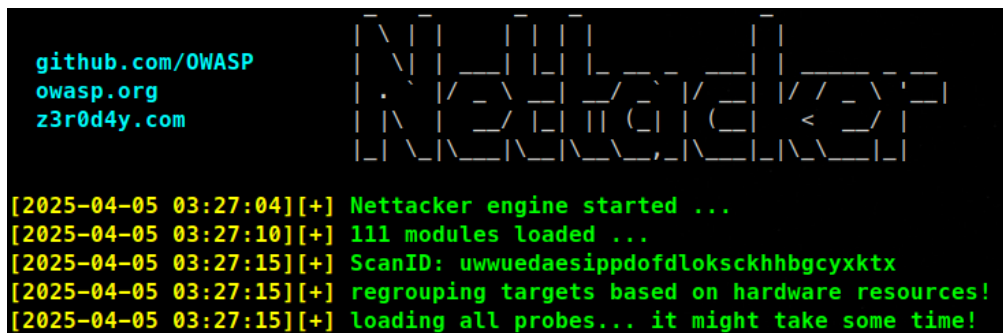
```

Why commit [@86923c](#) is an improvement

The changelog mentioned ([this one](#)) is not parallelized in the true sense because **each thread tries to open the yaml file** by calling `port_to_probes_and_matches`.

This results in a lot of CPU usage and unnecessary waste of resources and time. With this commit, I have opened the YAML file in and **saved it in memory** right at the start of the execution. The contents are then being passed around so that it reaches `port_to_probes_and_matches` where it will be acted upon. The rest of the code remains the same.

This is done by reading the file inside `app.py` right after it creates a `scan_id`. This allows for a nice message to be shown to the user as well!



```

github.com/OWASP
owasp.org
z3r0d4y.com

[2025-04-05 03:27:04][+] Netstacker engine started ...
[2025-04-05 03:27:10][+] 111 modules loaded ...
[2025-04-05 03:27:15][+] ScanID: uwwuedaesippdofdloksckhhbgcyxktx
[2025-04-05 03:27:15][+] regrouping targets based on hardware resources!
[2025-04-05 03:27:15][+] loading all probes... it might take some time!

```

I have still included the **changelog** because it contains definitions of other valuable functions which are used directly in commit [@86923c](#).

These are the outputs that were printed during execution (the debug statements are still present in the changelog and referenced commit):

This is demonstrating that for a specific port, obtaining the **probes_list** and a **response from custom probes** is working well. The caveat is the **matches_list** which is shown to be empty.

The reason for this will be coming up under the remaining work point number 3.

```
inside this if: port_number is: 443
probes_list: ['Probe TCP SSLSessionReq q|\\x16\\x03\\0\\0S\\x01\\0\\00\\x03\\0?G\\xd7\\
0n\\0\\0(\\0\\x16\\0\\x13\\0\\x0a\\0f\\0\\x05\\0\\x04\\0e\\0d\\0c\\0b\\0a\\0`\\0\\x15\\
\\0\\0\\x69\\x01\\0\\0\\x65\\x03\\x03U\\x1c\\xa7\\xe4random1random2random3random4\\0\\
\\x06\\x01\\x06\\x03\\x06\\x02\\x02\\x01\\x02\\x03\\x02\\x02\\x03\\x01\\x03\\x03\\x03\\
TCP SSLv23SessionReq q|\\x80\\x9e\\x01\\x03\\x01\\x00u\\x00\\x00\\x00 \\x00\\x00f\\x0
03\\x00\\x002\\x00\\x00\\x00\\x00\\x1b\\x00\\x00\\x1a\\x00\\x00\\x19\\x00\\x00\\x18\\
\\x00\\x00\\n\\x00\\x00\\t\\x00\\x00\\x08\\x00\\x00\\x06\\x00\\x00\\x05\\x00\\x00\\x04\\
0\\x00\\x02\\x00\\x00\\x01\\xe4i<+\\xf6\\xd6\\x9b\\xbb\\xd3\\x81\\x9f\\xbf\\x15\\xc1@\\
\\0\\0\\0\\0\\0\\0\\0\\0\\0|', 'Probe TCP OpenVPN q|\\0\\x0e87\\xa5&\\x08\\xa2\\x1b\\xa
UDP DTLSv2SessionReq q|\\x16\\xfe\\xff\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x3
40\\x1e\\x8a\\xc8\\x22\\xa0\\xa0\\x18\\xff\\x93\\x08\\xca\\xac\\x0a\\x64\\x2f\\xc9\\x2
\\r\\x89\\xc1\\x9c\\x1c*\\xff\\xfc\\xf1Q999\\x00|']
matches_list: [[]]
This is the regex_values_dict_list: [None]
got resonse from custom probing: b'HTTP/1.1 400 Bad Request\r\nServer: cloudflare\r\nD
e\r\nCF-RAY: -\r\n\r\n<html>\r\n<head><title>400 Bad Request</title></head>\r\n<body>\r
n'
```

Remaining work

1. **Parallelize probes** across threads

The basic execution described above leads to **each thread** having to go through the entire probe list. This is an unnecessary operation and it will be much better if a thread picks up a probe and executes only that.

2. Integrate the output with **Nettacker's logging** system

Self-evident, as I am only printing the found results and the functions do not return anything. The output logs should integrate the output of **tcp_version_scan**.

3. Handle **timeouts** and **empty matches_list**.

The empty matches_list is not an error, because according to the YAML file, the regex for port 443 is referencing ***id021** which is anchored at port 80, whose regex is an empty list.

The same run shows that for services like SIP at port 5060, we get the matches list and probes list properly (so it works).

