Friendly & Lightweight Unified Fuzzing Framework (FLUFF) - Version 1.2

13th March 2021

Michael Curnow
Defiant Networks, Inc
takko_the_boss@protonmail.com

Preamble

Abstract

Performing proper fuzz testing is paramount to conducting a comprehensive penetration test for varying technologies. As the need for security grows with the expansion of technology, so does the need to test the robustness of such. Fuzz testing is a dependable way to test whether a system is trustworthy enough to handle interaction with users and other pieces of integrated technologies & machinery. And the ways to employ fuzz testing seem as plentiful as the litany of the technology itself. Due to most frameworks being successful in the vacuum of their own technology stacks, coupled with the lack of an umbrella/all-encompassing guideline on how to perform fuzz-testing from front-to-end via a simple and easy to digest guideline, this document was produced. Differences in technology stacks can present a disparity in technique, nomenclature, and even procedures. FLUFF aims to extract the common denominators that exist amongst different fuzzing realms, and compile them into an easy to understand and employ framework.

INTRODUCTION

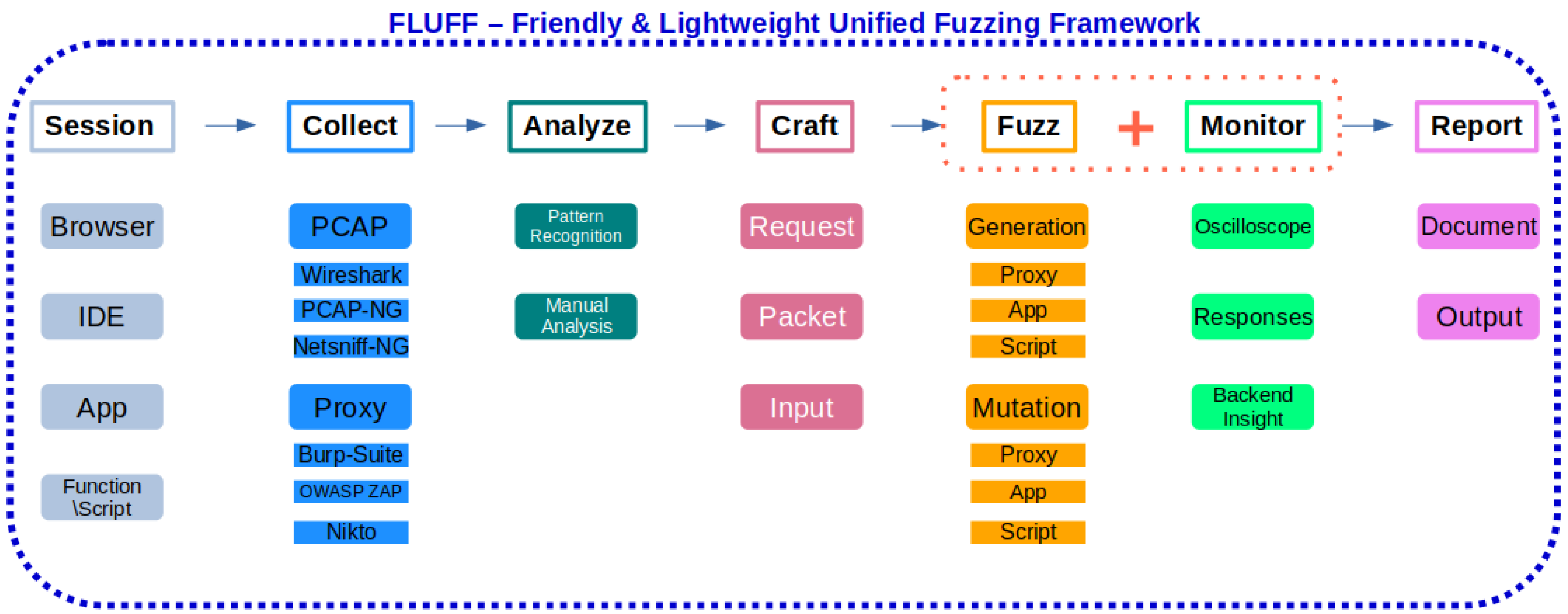
For those who are unfamiliar with the term or practice of ***Fuzz Testing***, it's essentially testing target system’s potential for glitches and crashes by employing a combination of testing input validation coupled with stress testing, which is performed by spraying inputs of a target device, application, or system with random and crafted input at high velocity and observing the effects on the target. Use cases can involve testing input forms on a website to ensure it doesn't accept foreign or script characters or ensuring that IIoT devices are programmed to be functionally robust.

