geneva

May 18, 2023

Getting Started:

Disclaimer: Running Geneva or Geneva's strategies may place you at risk if you use it within a censoring regime. Geneva takes overt actions that interfere with the normal operations of a censor and its strategies are detectable on the network. During the training process, Geneva will intentionally trip censorship many times. Geneva is not an anonymity tool, nor does it encrypt any traffic. Understand the risks of running Geneva in your country before trying it.

CHAPTER 1

Introduction

Geneva is an artificial intelligence tool that defeats censorship by exploiting bugs in censors, such as those in China, India, and Kazakhstan. Unlike many other anti-censorship solutions which require assistance from outside the censoring regime (Tor, VPNs, etc.), Geneva runs strictly on one side of the connection (either the client or server side).

Under the hood, Geneva uses a genetic algorithm to evolve censorship evasion strategies and has found several previously unknown bugs in censors. Geneva's strategies manipulate the network stream to confuse the censor without impacting the client/server communication. This makes Geneva effective against many types of in-network censorship (though it cannot be used against IP-blocking censorship).

Geneva is composed of two high level components: its *genetic algorithm* (which it uses to evolve new censorship evasion strategies) and its *strategy engine* (which is uses to run an individual censorship evasion strategy over a network connection).

Geneva's Github page contains the Geneva's full implementation: its genetic algorithm, strategy engine, Python API, and a subset of published strategies. With these tools, users and researchers alike can evolve new strategies or leverage existing strategies to evade censorship. To learn more about how Geneva works, see *How it Works*.

CHAPTER 2

Setup

Geneva has been developed and tested for Centos or Debian-based systems. Due to limitations of netfilter and raw sockets, Geneva does not work on OS X or Windows at this time and requires *python3.6*.

Install netfilterqueue dependencies:

```
# sudo apt-get install build-essential python-dev libnetfilter-queue-dev libffi-dev␣
↪libssl-dev iptables python3-pip
```

Install Python dependencies:

```
# python3 -m pip install -r requirements.txt
```

2.1 Docker (Optional)

Geneva has an internal system that can be used to test strategies using Docker. This is largely used for testing fitness functions with the mock censors provided - it is **not used for training against real censors**. Due to limitations of raw sockets inside docker containers in many builds of Docker, Geneva cannot be used inside a docker container to communicate with hosts outside of Docker's internal network.

When used with Docker, Geneva will spin up three docker containers: a client, a censor, and a server, and configure the networking routes such that the client and server communicate through the censor. To evaluate strategies (see much more detail in the evaluation section), Geneva will run the plugin client inside the client and attempt to communicate with the server through the censor.

Each docker container used by the evaluator runs out of the same base container.

Build the base container with:

```
docker build -t base:latest -f docker/Dockerfile .
```

Optionally, to manually run/inspect the docker image to explore the image, run:

```
docker run -it base
```

CHAPTER 3

Getting Started

See our website or our research papers for an in-depth read on how Geneva works.

This documentation will provide a walkthrough of the main concepts behind Geneva, the main components of the codebase, and how they can be used.

This section will give a high level overview on how Geneva works; before using it, you are **strongly recommended** to read through *How it Works*.

3.1 What is a Strategy?

A censorship evasion strategy is simply a *description of how network traffic should be modified*. A strategy is **not code**, it is a description that tells the *strategy engine* how it should manipulate network traffic.

The goal of a censorship evasion strategy is to modify the network traffic in a such a way that the censor is unable to censor it, but the client/server communication is unimpacted.

3.2 Strategies & Species

Because Geneva commonly identifies many different strategies, we have defined a *taxonomy* to classify strategies into.

The Strategy taxonomy is as follows, ordered from most general to most specific:

- 1. Species: The overarching bug a strategy exploits
- 2. Subspecies: The mechanism used to exploit the bug
- 3. Variant: Salient wireline differences using the same bug mechanism

The highest level classification is *species*, a broad class of strategies classified by the type of weakness it exploits in a censor implementation. `TCB Teardown` is an example of one such species; if the censor did not prematurely teardown TCBs, all the strategies in this species would cease to function.

Within each species, different *subspecies* represent unique ways to exploit the weakness that defines the strategy. For example, injecting an insertion TCP `RST` packet would comprise one subspecies within the TCB Teardown species; injecting a TCP `FIN` would comprise another.

Within each subspecies, we further record *variants*, unique strategies that leverage the same attack vector, but do so slightly differently: corrupting the checksum field on a `RST` packet is one variant of the `TCB Teardown w/ RST` subspecies of the `TCB Teardown` species; corrupting the `ack` field is another.

We refer to specific individuals as *extinct* if they once worked against a censor but are no longer effective (less than 5% success rate). That formerly successful approaches could, after a few years, become ineffective lends further motivation for a technique that can quickly learn new strategies.

3.3 Running a Strategy

For a fuller description of the DNA syntax, see *Censorship Evasion Strategies*.

```
# python3 engine.py --server-port 80 --strategy "\/" --log debug
2019-10-14 16:34:45 DEBUG:[ENGINE] Engine created with strategy \/ (ID bm3kdw3r) to␣
↪port 80
2019-10-14 16:34:45 DEBUG:[ENGINE] Configuring iptables rules
2019-10-14 16:34:45 DEBUG:[ENGINE] iptables -A OUTPUT -p tcp --sport 80 -j NFQUEUE --
↪queue-num 1
2019-10-14 16:34:45 DEBUG:[ENGINE] iptables -A INPUT -p tcp --dport 80 -j NFQUEUE --
↪queue-num 2
2019-10-14 16:34:45 DEBUG:[ENGINE] iptables -A OUTPUT -p udp --sport 80 -j NFQUEUE --
↪queue-num 1
2019-10-14 16:34:45 DEBUG:[ENGINE] iptables -A INPUT -p udp --dport 80 -j NFQUEUE --
↪queue-num 2
```

Note that if you have stale `iptables` rules or other rules that rely on Geneva's default queues, this will fail. To fix this, remove those rules.

3.4 Strategy Library

Geneva has found dozens of strategies that work against censors in China, Kazakhstan, India, and Iran. We include several of these strategies in strategies.md . Note that this file contains success rates for each individual country; a strategy that works in one country may not work as well as other countries.

Researchers have observed that strategies may have differing success rates based on your exact location. Although we have not observed this from our vantage points, you may find that some strategies may work differently in a country we have tested. If this is the case, don't be alarmed. However, please feel free to reach out to a member of the team directly or open an issue on this page so we can track how the strategies work from other geographic locations.

CHAPTER 4

Disclaimer

Running these strategies may place you at risk if you use it within a censoring regime. Geneva takes overt actions that interfere with the normal operations of a censor and its strategies are detectable on the network. During the training process, Geneva will intentionally trip censorship many times. Geneva is not an anonymity tool, nor does it encrypt any traffic. Understand the risks of running Geneva in your country before trying it.

CHAPTER 5

How it Works

See our website or our research papers for an in-depth read on how Geneva works.

This documentation will provide a walkthrough of the main concepts behind Geneva, the main components of the codebase, and how they can be used.

5.1 Censorship Evasion Strategies

A censorship evasion strategy is simply a *description of how network traffic should be modified*. A strategy is **not code**, it is a description that tells Geneva's strategy engine how it should manipulate network traffic.

The goal of a censorship evasion strategy is to modify the network traffic in a such a way that the censor is unable to censor it, but the client/server communication is unimpacted.

A censorship evasion strategy composed of one or more packet-level building blocks. Geneva's core building blocks are:

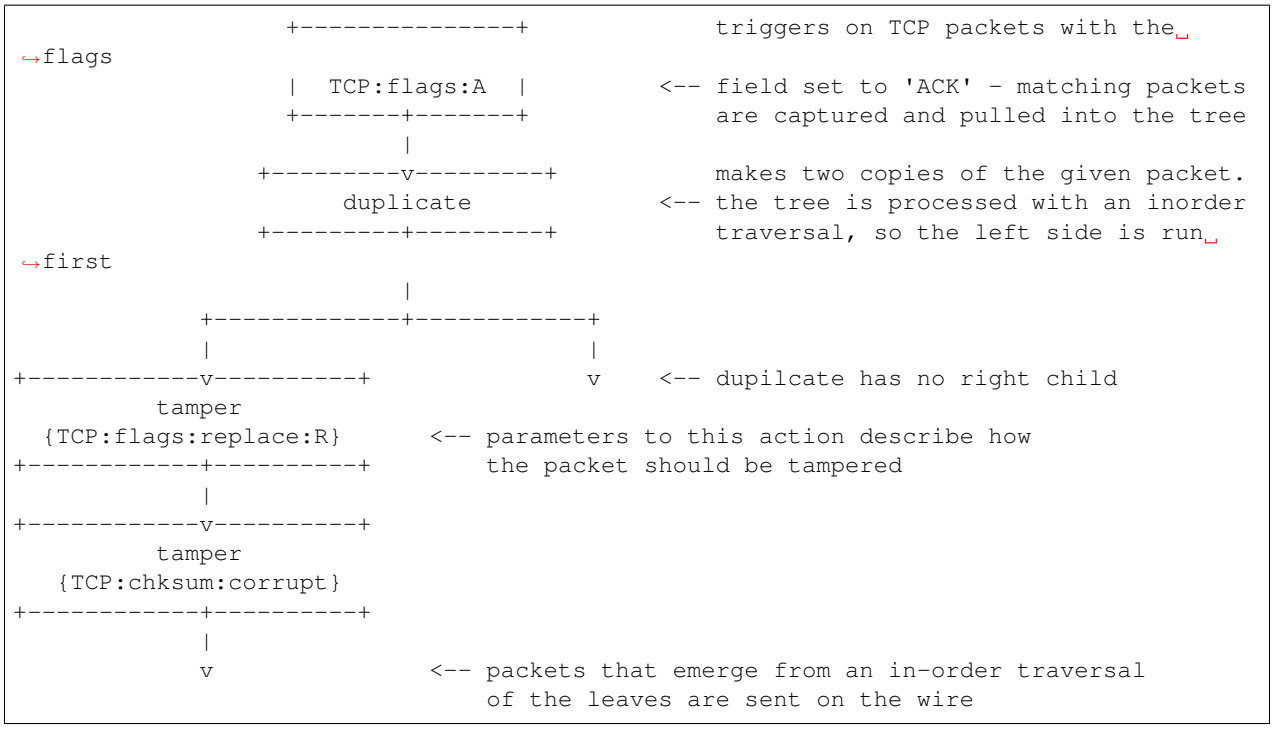
1. `duplicate`: takes one packet and returns two copies of the packet
2. `drop`: takes one packet and returns no packets (drops the packet)
3. `tamper`: takes one packet and returns the modified packet
4. `fragment`: takes one packet and returns two fragments or two segments