```
inside this if: port_number is: 5060
probes_list: ['Probe TCP GetRequest q|GET / HTTP/1.0\\r\\n\\r\\n|', 'Probe TCP SIPOptions q
: <sip:nm2@nm2>\\r\\nCall-ID: 50000\\r\\nCSeq: 42 OPTIONS\\r\\nMax-Forwards: 70\\r\\nContent
ions q|OPTIONS sip:nm SIP/2.0\\r\\nVia: SIP/2.0/UDP nm;branch=foo;rport\\r\\nFrom: <sip:nm@
\\r\\nContent-Length: 0\\r\\nContact: <sip:nm@nm>\\r\\nAccept: application/sdp\\r\\n\\r\\n|
matches_list: ['match sip m|^SIP/2\\.0 404 Not Found\\r\\n(?:[\\r\\n]+\\r\\n)*?User-Agent:
m|^SIP/2\\.0 200 OK\\r\\n(?:[\\r\\n]+\\r\\n)*?User-Agent: SAGEM / 3202\\.3 / 2601EC \\r\\
[\\r\\n]+\\r\\n)*?Server: sipXecs/(\\w._-|+) sipXecs/sipXproxy \\(Linux\\)\\r\\n|s p/SIPf
und\\r\\n(?:[\\r\\n]+\\r\\n)*?User-Agent: AVM (FRITZ!Box Fon WLAN [\\w._-|+)(?:Annex A )
atch sip m|^SIP/2\\.0 200 OK\\r\\n(?:[\\r\\n]+\\r\\n)*?Server: NetSapiens SiPBx 1-1205c\\r
oes Not Exist\\r\\nFrom: <sip:nm@nm>;tag=root\\r\\nTo: <sip:nm2@nm2>;tag=0-\\w+-\\w+-\\w+-\\
+;branch=foo\\r\\nContent-Length: 0\\r\\n\\r\\n$| p/Sony PCS-TL50 videoconferencing SIP/ cp
nm;branch=foo;rport\\r\\nFrom: <sip:nm@nm>;tag=root\\r\\nCall-ID: 50000\\r\\nTo: <sip:nm2@n
/a:ekiga:ekiga:3.2.7/', 'match sip m|^SIP/2\\.0 403 Forbidden\\r\\n(?:[\\r\\n]+\\r\\n)*?Fr
```

Hence, empty matches_list need to be handled by either **skipping them** (reduces unnecessary calls) or **logging the raw response** (will be messy, but the intermediate users will understand).

Most of the threads will timeout and raise an exception. This requires explicit handling as well. It's not a big problem though.

Differentiating between TCP and UDP

Nettacker currently analyses only TCP ports. This is not ideal because services like **DNS**, **NetBIOS Name Service** etc. will not be detected by Nettacker. This requires probing specifically for UDP ports and logging them neatly.

A sample implementation for detection of UDP services can be found [@2dde609](#) and [this changelog](#) in my local fork.

- This is an extension of the previous case, where the same YAML file will be utilized to probe for UDP services.
- The probes in the YAML file are formatted to include the protocol (either TCP or UDP), so for ports that are pre-assigned to UDP by default, these UDP probes will be used.