PURPOSE

Most fuzz testing frameworks currently exist in a fractured state, where different technology silos hold valuable processes often relayed in documents that reflect the vacuum they were drafted in. FLUFF serves to give offensive security testers of varying technology domains and expertise a resource to refer to when planning & conducting fuzz tests by extracting out high-level practices from domain-specific knowledge and procedure spread out amongst the different technology domains under one roof for easy practice and application.

Framework Architecture

FLUFF consists of 7 different domains. Of those, **Fuzz** & **Monitor** are the only 2 that should be performed in parallel (if possible).



Session

Preparing the means of which you'll communicate with the target and ideally will provide. For example: for a web application that could be a browser, or if a web-API is used then likely there is an API script made available to interact with the target(s).

Sessions can be initialized via:

- Web browser
  - *Example: Google Chrome, Mozilla Firefox, Opera, Brave, Waterfox, etc*
- Software specific IDE
  - *Example: MySQL Workbench, Proficy - Machine Edition, Arduino IDE, etc*
- Application meant to converse with the target
- A script of function built to interact with a certain API or system

Collect

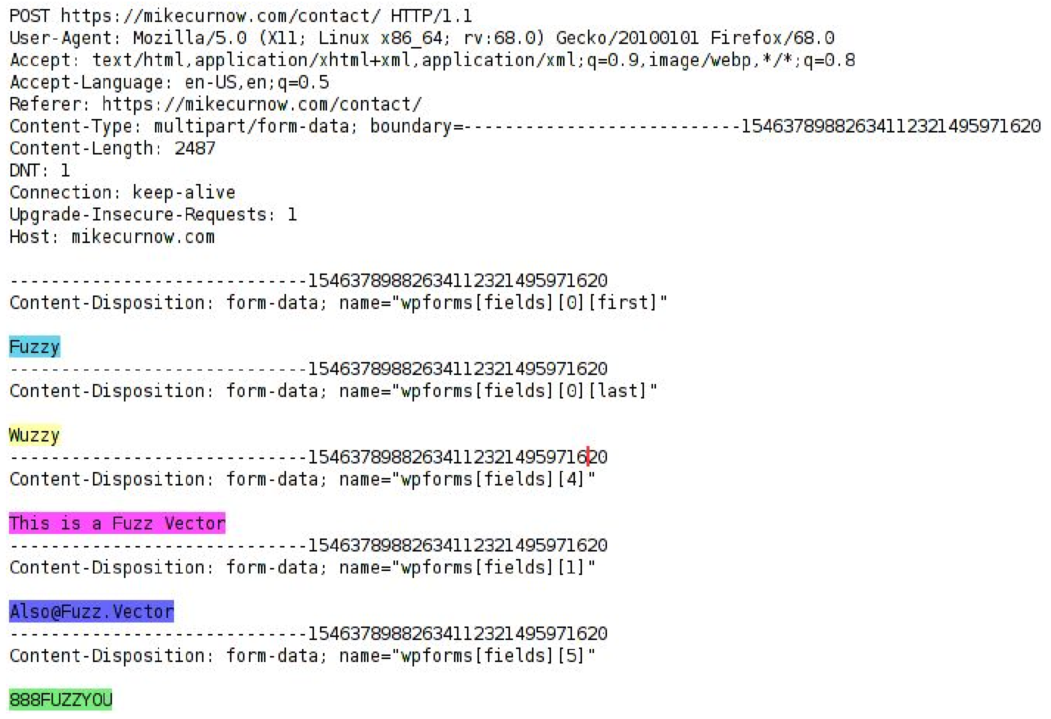
Generating session traffic against your target, and accumulating the conversation as either HTTP requests, network packet, or some other domain/technology output/feedback that's collectable. Typically collection types and tools are (but **not** limited to):

- Packet Capture (PCAP)
  - Wireshark
  - PCAP-NG
  - Netsniff-NG
- Proxied Requests
  - Burpsuite
  - OWASP ZAP
  - Nikto

It’s better to have more versus less in regards to *how much* you want to collect. It’d also behoove you as the tester to diversify your collection portfolio by enacting a wide variety of interactions to capture, thus allowing you to derive a broader range of use-cases in your analysis and crafting steps.