Since `duplicate` and `fragment` introduce *branching*, these actions are composed into a binary-tree structure called an *action tree*.

Each tree also has a *trigger*. The trigger describes which packets the tree should run on, and the tree describes what should happen to each of those packets when the trigger fires. Once a trigger fires on a packet, it pulls the packet into the tree for modifications, and the packets that emerge from the tree are sent on the wire. Recall that Geneva operates at the packet level, therefore all triggers are packet-level triggers.

Multiple action trees together form a *forest*. Geneva handles outbound and inbound packets differently, so strategies are composed of two forests: an outbound forest and an inbound forest.

Consider the following example of a simple Geneva strategy.



5.2 Strategy DNA Syntax

These strategies can be arbitrarily complicated, and Geneva defines a well-formatted string syntax for unambiguously expressing strategies.

A strategy divides how it handles outbound and inbound packets: these are separated in the DNA by a "\/". Specifically, the strategy format is `<outbound forest> \/ <inbound forest>`. If `\/` is not present in a strategy, all of the action trees are in the outbound forest.

Both forests are composed of action trees, and each forest is allowed an arbitrarily many trees.

Action trees always start with a trigger, which is formatted as: `[<protocol>:<field>:<value>]`. For example, the trigger: `[TCP:flags:S]` will run its corresponding tree whenever it sees a `TCP` packet with the `flags` field set to `SYN`. If the corresponding action tree is `[TCP:flags:S]-drop-|`, this action tree will cause the engine to drop any `SYN` packets. `[TCP:flags:S]-duplicate-|` will cause the engine to duplicate any SYN packets.

Triggers also can contain an optional 4th parameter for *gas*, which describes the number of times a trigger can fire. The triger `[IP:version:4:4]` will run only on the first 4 IPv4 packets it sees. If the gas is negative, the trigger acts as a *bomb* trigger, which means the trigger will not fire until a certain number of applicable packets have been seen. For example, the trigger `[IP:version:4:-2]` will trigger only after it has seen two matching packets (and it will not trigger on those first packets).

Syntactically, action trees end with `-|`.

Depending on the type of action, some actions can have up to two children (such as `duplicate`). These are represented with the following syntax: `[TCP:flags:S]-duplicate(<left_child>,<right_child>)-|`, where `<left_child>` and `<right_child>` themselves are trees. If `(,)` is not specified, any packets that emerge from the action will be sent on the wire. If an action only has one child (such as `tamper`), it is always the left child. `[TCP:flags:S]-tamper{<parameters>}(<left_child>,)-|`

Actions that have parameters specify those parameters within `{}`. For example, giving parameters to the `tamper` action could look like: `[TCP:flags:S]-tamper{TCP:flags:replace:A}-|`. This strategy would trigger

on TCP `SYN` packets and replace the TCP `flags` field to `ACK`.

Putting this all together, below is the strategy DNA representation of the above diagram: `[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:R}(tamper{TCP:chksum:corrupt},),)-| \/`

Geneva has code to parse this strategy DNA into strategies that can be applied to network traffic using the engine.

Note: Due to limitations of Scapy and NFQueue, actions that introduce branching (`fragment`, `duplicate`) are disabled for incoming action forests.

CHAPTER 6

Engine

The strategy engine (`engine.py`) applies a strategy to a network connection. The engine works by capturing all traffic to/from a specified port. Packets that match an active trigger are run through the associated action-tree, and packets that emerge from the tree are sent on the wire.

```
# python3 engine.py --server-port 80 --strategy "\/" --log debug
2019-10-14 16:34:45 DEBUG:[ENGINE] Engine created with strategy \/ (ID bm3kdw3r) to␣
↪port 80
2019-10-14 16:34:45 DEBUG:[ENGINE] Configuring iptables rules
2019-10-14 16:34:45 DEBUG:[ENGINE] iptables -A OUTPUT -p tcp --sport 80 -j NFQUEUE --
↪queue-num 1
2019-10-14 16:34:45 DEBUG:[ENGINE] iptables -A INPUT -p tcp --dport 80 -j NFQUEUE --
↪queue-num 2
2019-10-14 16:34:45 DEBUG:[ENGINE] iptables -A OUTPUT -p udp --sport 80 -j NFQUEUE --
↪queue-num 1
2019-10-14 16:34:45 DEBUG:[ENGINE] iptables -A INPUT -p udp --dport 80 -j NFQUEUE --
↪queue-num 2
```

The engine also has a Python API for using it in your application. It can be used as a context manager or invoked in the background as a thread. For example, consider the following simple application.

```
import os
import engine

# Port to run the engine on
port = 80
# Strategy to use
strategy =␣
↪"[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:R}(tamper{TCP:chksum:corrupt},),)-| \/"

# Create the engine in debug mode
with engine.Engine(port, strategy, log_level="debug") as eng:
    os.system("curl http://example.com?q=ultrasurf")
```

This script creates an instance of the engine with a specified strategy, and that strategy will be running for everything within the context manager. When the context manager exits, the engine will clean itself up. See the `examples/`

folder for more use cases of the engine.

Note: Due to limitations of scapy and NFQueue, the engine cannot be used to communicate with localhost.

CHAPTER 7

Evolution

Now that we have a concrete definition for a censorship evasion strategies and a way to use them, we can begin evolving new strategies with Geneva's genetic algorithm.

A genetic algorithm is a type of machine learning inspired by natural selection. Over the course of many *generations*, it optimizes a numerical *fitness* score of individuals; within each generation, individuals that have a low fitness will not survive to the next generation, and those with a high fitness score will survive and propagate.

Each individual in Geneva is a censorship evasion strategy. `evolve.py` is the main driver for Geneva's genetic algorithm and maintains the population pool of these strategies.

Each generation is comprised of the following:

1. Mutation/Crossover - randomly mutating/mating strategies in the pool
2. Strategy Evaluation - assigning a numerical fitness score to each strategy - decides which strategies should live on to the next generation
3. Selection - the selection process of which strategies should survive to the next generation

For more detail on mutation/crossover or the selection process, see our papers or the code in `evolve.py` for more detail. `evolve.py` exposes options to control hyperparameters of mutation/crossover, as well as other advanced options for rejecting mutations it has seen before.

Strategy evaluation is significantly more complex, and will be covered in depth in the next section.

For most of this documentation, we will show examples of using `evolve.py` with the `--eval-only <strategy_here>` option. This instructs `evolve.py` not to start the full genetic algorithm, but to instead perform a single strategy evaluation with the given parameters. `--eval-only` can be replaced with parameters to the genetic algorithm to start strategy evolution.

7.1 Argument Parsing

Geneva uses a pass-through system of argument parsing. Different parts of the system define which arguments they care about, and they will parse just those args out. If `--help` is used, `evolve.py` will collect the help messages for the relevant components (evaluator, plugins, etc).

7.2 Population Control

The two most important command line options for controlling evolutions are `--population` and `--generations`. These control the population size and number of generations evolution will take place for, respectively.

Geneva will not automatically halt evolution after population convergence occurs.

7.3 Seeding Population

Geneva allows you to seed the starting population pool of strategies using the `--seed <strategy>` parameter. This will create the initial population pool with nothing but copies of the given strategy. Mutation is applied to each individual before evaluation begins, so the first generation is not solely one individual being evaluated over and over again.

CHAPTER 8

Strategy Evaluation

Strategy evaluation tries to answer the question, *"which censorship evasion strategies should survive and propagate in the next generation?"* This is the job of the evaluator (`evaluator.py`). The `evaluator` assigns a numerical fitness score to each strategy based on a fitness function. The actual numerical fitness score itself is unimportant: as long as a 'better' strategy has a higher fitness score than a 'worse' strategy, our fitness function will be sound. Specific fitness numbers will be used in this section as examples, but all that matters is the _comparison_ between fitness scores.

Since the goal of Geneva is to find a strategy that evades censorship, to test a strategy, we can simply send a forbidden query through a censor with a strategy running and see if the request gets censored. Geneva operates at the network (TCP/IP) layer, so it is completely application agnostic: whether `curl` or Google Chrome is generating network traffic, the engine can capture and modify it.