```
- value: 53
  probe:
    - Probe UDP DNSVersionBindReq
q|\0\06\01\0\0\01\0\0\0\0\0\07version\04bind\0\0\10\0\03|
    - Probe TCP DNSVersionBindReqTCP
q|\0\1E\0\06\01\0\0\01\0\0\0\0\0\07version\04bind\0\0\10\0\03|
  regex: []
```

- The **regex list** is kept empty because UDP services don't reply in general, and if they do, it's usually an answer to a request.

Note that this YAML file is being used here because UDP drops **arbitrary** probes and hence we need good custom probes. A dropped connection results in the port being either closed or filtered (no definitive answer).

A brief summary for the above commit can be found below:

- A new flag called **-sU** is defined. This is done to not add the latency introduced by a huge file read in Netstacker during normal operation.
- Inside **app.py**, the YAML file is loaded and passed to the **arguments** (as was mentioned in the version logging case). This time a method is defined to make this process easy to follow. This is called inside **run()**.

```
def load_yaml(self):
    import yaml
    log.info(_("loading_probes"))
    with open(Config.path.probes_file) as stream:
        data = yaml.safe_load(stream)
    self.arguments.version_probes_raw_data = data
    log.info(_("loaded_probes"))
```

A function **udp_scan()** will be defined inside **socket.py**.

- It uses the UDP probes from the YAML file, and queries the server. If a response is received, it **confirms that the port is open**.
- If no response is received, then the port **might be closed** (might be closed and **not will be closed**, because it can be filtered via a firewall or it might be configured to drop bad probes)

This function calls the **port_to_probes_and_matches** function defined in `common.py` and then passes that to the **extract_UDP_probes** function to get UDP probes from the big list.

```
def udp_scan(self, host, port, timeout):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_probes_list =
extract_UDP_probes(port_to_probes_and_matches(port)["probes"])
    for probe in udp_probes_list:
        try:
            sock.sendto(probe, (target_host, target_port))
            response, _ = sock.recvfrom()
            sock.close()
        except Exception:
            try:
                sock.close()
                response = b""
            except Exception:
                response = b""
```

- This function is referenced inside **engine.run**, where it is called **if port_scan** was requested by the user and has set the **udp_scan** flag.

It is better to run this entire loop after the TCP scans are done so that the threads which exit the TCP scanning routine, immediately pick this up.

```
for _i in range(options["retries"]):
    try:
        if "port_scan" in
```

```

options.get("selected_modules") and udp_scan:
    udp_port_scanner = getattr(self.library(),
"udp_scan")

    sub_step["data"] = probes_data
    udp_port_scanner(**sub_step)
    # This is directly printing
except Exception:
    response = []

```

Remaining Work

The remaining work is the same as those for service logging. It's [referenced here](#).

Fixing Multi-threading issues

The issues are as follows:

- The current implementation of **threading** and **multiprocessing** continuously queries database variables, as noted by [@Ali Razmjoo](#) in Issue [#595](#), leading to increased CPU load
- [@Ali Razmjoo](#) also raised an issue [#609](#) which talks about unnecessary I/O operations, memory usage and CPU consumptions.

Given these, and the suggestion [#1037](#) by [@securestep9](#), I have worked on the following systems to try and solve the above issues:

1. [APSW](#) wrapper for **SQLite** C as a replacement for SQLite (while keeping SQLAlchemy for PostgreSQL and MySQL)
2. [Huey](#) as **task queues** for replacing threading and multiprocessing libraries (with a SQLite message broker and the result backend)

Porting from SQLite to APSW

A basic implementation for switching to APSW from SQLAlchemy can be found over [@261795f](#) in my local fork.

A brief summary of the changes made are:

- Created raw SQL queries for DB operations in **db.py**

```

session = create_connection()
return (
    session.query(HostsLog)
    .filter(
        HostsLog.target == target,
        HostsLog.module_name == module_name,
        HostsLog.scan_unique_id == scan_id,

connection, cursor = create_connection()

try:
    cursor.execute(
        """
        SELECT json_event FROM scan_events
        WHERE target = ? AND module_name = ? and scan_unique_id = ?
        """,
        (
            target, module_name, scan_id
        ),
    )
)

```

- Bypassed the **create_connection** function assuming only SQLite databases can be used.
- Updated **module.py** and **graph.py** to support the updated functions.

I tested APSW against SQLAlchemy by running a vulnerability profile scan against a host with 1000 threads.

A summary for the execution results are:

1. Peak RAM usage is **2x lesser** in case of APSW compared to SQLAlchemy
2. Number of DB writes to the disk is also **2x lesser** in case of APSW

Below is a direct comparison between the two processes. The description of these keys are in the [docs](#) for the **time** command.

APSW