Analyze

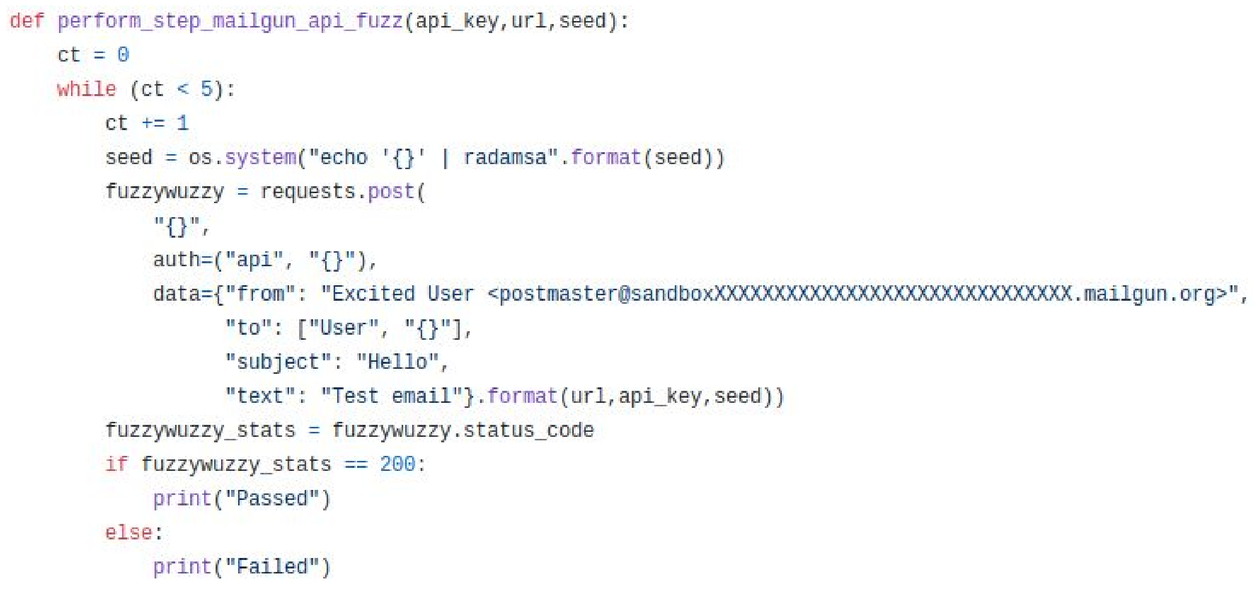
Placing your collected interactions under a scope of scrutinous examination to enumerate and identify any inputs to be leveraged as possible fuzz vectors.



```
POST https://mikecurnow.com/contact/ HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: https://mikecurnow.com/contact/
Content-Type: multipart/form-data; boundary=---------------------------15463789882634112321495971620
Content-Length: 2487
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Host: mikecurnow.com

-----------------------------15463789882634112321495971620
Content-Disposition: form-data; name="wpforms[fields][0][first]"

Fuzzy
-----------------------------15463789882634112321495971620
Content-Disposition: form-data; name="wpforms[fields][0][last]"

Wuzzy
-----------------------------15463789882634112321495971620
Content-Disposition: form-data; name="wpforms[fields][4]"

This is a Fuzz Vector
-----------------------------15463789882634112321495971620
Content-Disposition: form-data; name="wpforms[fields][1]"

Also@Fuzz.Vector
-----------------------------15463789882634112321495971620
Content-Disposition: form-data; name="wpforms[fields][5]"

888FUZZYOU
```

Craft

Creating the actual requests or means to send your input to the target’s fuzz vectors. This may look different in regards to the technology stack scope. However the picture below shows what a programmatic approach to fuzzing a web API in the form of a Python function would look like.



```
def perform_step_mailgun_api_fuzz(api_key,url,seed):
    ct = 0
    while (ct < 5):
        ct += 1
        seed = os.system("echo '{}' | radamsa".format(seed))
        fuzzywuzzy = requests.post(
            "{}",
            auth=("api", "{}"),
            data={"from": "Excited User <postmaster@sandboxXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.mailgun.org>",
                  "to": ["User", "{}"],
                  "subject": "Hello",
                  "text": "Test email"}.format(url,api_key,seed))
        fuzzywuzzy_stats = fuzzywuzzy.status_code
        if fuzzywuzzy_stats == 200:
            print("Passed")
        else:
            print("Failed")
```

Fuzz + Monitor

Fuzz

Executing the fuzz attempts on the target's fuzz vectors. This can be a lengthy process, taking hours or sometimes longer than a day. For true usefulness of fuzzing it needs to be employed quite regularly. Some run fuzzing at short iterations or constantly in a dev/test environment to reach the full usefulness of this effort.