8.1 Environment IDs

During each evaluation, every strategy under test is given a random identifier, called an 'environment id'. This is used to track each strategy during evaluation. As each strategy is evaluated, a log file is written out named after the environment ID. See *Logging* for more information on how logs are stored.

8.2 Plugins

To allow for evolution for many different applications, Geneva has a system of plugins (in *plugins/*). A plugin is used to drive an application to make a forbidden request, and defines the fitness function for that application. These plugins provide evaluator with a common interface to launch them and get the fitness score back.

Geneva provides some plugins with fitness functions for common protocols out of the box. Whenever the evaluator or `evolve.py` is run, you must specify which plugin it should use with `--test-type <plugin>`, such as `--test-type http`.

Plugins can define a client and optionally a server. The client will attempt to make a forbidden connection through the censor to an external server, or optionally if run from another computer, to an instance of the server plugin.

To evaluate a strategy, the evaluator will initialize the engine with the strategy, launch the plugin client, and record the fitness returned by the plugin.

See *Adding New Plugins* for more details on how plugins work, how they can override behavior in the evaluator, and how to add new ones to Geneva.

CHAPTER 9

Fitness Functions

Unlike many other machine learning scenarios, we have no gradient to learn against; censors give us only a binary signal (censored or not) to learn from. Therefore, we can't just write a fitness function that will directly guide us to the answer. Instead, we will use the fitness function to encourage the genetic algorithm to search the space of strategies that keeps the TCP connection alive.

As mentioned above, this guide might have specific fitness #s in examples, but the actual numeric values are not important - what is important is that a "bad" strategy gets a lower fitness number than a "good" strategy. We will use the fitness function to define a hierarchy of strategies.

The comparison order used by Geneva, ordered from best to worst:

- Strategy that does not get censored and generates a minimal number of packets, no unused triggers, and minimal size
- Strategy that does not get censored, but has unused actions, is too large, or imparts overhead
- Strategy that gets censored
- Strategy that does not trigger on any packets but gets censored
- Strategy that breaks the underlying TCP connection
- Empty strategy

Accomplishing this hierarchy is relatively straightforward - when evaluating a strategy, we assign a numerical fitness score such that strategies can be compared to one another and will be sorted according to this list.

This is done to guide the genetic algorithm in its search through the space of strategies. For example, consider a population pool that has 4 empty strategies, and one strategy that breaks the TCP connection. After evaluation, the empty strategies would be killed off, and the strategy that runs is propagated. When the offspring of this strategy mutate, if one of them no longer breaks the underlying TCP connection, it will be considered more fit than the other strategies, and so on.

This hierarchy accomplishes a significant *search space reduction*. Instead of Geneva fuzzing the entire space of possible strategies (for which there are many!), it instead quickly eliminates strategies that break the underlying connection and encourages the genetic algorithm to concentrate effort on only those strategies that keep the underlying connection alive.

Said another way, this hierarchy allows Geneva to differentiate between a strategy shooting ourselves in the foot, and being caught by the censor.

CHAPTER 10

Running the Evaluator

Since Geneva only needs to run on one side of the connection, Geneva is flexible as to where and how it can be run. Strategy evaluation can be done either on the server-side or the client-side. When used on the server-side, it can drive an external client to itself (or another server) for testing, or even act as a NAT-ing middlebox to interpose between the external client and true server.

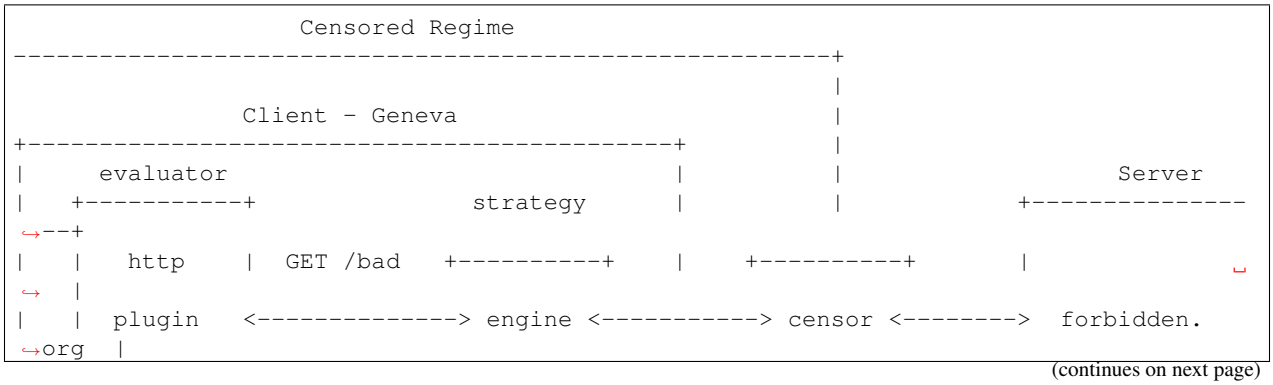
The evaluator also allows plugins to override its default logic for single strategy evaluation, or for evaluating the entire strategy pool. See *Adding New Plugins* for more detail on writing your own plugin.

10.1 Client-side Evaluation

The most common use case for Geneva is client-side evaluation. In this mode, Geneva is run at the client-side inside a censored regime, trying to make a query to a forbidden resource located outside the censored regime. For this example, we will use the HTTP plugin. The HTTP plugin creates a forbidden HTTP GET request (such as *example.com?q=ultrasurf* in China).

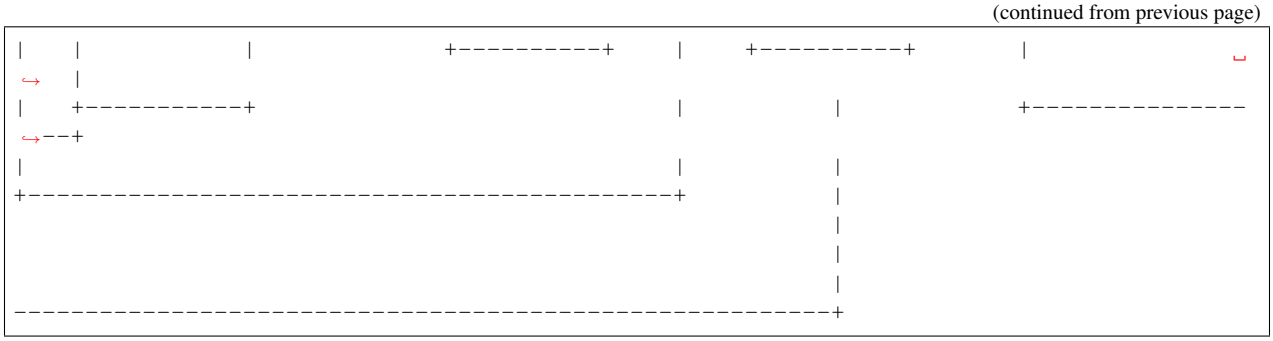
The evaluator will start the engine, launch the client http plugin (which will make a request), and then the client plugin will record if the request succeeded or not. Under the hood, the GET request generated by the client HTTP plugin is caught by the engine (which modifies the traffic according to the strategy it is running) before sending it.

This effectively looks like this:



(continues on next page)

(continued from previous page)



Note: For the purpose of this guide, examples will be given using `evolve.py`. `evolve.py` has an option `--eval-only` to run the evaluator on a strategy with given parameters and exit. As usual, `--eval-only` can also be replaced with parameters to the genetic algorithm to start the full genetic algorithm, instead of a single strategy evaluation.

To accomplish this, we can simply run:

```
# python3 evolve.py --eval-only "\/" --test-type http --server forbidden.org --log␣
↪debug
```

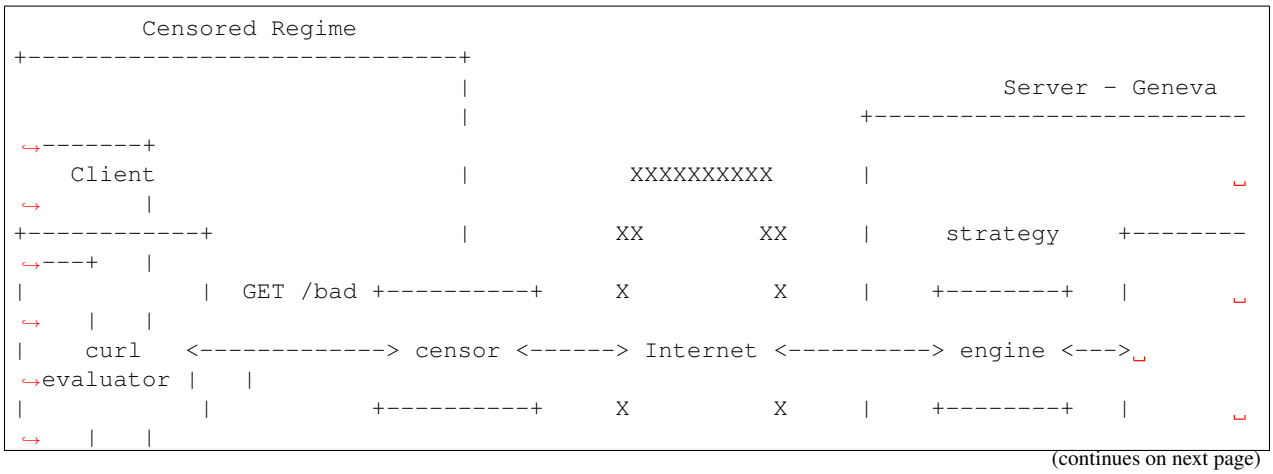
This will start `evolve.py`, which will launch the *evaluator* with the *http* plugin, configured to make a request to *forbidden.org*, in debug mode.