```

Command being timed: "python3 netttacker.py -i google.com --pr
User time (seconds): 659.54
System time (seconds): 402.48
Percent of CPU this job got: 175%
Elapsed (wall clock) time (h:mm:ss or m:ss): 10:04.49
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 335396
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 182134
Voluntary context switches: 18320807
Involuntary context switches: 192416
Swaps: 0
File system inputs: 0
File system outputs: 12088
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0

```

SQLAlchemy

```

Command being timed: "python3 netttacker.py -i google.com --pr
User time (seconds): 275.49
System time (seconds): 175.56
Percent of CPU this job got: 145%
Elapsed (wall clock) time (h:mm:ss or m:ss): 5:10.11
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 632276
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 265572
Voluntary context switches: 3100882
Involuntary context switches: 82745
Swaps: 0
File system inputs: 0
File system outputs: 20824
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0

```


The reason for the different execution times (and hence CPU usage) is that I didn't mimic a test environment very well and I had some processes running which seemed to affect the multiprocessing in the first case with APSW.

The results we care about (*I/O operations and RAM usage*) will still remain valid (CPU usage depends on the execution time and that was hindered, but RAM doesn't depend on that and the *time* module calculates the peak usage for the process only).

The official benchmarks are described in these [docs](#) for APSW.

Remaining Work

I had to manually set a timeout value using *connection.setbusytimeout* to avoid database locking conditions even after *WAL* was enabled.

Additionally some vulnerability scan modules like *log4j_cve_2021_44228_vuln* submit a lot of temporary data through the *submit_temp_logs_to_db* function. When using APSW, I hit *apsw.CantOpenError* because there are lots of concurrent writes.

There are two ways to avoid this:

1. Explicitly increase the limit of the OS (the user needs to do this) using *ulimit -n*.
2. Manage it manually since APSW *doesn't have connection pooling* support like SQLAlchemy.

I will be working on the **second part** during my coding phase and **integrating this with SQLAlchemy** to support PostgreSQL and MySQL as well.

Porting to Huey

A basic implementation for switching to *Huey* can be found over at [@7aba750](#) in my local fork of *Nettacker*.

A brief summary of the changes made are:

- Created **tasks.py** to define **run_engine_task** and **scan_target_task** as Huey tasks.

```
@huey.task(retries=3, retry_delay=2)
def scan_target_task(target, arguments, module_name, scan_id,
process_number, thread_number, total_number_threads):
    options = argparse.Namespace(**copy.deepcopy(arguments))
    socket.socket, socket.getaddrinfo =
set_socks_proxy(options.socks_proxy)
    module = Module(
        module_name,
        options,
        target,
        scan_id,
        process_number,
        thread_number,
        total_number_threads,
    )

    module.load()
    module.generate_loops()
    module.sort_loops()
    module.start()

    log.verbose_event_info(
        _("finished_parallel_module_scan").format(
            process_number, module_name, target, thread_number,
total_number_threads
        )
    )
    return os.EX_OK
```

- Replaced **engine.run()** and **scan_target()** with huey tasks.
- Called **huey_consumer** to query and execute them in the background.

```

> huey_consumer.py nettracker.core.tasks.huey -w 8 -k process
/usr/local/bin/huey_consumer.py:4: DeprecationWarning: pkg_resources is deprecated as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html
  __import__('pkg_resources').run_script('huey==2.5.2', 'huey_consumer.py')
[2025-03-29 22:06:14,240] INFO:huey.consumer:735979:Huey consumer started with 8 process, PID 735979 at 2025-03-29 22:06:14,240
[2025-03-29 22:06:14,240] INFO:huey.consumer:735979:Scheduler runs every 1 second(s).
[2025-03-29 22:06:14,240] INFO:huey.consumer:735979:Periodic tasks are enabled.
[2025-03-29 22:06:14,241] INFO:huey.consumer:735979:The following commands are available:
+ nettracker.core.tasks.run_engine_task
+ nettracker.core.tasks.scan_target_task

```

Remaining Work

There are **three issues** that I will need to fix to make this implementation robust:

1. The logs are being printed on the huey_consumer side currently.