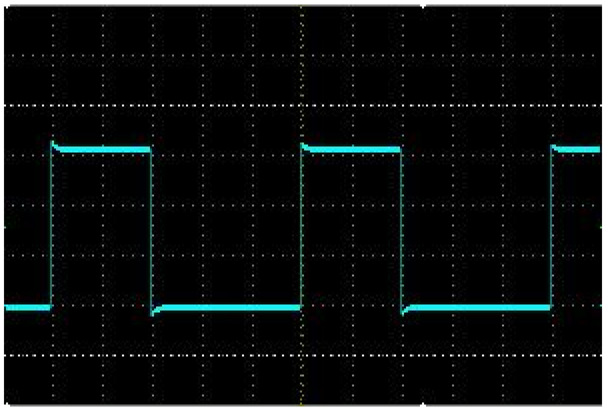
Monitor

Whether it be via simple HTTP responses, backend monitoring, or whatever method you wish to use, it's important to have a semblance of monitoring engaged while fuzzing. It's important to know what the feedback is, if you need to throttle requests sent to a physical controller perhaps, etc. Below are a couple examples of what monitoring *can* look like (but not limited to).

Simple web API response output.

```
def perform_step_mailgun_api_fuzz(api_key,url,seed):
    ct = 0
    while (ct < 5):
        ct += 1
        seed = os.system("echo '{}' | radamsa".format(seed))
        fuzzywuzzy = requests.post(
            "{}",
            auth=("api", "{}"),
            data={"from": "Excited User <postmaster@sandboxXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.mailgun.org>",
                  "to": ["User", "{}"],
                  "subject": "Hello",
                  "text": "Test email"}.format(url,api_key,seed))
        fuzzywuzzy_stats = fuzzywuzzy.status_code
        if fuzzywuzzy_stats == 200:
            print("Passed")
        else:
            print("Failed")
```

Observed normal output from an oscilloscope that could be hooked up to a programmable logic controller (PLC) to measure voltage amplitude during testing period.



Report

Different technology domains will have their own idiosyncratic & subjective benchmarks, which at a certain level looks like a sense of inconsistency. However when reporting, make sure you're tracking relevant dimensions unique to *your own* test/use case. For example - if you're fuzzing a PLC, you're looking for the following (most likely):

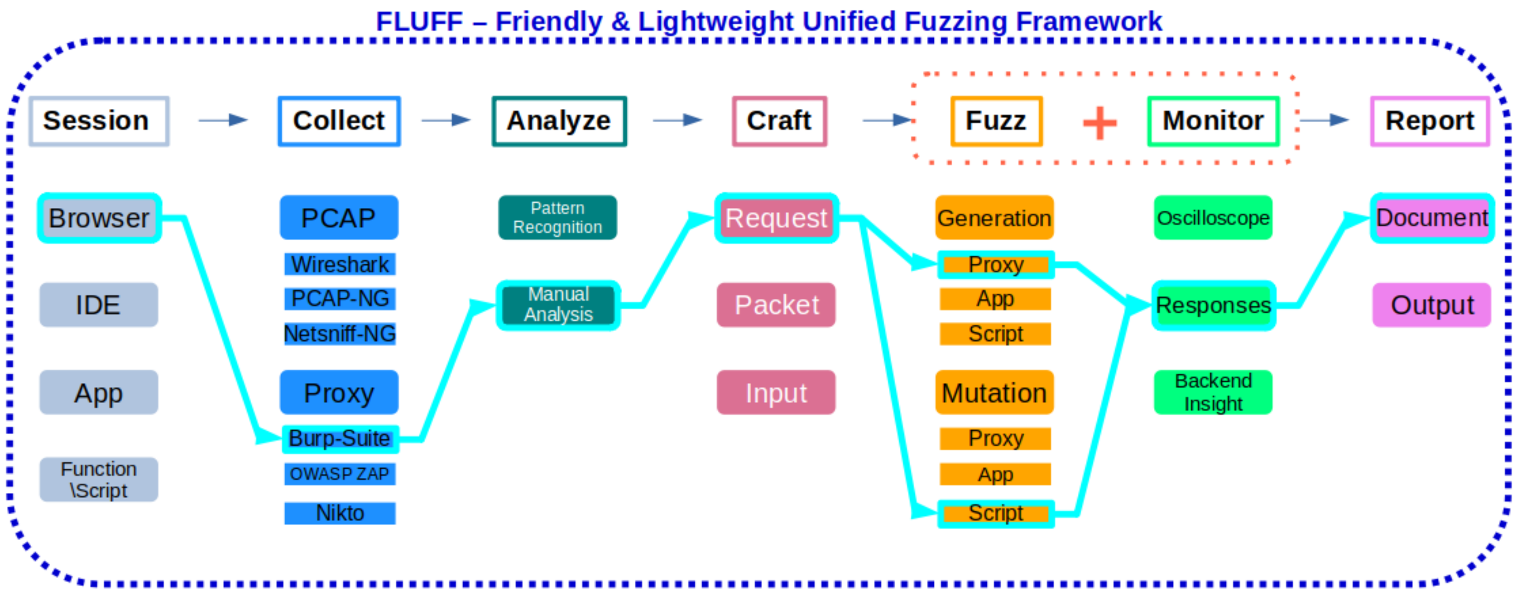
- Robustness: Does the smallest and/or most negligible fuzz request make the device freeze?
- Input validation on Holding Register: Can I enter nonsense input to the PLC, and will it accept and apply it?
- Timeout: *How Long* does it take to stall after fuzzing?
- Coil logic validation: Can I enter conflicting logic to what's already programmed in the PLC's logic programming (Binary Values)?