10.2 Server-side Evaluation

Beyond using Geneva from the client-side, we can also use it from the server-side.

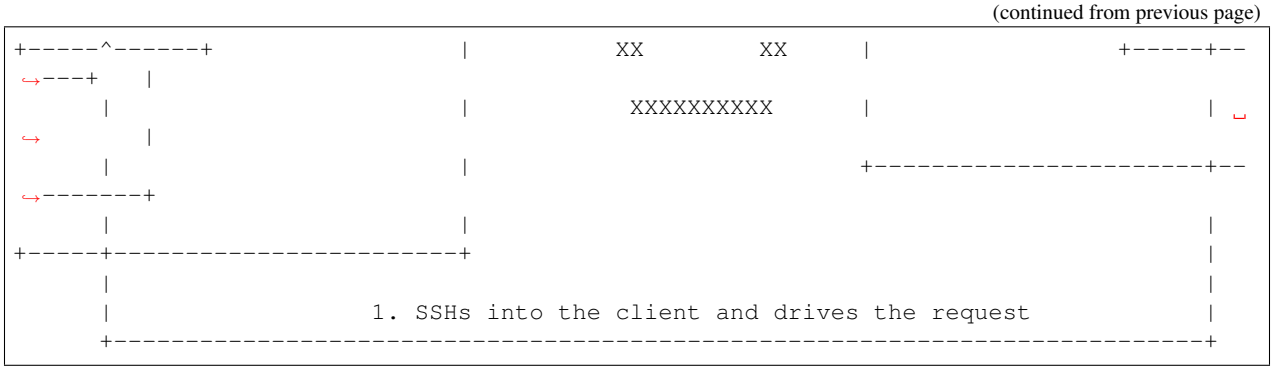
In this mode, Geneva can learn strategies that work from the server-side, subverting censorship on client's behalf. This requires a copy of Geneva to be checked out on both sides of the connection. To do this, we must specify the `--server-side` flag.

Geneva will first start up a server for the specified plugin, and start the engine with the given strategy in front of the server. It next SSHes into an external client worker located inside a censored regime and runs the specified plugin through the SSH session. That plugin will generate a forbidden request to our server, and the plugin on the external client records if the request succeeded or not; the evaluator retrieves this fitness over the SSH session and evaluation is complete. This looks like this:



(continues on next page)

(continued from previous page)



To run this with `evolve.py` to evaluate an empty strategy (`\/`), we can do:

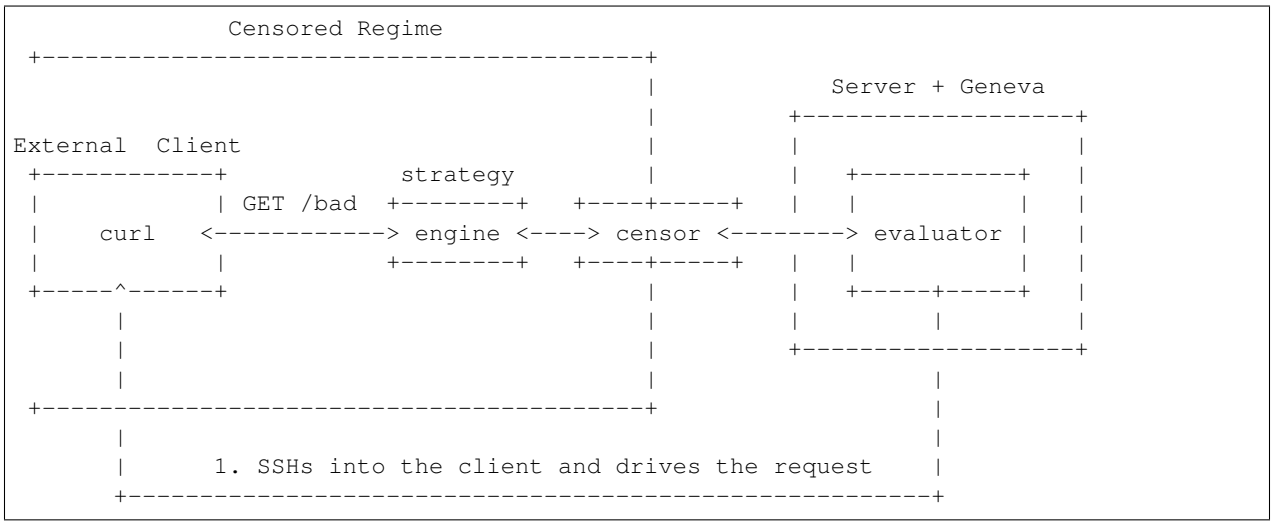
```
# python3 evolve.py --test-type http --public-ip <mypublicip> --external-client␣
↪example
  --log debug --eval-only "\/" --server-side
```

Note that we specified two additional options here: the public IP address of the test computer (used for making the request), and the external client worker we want to use.

Note: See *Adding a Worker* on how to configure external workers for Geneva.

10.2.1 External Clients with Local Servers

Alternatively, we can evaluate the strategies on the client side, but have Geneva host the server for us so we can observe the traffic on both sides of the connection. By simply not specifying the `--server-side` flag, we can accomplish this.

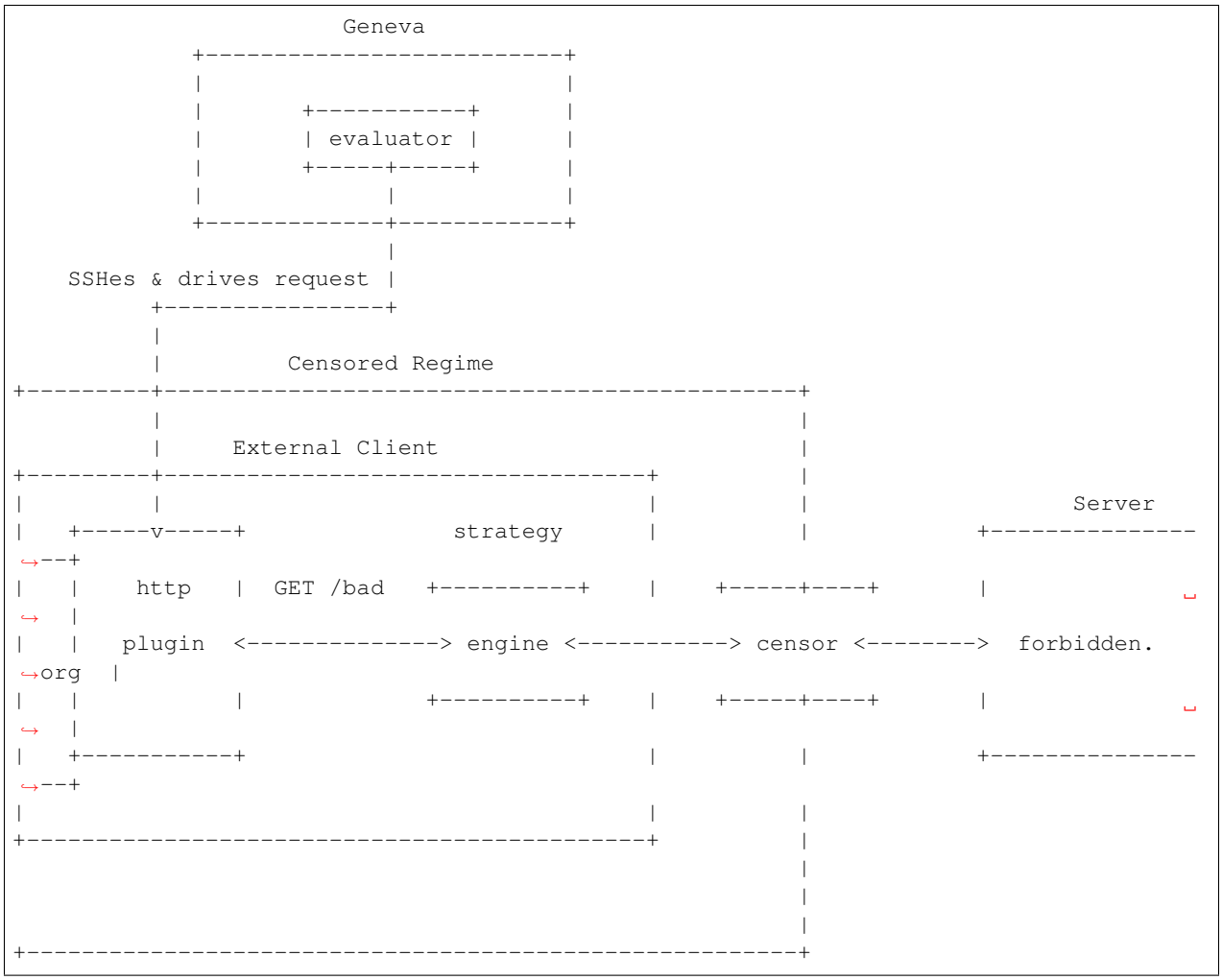



```
# python3 evolve.py --test-type http --public-ip <mypublicip> --external-client␣
↪example
  --log debug --eval-only "\/" --server-side
```

10.2.2 External Clients with External Servers

Alternatively, the client can be driven to a different server from outside the censored regime with the `'--external-server'` flag, and the server can be specified with `'--server'`:

```
# python3 evolve.py --test-type http --public-ip <mypublicip> --external-client␣
↪example
  --log debug --eval-only "\/" --external-server --server http://wikipedia.org
```



10.2.3 Engine as a Middlebox

There are many cases in which we cannot trigger a censor while communicating with servers we control. In these cases, we can use Geneva as a middlebox, and interpose between the client and server and apply the strategy in between.