```

[2025-03-22 00:33:03,373] INFO:huey:749027:Executing nettracker.core.tasks.run_engine_task: 54248b86-ebc4-4712-b526-e21ad6
[2025-03-22 00:33:03,402] INFO:huey:749028:nettracker.core.tasks.run_engine_task: a3balf62-284e-4c9b-968a-a700d2f882ab 3
[2025-03-22 00:33:03,403] INFO:huey:749028:Executing nettracker.core.tasks.run_engine_task: 13c84a81-042b-4bbb-8ed0-1f2487
[2025-03-22 00:33:03,411] INFO:huey:749027:nettracker.core.tasks.run_engine_task: 54248b86-ebc4-4712-b526-e21ad6dee255 3
[2025-03-22 00:33:03,413] INFO:huey:749027:Executing nettracker.core.tasks.run_engine_task: c2f0b5c0-6ed4-4483-abec-09dde
[2025-03-22 00:33:03,440] INFO:huey:749030:nettracker.core.tasks.run_engine_task: 450c91c7-6c52-4e76-95d1-656dccbf1f5e5 3
[2025-03-22 00:33:03,442] INFO:huey:749030:Executing nettracker.core.tasks.run_engine_task: e878b359-684c-4e42-979f-4c40f
[2025-03-22 00:33:03,450] INFO:huey:749028:nettracker.core.tasks.run_engine_task: 13c84a81-042b-4bbb-8ed0-1f248765d0c2 3
[2025-03-22 00:33:03,451] INFO:huey:749028:Executing nettracker.core.tasks.run_engine_task: d69810af-f68c-41e4-b3da-3bbba
[2025-03-22 00:33:03,451] INFO:huey:749027:nettracker.core.tasks.run_engine_task: c2f0b5c0-6ed4-4483-abec-09dde436eead 3
[2025-03-22 00:33:03,453] INFO:huey:749027:Executing nettracker.core.tasks.run_engine_task: 47494213-74ed-4dc7-b6f0-1d84d
[2025-03-22 00:33:03,481] INFO:huey:749030:nettracker.core.tasks.run_engine_task: e878b359-684c-4e42-979f-4c40fb1b2e87 3
[2025-03-22 00:33:03,483] INFO:huey:749030:Executing nettracker.core.tasks.run_engine_task: 8f5c67e2-7b0c-451c-9f51-a95a6
[2025-03-22 00:33:03][+++] process-0|port_scan|en.wikipedia.org|module-thread 1/1|request-thread 78/1005|host: en.wikiped
_and_receive port: 443 timeout: 3.0|
success_condition (s):
conditions: http: - 'Content-Type: ' - HTTP/1.1 400

```

2. Huey returns immediately causing databases to be queried before they are populated. Thus the scans stop as it enters this if statement inside `app.py`.

```

if not self.arguments.targets:
    log.info(_("no_live_service_found"))
    return True

```

Huey tasks execute in the background and return from the main function immediately. Running `scan_target_task` inside `scan_target_groups` in `app.py` makes the latter return **True** immediately. The `find_events` function expects the `scan_id/logs` to be there when it is called and this doesn't happen as the consumer is still processing the task. This is a [small doc](#) explaining this in a little more detail.

To solve this, we will need to query the `temp result` database and see if it's populated before returning directly.

3. The user shouldn't have to run the huey_consumer themselves as this implementation requires.

For this the **subprocess** module will be used with user defined values for threads and cores etc. The arguments passed on to Nettacker will be used to define values to run the huey_consumer inside `app.py`.

Adding New Output types

This section describes the integration of Nettacker with [DefectDojo](#) (DD) and inclusion of the **SARIF** output format.

Integrating Nettacker with DefectDojo

The integration between Nettacker and DefectDojo is achieved via creating an output type that is compatible with the **generic parser** that is present with the latter.

A sample output for a Nettacker scan in DefectDojo compatible format is present in this [Gdrive link](#).

The base code for integration with DefectDojo can be found [@81cc8a9](#) in my local fork.

A small summary is presented below (please refer to the commit above for the code):

The minimum requirements for the generic parser to work can be confirmed with the [parser's code](#).

```
# check for required keys
required = ["title", "severity", "description"]
missing = sorted(required.difference(item))
if missing:
    msg = f"Required fields are missing: {missing}"
    raise ValueError(msg)
```

- The **/core/graph.py** file needs to include a new file type extension. I am calling this **.dd.json**.
- A new function called **create_dd_specific_json** needs to be created which will take **all_scan_logs** and output a json dump in the generic parser's expected format.
- A translation is made from CVSS score to DefectDojo compatible severity values, which are **Info**, **Low**, **Medium**, **High** and **Critical**.

To ensure proper representation of the vulnerability, the following changes will be made:

Old Keys (all_scan_logs)	New Keys (DefectDojo)	Comments
date (YYYY-MM-DD hh-mm-ss)	date (YYYY-MM-DD)	Trimming time details
target	service	Mapping target to service
port	param	No better alternative found
module_name	title	Required field
event	impact	Closest alternative
json_event	severity_justification	Closest alternative
scan_id	unique_id_from_tool	Unique identifier

Additional Required Fields;

description	Pulled from YAML files (required field)
severity	Pulled from YAML files (required field)

The final json can be parsed by the generic parser easily.

SARIF output format

I will be referring to the [sarif-schema-2.1.0.json](#) for the description and definition of the keys used. The SARIF files are being validated through [this tool](#).

A prototype code for SARIF output integration can be found over [@3f46f5](#) at my local fork.

The minimum requirements for a valid SARIF format is:

```
{
  "version": "2.1.0",
  "$schema": "https://json.schemastore.org/sarif-2.1.0.json",
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "ExampleAnalyzer",
          "version": "1.0.0",
          "informationUri": "https://example.com/analyzer"
        }
      }
    }
  ]
}
```

A new output type can be created as described before. The function to convert `all_scan_logs` to valid SARIF will be called `create_sarif_output`.

The following substitutions will be made:

<code>ruleId</code>	<code>name</code> of the module
<code>message</code>	<code>event</code> value for each log in <code>all_scan_logs</code>
<code>locations.physicalLocations.artifactLocation.uri</code>	<code>target</code> value

webRequest.properties.json_event	json_event value for each log in all_scan_logs
properties.scan_id	scan_id unique value for each run
properties.date	date field specified in all_scan_logs

The resulting SARIF file can be found on this [Gdrive link](#) and can be verified with the above link.

Web Server Improvements

There are two things that I want to implement here that a pentester or a bug bounty hunter would benefit from:

1. Creating input fields for **custom enumeration, usernames and password** wordlists
2. Integrating **Huey with flask** for faster API calls.

Creating input fields for custom directory enumeration wordlist

This requires directly modifying the value of the **read_from_file** argument being passed to Nettare and changing it the path the user provides.

The basic case has been implemented over in this [changelog](#) and this commit [@1e798b8](#) in my local fork. A small summary is provided below:

- The field will be placed like this:

Submit a new scan

Basic

Advanced

Targets

Add new target

Enum wordlist

filename

Custom enum wordlist

- Since this is a new feature, it should be included in the **introduction sequence**, which can be achieved by modifying **main.js**.
- Then the **read_from_file** value can be overwritten inside **engine.py**.

The remaining will be handled by previously pushed code through PR [#1026](#).

Creating input fields for username and password files

A base implementation can be found over [@4469c41](#) in my local fork.

A brief description of the changes is provided below.

- This is how the fields will be placed:

- If the user has entered a filename, then that is added to the **post request data**, else we add the normal input.
- The **usernames_list** and **passwords_list** values are overwritten inside **engine.py** if the user enters files for them, else they're kept as default.

The remaining is handled by Netttacker.

Integrating Huey with the Web Server

This integration will allow the API to take multiple requests without having to process them first.

A basic implementation for running Huey with the web server can be found over [@c305812](#) in my local fork. This is an extension of the commit [@7aba750](#).

A summary of the changes are:

1. Created a huey task for the `new_scan` endpoint in `engine.py`.

```
nettacker_app = Nettacker(api_arguments=SimpleNamespace(**form_values))
app.config["OWASP_NETTACKER_CONFIG"]["options"] = nettacker_app.arguments
thread = Thread(target=nettacker_app.run)
thread.start()
task_result = new_scan_task(form_values)
```

2. Tasks for `engine.run()` and `scan_target` are also defined as done previously.
3. The `new_scan_task` returns immediately, and the task is queued.

Running `huey_consumer` now will result in the execution of the tasks. This reduces overhead in the server and avoids latency.

This is the output of running a `dir_scan` on `owasp.org` (the HTML output will come on the website). The logs are printed on the `huey_consumer` side.