If you're testing an API, your benchmarks are going to look quite different than the above. So don't let cross-domain inconsistencies hinder this process. The easiest way to start listing benchmarks is to first identify all the faults your target ***could*** have, and try to prove them via fuzz. Remember, your goal when fuzz testing shouldn't be to maintain a semblance of consistency across varying technology domains.

Example Use Cases for FLUFF (some)

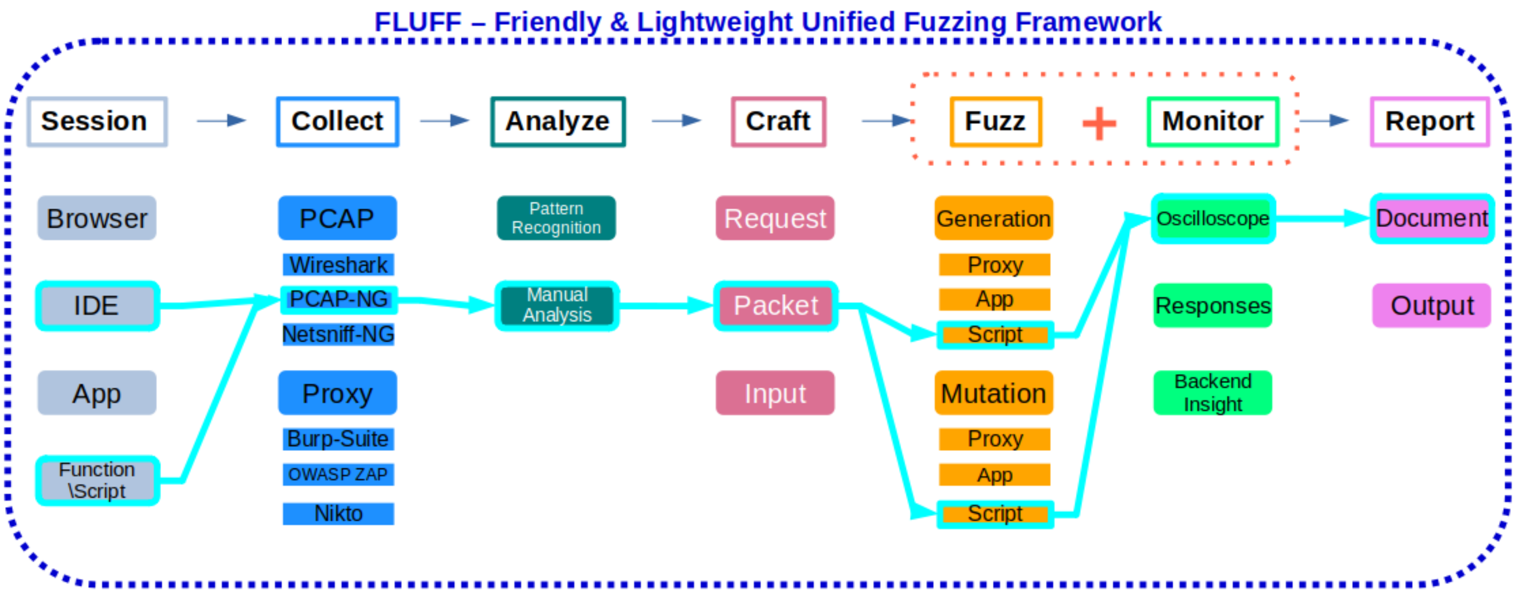
Website Contact Form

When testing a web app contact form, you might apply the following path for your fuzz test.



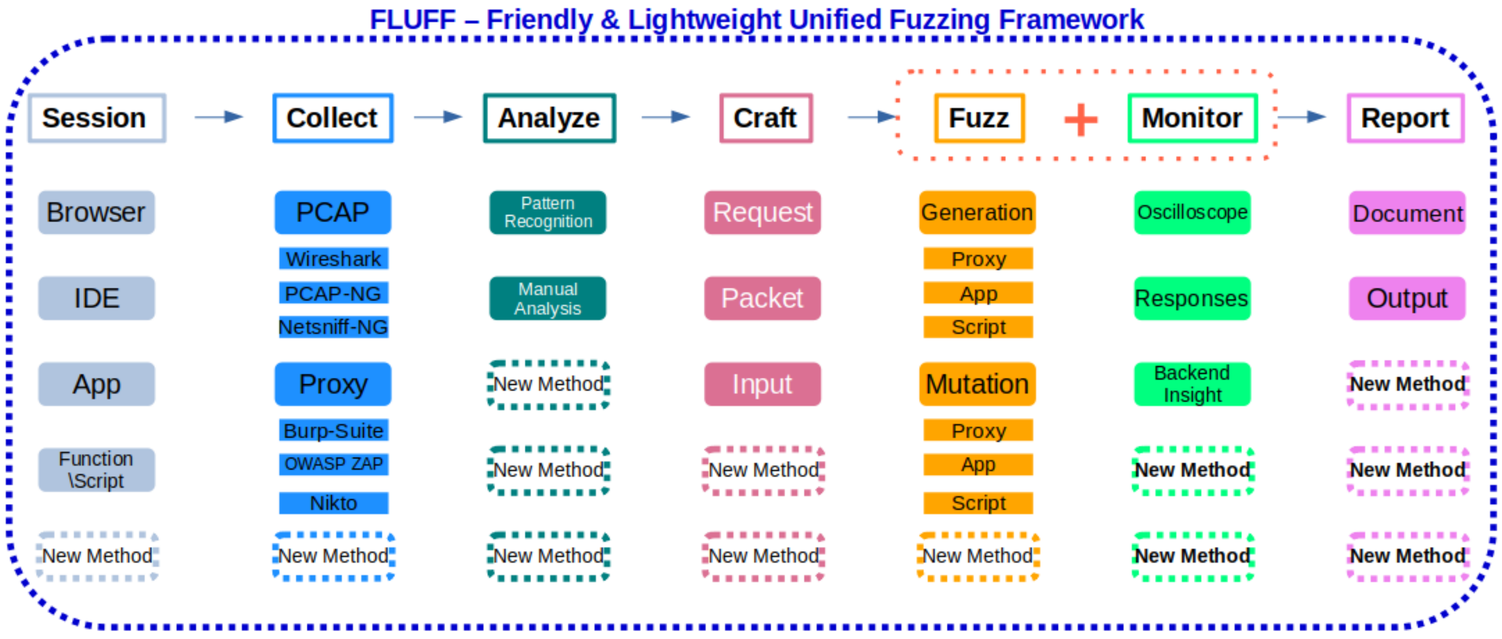
Unknown/Proprietary Industrial Communications Protocol

When testing an industrial protocol you know nothing about, you might apply the following path for your fuzz test.



Extendibility

The architecture is flexible and extendable. Meaning, as you discover additional methods for each step, and you can add those methods into the process.



Final thoughts and conclusions

This framework should serve as a high-level guideline which is emplaced like a stencil over your test case. If anything, this document should serve you as *a place to start* and build off of. Fuzz testing is the epitome of diving into the unknown, and it's easy to get lost or go off course. Use this a guide to ground you in your fuzzing endeavors.

May the *fuzz* be with you.