To accomplish this, we will specify `--act-as-middlebox`, and specify three additional routing options:

Routing options:

- `--routing-ip`: Internal IP address of the middlebox
- `--forward-ip`: IP address we are forwarding to

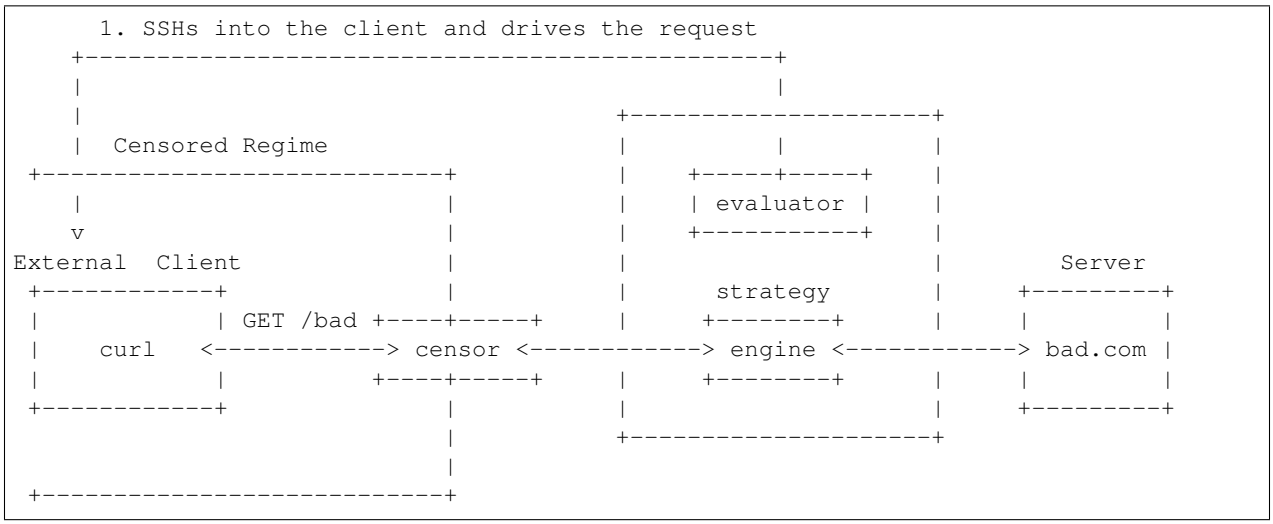
- `--sender-ip`: IP address we are forwarding from

Note: Because Geneva operates at the packet-level, it cannot be used out of the box as a general purpose NAT, as it does not do connection tracking.

In this mode, the external client will communicate directly with our IP address, which will in turn forward the packets to the specified destination after they are modified according to the strategy under evaluation.

To run the strategy engine as a middlebox and drive an external client, we can add additional routing options:

```
# python3 evolve.py --test-type http --public-ip <mypublicip> --external-client␣
↪example --log
  debug --eval-only "\/" --server-side --act-as-middlebox --routing-ip <myinternalip>
  --forward-ip <iptoforwardto> --sender-ip <ipofexternalclient>
```



Note: The `--routing-ip` is NOT the public IP address of your machine, its the internal address. This means that if you're running on an EC2 machine, the `--public-ip` is your external IP, but the `--routing-ip` is the IP you see when you run `'ifconfig'`.

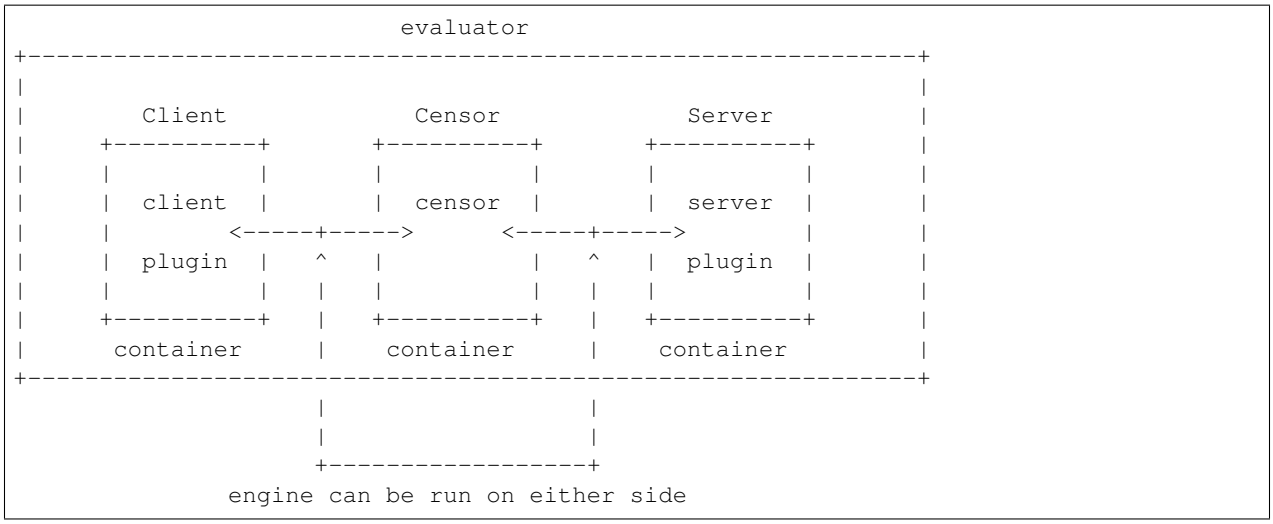
10.3 Internal Evaluation with Docker

For the purpose of internal testing and fitness function development, Geneva provides a set of simple mock censors (see `censors/`). The evaluator can be configured to train against these censors.

Note: In order to do internal evaluation against these mock censors, you must have set up Geneva's base Docker container. See *Setup* for how to do this.

When used with Docker, Geneva will spin up three docker containers: a client, a censor, and a server, and configure the networking routes such that the client and server communicate through the censor. To evaluate strategies, Geneva will run the plugin client inside the client and attempt to communicate with the server through the censor.

Each docker container used by the evaluator runs out of the same base container.



We can accomplish this by specifying the `--censor <censor>` option. This will enable running with Docker, start the censor, and perform evaluation. Depending on whether `--server-side` is specified, the engine will be run on either the client or server side.

```
# python3 evolve.py --eval-only "" --test-type echo --censor censor2 --log debug
```

Note: `--censor` cannot be used with `--external-client` or `--external-server`.

CHAPTER 11

Adding a Worker

Below is how to add a new external client worker to Geneva. An external client worker is an external, SSH-accessible machine under the control of the user running Geneva for the purpose of performing strategy evaluation from outside the censored regime.

Geneva expects each of its worker to be defined in the `workers` folder.

For this section, let us assume we are trying to allow Geneva to use a new external worker located in China.

First, make a new subfolder for that worker under the `workers/` directory.

```
# mkdir workers/test
# ls workers/
example test
```

Each worker is defined by a `worker.json` file located inside its subfolder.

The structure of the worker looks like this:

```
{
    "name": "test",
    "ip": "<ip_here>",
    "hostname": "<hostname>",
    "username": "user",
    "password": null,
    "port": 22,
    "python": "python3",
    "city": "Bejing",
    "keyfile": "example.pem",
    "country": "China",
    "geneva_path": "~/geneva"
}
```

If passwordless SSH is used, you can optionally specify a keyfile for it to SSH with.

Once this is defined, we can specify `--external-client test` during strategy evaluation, and the evaluator will SSH to this worker for training!

Note: Remember, external client workers must have Geneva cloned to the directory specified in `geneva_path` and depencies set up before use.

CHAPTER 12

Logging

Geneva uses multiple loggers during execution. `evolve.py` creates the parent logger, and creates a subfolder of the current time under the `trials/` directory.

Within this directory, it creates 5 subfolders:

- `data`: used for misc. data related to strategy evaluation
- `flags`: used to write status files to set events
- `generations`: used to store the full generations and hall of fame after each generation
- `logs`: stores logs for evaluation
- `packets`: stores packet captures during strategy evolution

The two main log files used by `evolve.py` are `ga.log` and `ga_debug.log` (everything in debug mode). As each strategy is evaluated, a `<id>_engine.log`, `<id>_server.log`, and `<id>_client.log` files are generated.

For example, one run's output could be:

```
# ls trials
2020-03-23_20:03:08

# ls trials/2020-03-23_20:03:08
data        flags       generations logs        packets

# ls trials/2020-03-23_20:03:08/logs
ga.log      ga_debug.log    zhak1n81_client.log  zhak1n81_engine.log  zhak1n81_server.
↪log
```

CHAPTER 13

Automated Tests

Geneva has a system of automated tests in the `tests/` directory, powered by pytest. Unless you are doing modifications to the source code, you can generally ignore these.

If you need to run them yourself, you can do so with:

```
# python3 -m pytest -sv tests/
```

To put the tests in debug mode, you can add `--evolve-logger debug`.