```
[2025-03-31 18:59:18,738] INFO:huey:64293:Executing nettacker.core.tasks.run_engine_task: b4d8b10c-4e8e-441e-9aa7-afb439ab
[2025-03-31 18:59:18,739] INFO:huey:64292:Executing nettacker.core.tasks.run_engine_task: 01b62c4f-2f6c-4e66-8844-449c7fb1
[2025-03-31 18:59:18,739] INFO:huey:64295:nettacker.core.tasks.run_engine_task: 4bb132d6-6185-4e9b-a459-2ca0b92aacc8 3 ret
[2025-03-31 18:59:18,741] INFO:huey:64295:Executing nettacker.core.tasks.run_engine_task: 3f5b53e4-9170-41cb-bd8c-39feca04
[2025-03-31 18:59:18,744] INFO:huey:64294:nettacker.core.tasks.run_engine_task: 42ec66c3-2ec5-4fc3-82c1-1fab25202102 3 ret
[2025-03-31 18:59:18,746] INFO:huey:64294:Executing nettacker.core.tasks.run_engine_task: 5221572b-6bcb-4dd3-b79d-4897cf90
[2025-03-31 18:59:19][+++] process-0|port_scan|owasp.org|module-thread 1/1|request-thread 29/1005|host: owasp.org method:
success_condition (s):
conditions: http: - 'Server: ' - 'Content-Type: ' - 'Content-Length:...' [see the full content in the report]
[2025-03-31 18:59:19,970] INFO:huey:64292:nettacker.core.tasks.run_engine_task: 01b62c4f-2f6c-4e66-8844-449c7fb1f51c 3 ret
[2025-03-31 18:59:19,972] INFO:huey:64292:Executing nettacker.core.tasks.run_engine_task: d3a3625a-8f70-4006-8e47-002220b
```

Remaining Work

1. Running the consumer in the background using python `subprocess`.
2. Combining this with `APSW` for more optimization. (Huey by itself cannot use APSW as a broker but I will use this for the `result database`).

Note that even if we don't use huey for engine.run and scan_target, `new_scan_task` will suffice. It just won't be as efficient because the background processing will not be pure.

Improving Test coverage

I raised PRs [#1042](#), [#1043](#) and [#1039](#) to familiarize myself with the general test structure.

Test cases for `template.py` and `core.py` have been implemented (and tested) in my local fork in this [changelog](#).

The timeline will explain how and when I plan to write test cases. I hope to do the majority of this work before the coding period begins so I can focus on writing tests for new functions that I will write for my ideas.

Timeline

Application Review Period

April 8th - May 8th

- My end-of-semester exams will start from **May 5th** and continue until **May 9th**. I will need a week leading up to my exams in order to prepare.
- In the meantime, from April 8th to April 20th, I will focus on **preparing my code** for the coding period to avoid potential delays.
 - I will start to make my implementation of **version scanning** more robust and start ironing out issues. This will allow me more time to work on it and get it right.
 - I will also be working on a fully fledged implementation for integrating **flask with huey** for the webserver.
 - Additionally I will start **improving regexes** in port.yaml by testing locally.

I wish to work on the last two ideas above, because regardless of the outcome of GSoC they will help Netstacker and users.

Community Bonding Period

May 8th - June 1st

- I plan to use the Community Bonding Period in its true sense and learn more about OWASP. I have a few more things I want to try and explore under owasp, especially figure out how the **OWASP top 10 is formulated** and the data collection that goes behind that.
- I will also continue to spend this time on my project ideas and convert the pseudocodes I provided above to fully fledged ones.

This will take up the majority of this period. Specifically, I will use this time to start **writing test cases**, and I hope to finish the majority of them, and fix issues with my **version scanning** approach.

- By the end of this period, I will push a **PR** for porting from SQLAlchemy to **APSW for SQLite**, as I have most of the implementation ready

GSoC Period

I plan on dividing my time in **4 distinct phases** of 3 weeks each. The following neatly describes the time I plan on taking to implement each idea properly in each phase.

I have included my **priority** preference with the same, and I have kept the high priority task based on what requires the **most work and will be beneficial** to Netstacker.

Phase 1 (week 0 - 3)

June 2nd - June 23rd

Medium priority

- I will continue working on the PR I raise during the Community Bonding Period for the **port to APSW** and aim to finish it within a week
- I will also start implementing **version logging** and continue this work into Phase 2.
- I will also push five PRs for the **5 test cases in parallel**. These test cases will be for files :
 - a. `arg_parser.py`
 - b. `template.py`
 - c. `logger.py` (this requires 48% more coverage)
 - d. `socket.py` (a few branching conditions need to be tested)
 - e. `graph.py`
- I hope to have **already implemented** most of these tests during the community bonding period, so I will just be pushing them.
- **graph.py** will require tests for APSW, which is what I will be **writing** in this phase.

Phase 2 (week 3 - 6)

June 23rd - July 14th

High priority

- I will continue working on improving **version logging**. I hope to finish this in the first 2 weeks and keep the last one for **TCP and UDP** port detection.
- In parallel, I will write two test cases for the following files:
 - a. `core.py`
 - b. `module.py`
- I chose these because they **won't be modified significantly** by my implementation of version scanning, and hence could be done independently.
- I will utilize this phase fully till July 14th and start the new phase independently.

Phase 3 (week 6 - 9)

July 14th - August 4th

Medium priority

- Here, I will make my code for the **new output types** more robust and raise PRs for them. I will aim to finish this within a week.

- In parallel, I will create four test cases for the following four files:
 - a. lib/icmp/engine.py
 - b. api/engine.py
 - c. app.py
 - d. core/lib/ssh.py
- I will finish creating **input fields** for custom passwords, usernames and custom wordlists and raise 2 PRs for them respectively.
- During the last week, I will start preparing my implementation of using **Huey with Flask** for Netttacker by addressing the issues mentioned under the remaining work section.
- Additionally, after making these changes, I will dedicate extra time to include tests for them.

Phase 4 (week 9 - 12)

August 4th - August 25th

Medium priority

I will break this phase into two alternatives. **Alternative A** will describe the ideal deliverables during this time and **alternative B** will be the minimum deliverables. This is because I will be in Australia for 10 days in August and might not be able to get much work during that period.

Alternative A:

- I would raise a PR and continue working on **huey with flask** for the web server in this phase (this includes porting to huey completely). I hope to have achieved a majority of the code during the Application review period so this shouldn't be long.
- The following 3 test cases will be created in parallel:
 - a. core/lib/base.py (would have incurred changes during GSoC period)
 - b. core/lib/telnet.py
 - c. core/lib/ftp.py

Alternative B:

- I will raise a PR and finish the **port to huey** implementation **without** similar support for the webserver. This is something I already tried and tested, so I will be cleaning up issues majorly, without the added complexity of doing the same for the webserver.
- The following 2 test cases will be created:

- a. core/lib/base.py (would have incurred changes during GSoC period)
- b. core/lib/ftp.py

I would like to keep the **final week as a buffer** to work on any unfinished ideas (turn Alternative B to Alternative A). In the case where there are none, I will spend this time writing documentation for all the changes I have made.

I usually write draft notes in parallel to whatever I do, so I will refine them for a more professional style.

Post GSoC Period

- I want to develop Netacker into a better overall tool for **industrial recognition**, than it was when I started. After my tenure in GSoC I will try to guide my fellow contributors into developing this further by providing **new suggestions** and **reviewing their code**.
- I will update the README and documentation (if the maintainers need me to) in order to document all the new features and changes made, if I cannot do this during the final week.
- In case I have any pending PRs related to GSoC, I will ensure that I finish them first.

OWASP was my **introduction** to open source and is therefore quite dear to me. I plan to continue contributing to the organization in any possible way and implementing the interesting new things that I learn about cybersecurity in it.

I have some new ideas that I couldn't articulate well enough in time, so I will discuss the feasibility of these with the mentors to see if we can work on them before any other major code refactoring.

I want to eventually come back to **Netacker as a mentor** (hopefully having learned enough to lessen the burden on the maintainers) or if OWASP would allow, I would love to take part in **GSoC again next year**, implementing newer and cooler ideas.

Time commitments

- I will be going to **Australia** during the month of August as an electrical engineer in the **World Solar Challenge** for **10 days**. This is a well planned initiative and there will **not be any additional delays** past this.
- My work will majorly be during the morning hours from **6am to 5pm** (the sun sets at around 5pm) and I will be free to pursue my **GSoC goals after that period**.

My timeline also accounts for this delay, and I have ensured that my work will not be hindered by having an alternate plan. The alternate plan uses the same base code as the original, so depending on the situation, I will be able to switch.

I will be extremely free during the **month of July** and can easily compensate for any potential backlogs during August. (This is where I have allocated my high-priority task)

- Other than that, I have no other commitments, and in the weeks leading up to August, I can easily dedicate upwards of 30 hours a week to ensure consistent progress.

If I am caught up with any work (personal or miscellaneous) and might miss a deadline or a weekly meeting, I would **inform my mentor way beforehand** and would also indicate the date spans where I would put in more time and get the job done.

Acknowledgements

As I finish my proposal, I want to thank [@securestep9](#) for guiding me through my first PRs and opening up the world of open-source for me. I also want to thank [@impaler343](#) (Pranav Raghu) and [@anutosh491](#) (Anutosh Bhat) for providing a community, helping me with my proposal and giving insanely good feedback. Additionally I am grateful for [@jgadsden](#)'s (Jon Gadsden) and [@jmanico](#)'s (Jim Manico) support, it greatly boosted my confidence in the early days.