CHAPTER 14

Putting it all Together

Now that we know how to leverage the framework, this short section will give a high level of how to put it all together and run Geneva's genetic algorithm yourself.

The two most important command line options for controlling evolutions are `--population` and `--generations`. These control the population size and number of generations evolution will take place for, respectively.

Generally, these need not be very large. Geneva intentionally does strategy evaluation serially & slowly, so increasing the size dramatically will make evolution take longer. For our existing research papers, the population count rarely exceeded 300.

For example, to run a client-side evolution against HTTP censorship with a population pool of 200 and 25 generations, the following can be used:

```
# python3 evolve.py --population 200 --generations 25 --test-type http --server␣
↪forbidden.org
```

Before running any evolution, it is recommended to spot test the plugins against whatever censor is used as the adversary to confirm the fitness function properly sets up the desirable hierarchy of individuals as specified in *Fitness Functions*: strategies that break the connection get lower fitness than those that get censored, which get a lower fitness than those that succeed.

And that's it!

CHAPTER 15

Adding New Plugins

This section will describe the process to add a new application plugin to Geneva. Application plugins serve as the fitness function for Geneva during evolution, and allow it to evolve strategies to defeat certain types of censorship.

Plugins are run by the Evaluator; if you have not yet read how the Evaluator works, see the *Strategy Evaluation* section.

There are three types of plugins: clients, servers, and overriding plugins. A developer can choose to implement any one of, or all three of these plugins.

For this section, we will build an example plugin and walk through the existing plugins to tour through the plugin API.

Plugins are expected to be in the `plugins/` folder in geneva's repo. The folder name is the plugin name, and Geneva will discover these automatically. Plugins are specified to the evaluator or evolve with the `--test-type` flag. Within the plugin folder, plugins must adhere to the following naming scheme:

- `client.py` - for plugin clients
- `server.py` - for plugin servers [optional]
- `plugin.py` - for an overriding plugin to customize logic [optional]

`server.py` and `plugin.py` are optional. The server plugin is required to do server-side evaluation, but the overriding plugin definition is only required to if a developer wishes to override the evaluator's default behavior.

If an overriding plugin is provided, the evaluator will simply invoke it at the start of strategy evaluation and the overriding plugin will be responsible for calling the client and server. This section will assume that no overriding plugin is specified to describe the evaluator's default behavior with plugins, and cover use cases for overriding plugins at the end.

Depending on the evaluation setup, some (or all) of these plugins will be used during evaluation. For example, during an exclusively client-side evaluation, only the client plugin is needed.

15.1 Client Plugins

During exclusively client-side evolution, the evaluator will start the engine with the strategy under evaluation, and then run the client plugin. During server-side evolution, the evaluator will run the engine on the server-side, start the server

plugin, and then start the client plugin via an SSH session to the remote client worker. (See *Adding a Worker* on how external workers can be used).

The client plugin subclasses from the PluginClient object. To tour through the API, we will walk through the development of a custom client plugin.

15.1.1 Writing Our Own

Let us write a fitness function to test Iran's whitelisting system.

Iran's protocol whitelister was a recently deployed new censorship mechanism to censor non-whitelisted protocols on certain ports (53, 80, 443). We deployed Geneva against the whitelister, and discovered multiple ways to evade it in just one evolution of the genetic algorithm. (The results of that investigation is located here).

The whitelister worked by checking the first 2 packets of a flow, and if they did not match a fingerprint, it would destroy the flow.

In order to run Geneva against whitelister, we will define a client plugin that will try to trigger the whitelister and record whether the whitelister successfully censored its connection or not.

First, let's make a new folder in the `plugins/` directory called "whitelister". We'll create a "client.py" and create the plugin object as a subclass of ClientPlugin.

```
class WhitelisterClient(ClientPlugin):
    """
    Defines the whitelister client.
    """
    name = "whitelister"

    def __init__(self, args):
        """
        Initializes the whitelister client.
        """
        ClientPlugin.__init__(self)
        self.args = args
```

Done! Next, let's define argument parsing for this plugin. Geneva uses a pass-through system of argument parsing: when command-line arguments are specified, evolve.py parses the options it knows and passes the rest to the evaluator. The evaluator prases the options it knows, and passes the list to the plugins. This allows developers to easily add their own arguments just to their plugin and use them from the command-line without changing any of the intermediate code.

In this case, we need our client to make a TCP connection to a server located outside of Iran to send our whitelister triggering messages to. Let's add an argument so the user can specify which server to connect to.

We can do this by adding a `get_args` static method. The evaluator will call this method when the plugin is created and give it the full command line list, so the plugin is free to parse it how it chooses. For this example, we will use the standard `argparse` library.

Since the superclass also defines args, we'll pass the command line list up to the super class as well to collect those arguments.

```
@staticmethod
def get_args(command):
    """
    Defines args for this plugin
    """
    super_args = ClientPlugin.get_args(command)
```

(continues on next page)

(continued from previous page)

```
    parser = argparse.ArgumentParser(description='Whitelister Client')

    parser.add_argument('--server', action='store', help="server to connect to")

    args, _ = parser.parse_known_args(command)
    args = vars(args)

    super_args.update(args)
    return super_args
```

Now, we just need to define a `run()` method. The `run()` method is called by the evaluator to run the plugin. It provides the parsed arguments, a logger to log with, and a reference to an instance of the strategy engine that is running the strategy (see *Engine* for more information on how the engine works.)

Let's start by defining the run method. We'll pull out the argument for the server we defined earlier, connect to it with a python socket, and then just send "G", "E", and "T" in separate messages to trigger the whitelister. Since the whitelister censors connections by blackholing them, if the strategy failed to defeat the whitelister, we would expect our network connection to timeout; if we can send our messages and get a response from the server, the strategy under evaluation may have defeated the whitelister.

```
def run(self, args, logger, engine=None):
    """
    Try to open a socket, send two messages, and see if the messages
    time out.
    """
    fitness = 0
    port = int(args["port"])
    server = args["server"]
    try:
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client.settimeout(3)
        client.connect((server, port))
        client.sendall(b"G")
        time.sleep(0.25)
        client.sendall(b"E")
        time.sleep(0.25)
        client.sendall(b"T\r\n\r\n")
        server_data = client.recv(1024)
        logger.debug("Data recieved: %s", server_data.decode('utf-8', 'ignore'))
        if server_data:
            fitness += 100
        else:
            fitness -= 90
        client.close()
        # ...
```

Now we just need to define the error handling for this code. This is critical to the fitness function: we want to kill off strategies that damage the underlying TCP connection, so Geneva does not waste time searching this space of strategies.

Our goal is to set the fitness metric such that a *censorship event has a higher fitness than the strategy damaging the connection*. Since we can distinguish these cases based on the socket error, we will set a lower fitness if any other exception is raised besides the timeout.

Lastly, we'll inflate the numerical fitness metric to make it a larger number. The evaluator does additional punishments to the fitness score based on the strategy (see *Strategy Evaluation*), so we want the number to be sufficiently large to not push succeeding strategies to negative numbers.

```
    except socket.timeout:
        logger.debug("Client: Timeout")
        fitness -= 90
    except socket.error as exc:
        fitness -= 100
        logger.exception("Socket error caught in client echo test.")
    finally:
        logger.debug("Client finished whitelister test.")
return fitness * 4
```

Putting it all together:

```
class WhitelisterClient(ClientPlugin):
    """
    Defines the whitelister client.
    """
    name = "whitelister"

    def __init__(self, args):
        """
        Initializes the whitelister client.
        """
        ClientPlugin.__init__(self)
        self.args = args

    @staticmethod
    def get_args(command):
        """
        Defines args for this plugin
        """
        super_args = ClientPlugin.get_args(command)
        parser = argparse.ArgumentParser(description='Whitelister Client')

        parser.add_argument('--server', action='store', help="server to connect to")

        args, _ = parser.parse_known_args(command)
        args = vars(args)

        super_args.update(args)
        return super_args

    def run(self, args, logger, engine=None):
        """
        Try to open a socket, send two messages, and see if the messages
        time out.
        """
        fitness = 0
        port = int(args["port"])
        server = args["server"]
        try:
            client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client.settimeout(3)
            client.connect((server, port))
            client.sendall(b"G")
            time.sleep(0.25)
            client.sendall(b"E")
            time.sleep(0.25)
```

(continues on next page)

(continued from previous page)

```
            client.sendall(b"T\r\n\r\n")
            server_data = client.recv(1024)
            logger.debug("Data recieved: %s", server_data.decode('utf-8', 'ignore'))
            if server_data:
                fitness += 100
            else:
                fitness -= 90
            client.close()
        except socket.timeout:
            logger.debug("Client: Timeout")
            fitness -= 90
        except socket.error as exc:
            fitness -= 100
            logger.exception("Socket error caught in client echo test.")
        finally:
            logger.debug("Client finished whitelister test.")
    return fitness * 4
```

15.2 Server Plugins

Coming soon!

15.3 Override Plugins

Coming soon!

CHAPTER 16

Defining New Actions

It is simple to add a new packet-level action to Geneva.

Let us assume we are adding a new action, called "mytamper", which simply sets the `ipid` field of a packet. Our action will take 1 packet and return 1 packet, and we'll start by making it always set the IPID to 1.

We will subclasss the `Action` class, and specify an `__init__` method and a `run` method.

In the `__init__` method, we will specify that our action's name is 'mytamper' and can run in `both` inbound and outbound. Then, in the `run()` method, we will use Geneva's packet API to set the `ipid` field to 1 and simply return the packet.

```
from actions.action import Action

class MyTamperAction(Action):
    """
    Geneva action to set the IPID to 1.
    """
    # Controls frequency with which this action is chosen by the genetic algorithm
    # during mutation
    frequency = 0

    def __init__(self, environment_id=None):
        Action.__init__(self, "mytamper", "both")

    def run(self, packet, logger):
        """
        The mytamper action returns a modified packet as the left child.
        """
        logger.debug("  - Changing IPID field to 1")
        packet.set("IP", "ipid", 1)
        return packet, None
```

And that's it! Now, we can specify this action in our normal strategy DNA: Geneva will discover it dynamically on startup, import it, and we can use it.

16.1 Adding Parameters

Let's now assume we want to make our action take parameters. We will add two new methods: `parse()` and `__str__()`. We'll start by adding a new instance variable `self.ipid_value`.

```
def __init__(self, environment_id=None, ipid_value=1):
    Action.__init__(self, "mytamper", "both")
    self.ipid_value = ipid_value
```

Next, we'll add the `__str__` method so when our action is printed in the strategy DNA, its components are too:

```
def __str__(self):
    """
    Returns a string representation.
    """
    s = Action.__str__(self)
    s += "{%g}" % self.ipid_value
    return s
```

Finally, we'll add the `parse()` method so we can parse the value from a string strategy DNA to a live action.

```
def parse(self, string, logger):
    """
    Parses a string representation for this object.
    """
    try:
        if string:
            self.ipid_value = float(string)
    except ValueError:
        logger.exception("Cannot parse ipid_value %s" % string)
        return False

    return True
```

Putting it all together:

```
from actions.action import Action

class MyTamperAction(Action):
    """
    Geneva action to set the IPID to 1.
    """
    # Controls frequency with which this action is chosen by the genetic algorithm
    # during mutation
    frequency = 0

    def __init__(self, environment_id=None, ipid_value=1):
        Action.__init__(self, "mytamper", "both")
        self.ipid_value = ipid_value

    def run(self, packet, logger):
        """
        The mytamper action returns a modified packet as the left child.
        """
        logger.debug("  - Changing IPID field to 1")
        packet.set("IP", "ipid", 1)
        return packet, None
```

(continues on next page)

(continued from previous page)

```
    def __str__(self):
        """
        Returns a string representation.
        """
        s = Action.__str__(self)
        s += "{%g}" % self.ipid_value
        return s

    def parse(self, string, logger):
        """
        Parses a string representation for this object.
        """
        try:
            if string:
                self.ipid_value = float(string)
        except ValueError:
            logger.exception("Cannot parse ipid_value %s" % string)
            return False

        return True
```

And we're done! Now, we can write strategies like: `[TCP:flags:PA]-mytamper{10}-|`, and any TCP packet with the flags field set to `PA` will have its `ipid` field set to 10.

CHAPTER 17

Contributing

Contributions are welcome! You are encouraged to fork the repository, open Github issues with us, or just make a pull request. If you are interested in becoming more involved with the team or development, checkout our website a https://censorship.ai and drop us a line!

CHAPTER 18

geneva.engine

Geneva Strategy Engine

Given a strategy and a server port, the engine configures NFQueue to capture all traffic into and out of that port so the strategy can run over the connection.

class engine.**Engine**(*server_port*, *string_strategy*, *environment_id=None*, *server_side=False*, *output_directory='trials'*, *log_level='info'*, *file_log_level='info'*, *enabled=True*, *in_queue_num=None*, *out_queue_num=None*, *forwarder=None*, *save_seen_packets=True*, *demo_mode=False*)

Bases: `object`

__init__(*server_port*, *string_strategy*, *environment_id=None*, *server_side=False*, *output_directory='trials'*, *log_level='info'*, *file_log_level='info'*, *enabled=True*, *in_queue_num=None*, *out_queue_num=None*, *forwarder=None*, *save_seen_packets=True*, *demo_mode=False*)

Parameters

- **server_port** (*str*) – The port(s) the engine will monitor
- **string_strategy** (*str*) – String representation of strategy DNA to apply to the network
- **environment_id** (*str, None*) – ID of the given strategy
- **server_side** (*bool, False*) – Whether or not the engine is running on the server side of the connection
- **output_directory** (*str, 'trials'*) – The path logs and packet captures should be written to
- **enabled** (*bool, True*) – whether or not the engine should be started (used for conditional context managers)
- **in_queue_num** (*int, None*) – override the netfilterqueue number used for inbound packets. Used for running multiple instances of the engine at the same time. Defaults to None.

- **out_queue_num** (*int, None*) – override the netfilterqueue number used for outbound packets. Used for running multiple instances of the engine at the same time. Defaults to None.
- **save_seen_packets** (*bool, True*) – whether or not the engine should record and save packets it sees while running. Defaults to True, but it is recommended this be disabled on higher throughput systems.
- **demo_mode** (*bool, False*) – whether to replace IPs in log messages with random IPs to hide sensitive IP addresses.

configure_iptables(*remove=False*)
Handles setting up ipables for this run

delayed_send(*packet*, *delay*)
Method to be started by a thread to delay the sending of a packet without blocking the main thread.

do_nat(*packet*)
NATs packet: changes the sources and destination IP if it matches the configured route, and clears the checksums for recalculating

Parameters **packet** (*layers.packet.Packet*) – packet to modify before sending

Returns the modified packet

Return type layers.packet.Packet

handle_packet(*packet*)
Handles processing an outbound packet through the engine.

in_callback(*nfpacket*)
Callback bound to the incoming nfqueue rule. Since we can't manually send packets to ourself, process the given packet here.

initialize_nfqueue()
Initializes the nfqueue for input and output forests.

mysend(*packet*)
Helper scapy sending method. Expects a Geneva Packet input.

out_callback(*nfpacket*)
Callback bound to the outgoing nfqueue rule to run the outbound strategy.

run_nfqueue(*nfqueue*, *nfqueue_socket*, *direction*)
Handles running the outbound nfqueue socket with the socket timeout.

shutdown_nfqueue()
Shutdown nfqueue.

engine.**get_args**()
Sets up argparse and collects arguments.

engine.**main**(*args*)
Kicks off the engine with the given arguments.

CHAPTER 19

geneva.evaluator

The Evaluator is charged with evaluating a given strategy and assigning a numerical fitness metric to it.

class evaluator.**Evaluator**(*command*, *logger*)
Bases: object

__init__(*command*, *logger*)
Initialize the global evaluator for this evolution.

Parameters

- **command** (*list*) – sys.argv or list of arguments
- **logger** (logging.Logger) – logger passed in from the main driver to log from

assign_ids(*ind_list*)
Assigns random environment ids to each individual to be evaluated.

Parameters **ind_list** (*list*) – List of individuals to assign random IDs to

canary_phase(*canary*)
Learning phase runs the client against the censor to collect packets.

Parameters **canary** (actions.strategy.Strategy) – A (usually empty) strategy object to evaluate

Returns canary id used ("canary")

Return type str

create_test_environment(*worker_id*)
Creates a test environment in docker.

Parameters **worker_id** (*int*) – Worker ID of this worker

Returns Environment dictionary to use

Return type dict

dump_logs(*environment_id*)
Dumps client, engine, server, and censor logs, to be called on test failure at ERROR level.

Parameters **environment_id** (*str*) – Environment ID of a strategy to dump

evaluate(*ind_list*)
Perform the overall fitness evaluation driving.

Parameters **ind_list** (*list*) – list of individuals to evaluate

Returns Population list after evaluation

Return type list

exec_cmd(*command*, *timeout=60*)
Runs a subprocess command at the correct log level.

Parameters

- **command** (*list*) – Command to execute.
- **timeout** (*int, optional*) – Timeout for execution

exec_cmd_output(*command*, *timeout=60*)
Runs a subprocess command at the correct log level. This is a separate method from the above exec_cmd, since that is used to stream output to the screen (so check_output is not appropriate).

Parameters

- **command** (*list*) – Command to execute.
- **timeout** (*int, optional*) – Timeout for execution

Returns Output of command

Return type str

get_ip()
Gets IP of evaluator computer.

Returns Public IP provided

Return type str

get_log(*remote*, *worker*, *log_name*, *logger*)
Retrieves a log from the remote server and writes it to disk.

Parameters

- **remote** – A Paramiko SSH channel to execute over
- **worker** (*dict*) – Dictionary describing external client worker
- **log_name** (*str*) – Log name to retrieve
- **logger** (logging.Logger) – A logger to log with

get_pid(*container*)
Returns PID of first actively running python process.

Parameters **container** – Docker container object to query

Returns PID of Python process

Return type int

initialize_base_container(*name*)
Builds a base container with a given name and connects it to a given network. Also retrieves lower level settings and the IP address of the container.

Parameters **name** (*str*) – Name of this docker container

Returns Dictionary containing docker container object and relevant information

Return type dict

parse_ip(*container*, *iface*)
Helper method to parse an IP address from ifconfig.

Parameters

- **container** – Docker container object to execute within
- **iface** (*str*) – Interface to parse from

Returns IP address

Return type str

read_fitness(*ind*)
Looks for this individual's fitness file on disk, opens it, and stores the fitness in the given individual.

Parameters **ind** (actions.strategy.Strategy) – Individual to read fitness for

remote_exec_cmd(*remote*, *command*, *logger*, *timeout=15*, *verbose=True*)
Given a remote SSH session, executes a string command. Blocks until command completes, and returns the stdout and stderr. If the SSH connection is broken, it will try again.

Parameters

- **remote** – Paramiko SSH channel to execute commands over
- **command** (*str*) – Command to execute
- **logger** (logging.Logger) – A logger to log with
- **timeout** (*int, optional*) – Timeout for the command
- **verbose** (*bool, optional*) – Whether the output should be printed

Returns (stdout, stderr) of command, each is a list

Return type tuple

run_client(*args*, *environment*, *logger*)
Runs the plugin client given the current configuration

Parameters

- **args** (*dict*) – Dictionary of arguments
- **environment** (*dict*) – Dictionary describing environment configuration for this evaluation
- **logger** (logging.Logger) – A logger to log with

Returns Fitness of individual

Return type float

run_docker_client(*args*, *environment*, *logger*)
Runs client within the docker container. Does not return fitness; instead fitness is written via the flags directory and read back in later.

Parameters

- **args** (*dict*) – Dictionary of arguments
- **environment** (*dict*) – Dictionary describing environment configuration for this evaluation

- **logger** (logging.Logger) – A logger to log with

run_docker_server(*args*, *environment*, *logger*)
Runs server and censor in their respective docker containers.

Parameters

- **args** (*dict*) – Dictionary of arguments
- **environment** (*dict*) – Dictionary describing environment configuration for this evaluation
- **logger** (logging.Logger) – A logger to log with

run_local_client(*args*, *environment*, *logger*)
Runs client locally. Does not return fitness.

Parameters

- **args** (*dict*) – Dictionary of arguments
- **environment** (*dict*) – Dictionary describing environment configuration for this evaluation
- **logger** (logging.Logger) – A logger to log with

run_local_server(*args*, *environment*, *logger*)
Runs local server.

Parameters

- **args** (*dict*) – Dictionary of arguments
- **environment** (*dict*) – Dictionary describing environment configuration for this evaluation
- **logger** (logging.Logger) – A logger to log with

run_remote_client(*args*, *environment*, *logger*)
Runs client remotely over SSH, using the given SSH channel

Parameters

- **args** (*dict*) – Dictionary of arguments
- **environment** (*dict*) – Dictionary describing environment configuration for this evaluation
- **logger** (logging.Logger) – A logger to log with

Returns Fitness of individual

Return type float

run_test(*environment*, *ind*)
Conducts a test of a given individual in the environment.

Parameters

- **environment** (*dict*) – Dictionary of environment variables
- **ind** (actions.strategy.Strategy) – A strategy object to test with

Returns (ind.environment_id, ind.fitness) environment ID of strategy and fitness

Return type tuple

setup_remote()
Opens an SSH tunnel to the remote client worker.

shutdown()
Shuts down all active environments

shutdown_container(*container*)
Tries to shutdown a given container and eats a NotFound exception if the container has already exited.

Parameters **container** – docker container object to call stop() on

shutdown_environment(*environment*)
Shuts down the evaluation environment. If Docker, shuts down server and client container. If a remote SSH connection, the connection is shut down.

start_censor(*environment*, *environment_id*)
Starts the censor in the server environment container.

Parameters

- **environment** (*dict*) – Environment dictionary
- **environment_id** (*str*) – Environment ID of the censor to stop

start_server(*args*, *environment*, *logger*)
Launches the server.

Parameters

- **args** (*dict*) – Dictionary of arguments
- **environment** (*dict*) – Dictionary describing environment configuration for this evaluation
- **logger** (logging.Logger) – A logger to log with

Returns fitness of individual (if one is provided)

Return type float

stop_censor(*environment*)
Send SIGKILL to all remaining python processes in the censor container. This is done intentionally over a SIGINT or a graceful shutdown mecahnism - due to dynamics with signal handling in nfqueue callbacks (threads), SIGINTs can be ignored and graceful shutdown mechanisms may not be picked up (or be fast enough).

The output this method parses is below:

```
# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.0  21944  3376 pts/0    Ss+  13:30   0:00 /bin/bash
root          14 24.0  0.4 200376 38564 ?        Ss   13:30   0:00 python code/
↪censor_driver.py censor2 jgsko1rf trials/2018-10-30_06:30:48 60181
root          32  0.0  0.0  19188  2400 ?        Rs   13:30   0:00 ps aux
```

Parameters **environment** (*dict*) – Environment dictionary

stop_server(*environment*, *server*)
Stops server.

Parameters

- **environment** (*dict*) – Environment dictionary

- **server** (plugins.plugin_server.ServerPlugin) – A plugin server to stop

terminate_docker()
Terminates any hanging running containers.

update_ports(*environment*)
Checks that the chosen port is open inside the docker container - if not, it chooses a new port.

Parameters **environment** (*dict*) – Dictionary describing docker environment

worker(*ind_list*, *worker_id*, *environment*)
Perform the actual fitness evaluation as a multithreaded worker. The worker pops off an individual from the list and evaluates it.

Parameters

- **ind_list** (*list*) – List of strategy objects to evaluate
- **worker_id** (*int*) – ID of this worker
- **environment** (*dict*) – Environment dictionary

evaluator.**collect_plugin**(*test_plugin*, *plugin_type*, *command*, *full_args*, *plugin_args*)
Import a given plugin

Parameters

- **test_plugin** (*str*) – Plugin name to import ("http")
- **plugin_type** (*str*) – Component of plugin to import ("client")
- **command** (*list*) – sys.argv or list of arguments
- **full_args** (*dict*) – Parsed full list of arguments already maintained by the parent plugin
- **plugin_args** (*dict*) – Dictionary of args specific to this plugin component

Returns Imported plugin class for instantiation later

evaluator.**get_arg_parser**(*single_use=False*)
Sets up argparse. This is done separately to enable collection of help messages.

Parameters **single_use** (*bool, optional*) – whether this evaluator will only be used for one strategy, used to configure sane defaults

evaluator.**get_args**(*cmd*, *single_use=False*)
Creates an argparser and collects arguments.

Parameters **single_use** (*bool, optional*) – whether this evaluator will only be used for one strategy, used to configure sane defaults

Returns parsed args

Return type dict

evaluator.**get_random_open_port**()
Selects a random ephemeral port that is open.

Returns Open port

Return type int

CHAPTER 20

geneva.evolve

Main evolution driver for Geneva (GENetic EVAsion). This file performs the genetic algorithm, and relies on the evaluator (evaluator.py) to provide fitness evaluations of each individual.

evolve.**add_to_hof**(*hof*, *population*)
Iterates over the current population and updates the hall of fame. The hall of fame is a dictionary that tracks the fitness of every run of every strategy ever.

Parameters

- **hof** (*dict*) – Current hall of fame
- **population** (*list*) – Population list

Returns Updated hall of fame

Return type dict

evolve.**collect_plugin_args**(*cmd*, *plugin*, *plugin_type*, *message=None*)
Collects and prints arguments for a given plugin.

Parameters

- **cmd** (*list*) – sys.argv or a list of args to parse
- **plugin** (*str*) – Name of plugin to import ("http")
- **plugin_type** (*str*) – Component of plugin to import ("client")
- **message** (*str*) – message to override for printing

evolve.**collect_results**(*hall_of_fame*)
Collect final results from offspring.

Parameters **hall_of_fame** (*dict*) – Hall of fame of individuals

Returns Formatted printout of the hall of fame

Return type str

evolve.**driver**(*cmd*)
Main workflow driver for the solver. Parses flags and input data, and initiates solving.

Parameters **cmd** (*list*) – sys.argv or a list of arguments

Returns Hall of fame of individuals

Return type dict

evolve.**eval_only**(*logger*, *requested*, *ga_evaluator*, *runs=1*)
Parses a string representation of a given strategy and runs it through the evaluator.

Parameters

- **logger** (logging.Logger) – A logger to log with
- **requested** (*str*) – String representation of requested strategy or filename of strategies
- **ga_evaluator** (*evaluator.Evaluator*) – An evaluator to evaluate with
- **runs** (*int*) – Number of times each strategy should be evaluated

Returns Success rate of tested strategies

Return type float

evolve.**fitness_function**(*logger*, *population*, *ga_evaluator*)
Calls the evaluator to evaluate a given population of strategies. Sets the .fitness attribute of each individual.

Parameters

- **logger** (logging.Logger) – A logger to log with
- **population** (*list*) – List of individuals to evaluate
- **ga_evaluator** (*evaluator.Evaluator*) – An evaluator object to evaluate with

Returns Population post-evaluation

Return type list

evolve.**generate_strategy**(*logger*, *num_in_trees*, *num_out_trees*, *num_in_actions*, *num_out_actions*, *seed*, *environment_id=None*, *disabled=None*)
Generates a strategy individual.

Parameters

- **logger** (logging.Logger) – A logger to log with
- **num_in_trees** (*int*) – Number of trees to initialize in the inbound forest
- **num_out_trees** (*int*) – Number of trees to initialize in the outbound forest
- **num_in_actions** (*int*) – Number of actions to initialize in the each inbound tree
- **num_out_actions** (*int*) – Number of actions to initialize in the each outbound tree
- **environment_id** (*str, optional*) – Environment ID to assign to the new individual
- **disabled** (*str, optional*) – List of actions that should not be considered in building a new strategy

Returns A new strategy object

Return type actions.strategy.Strategy

evolve.**genetic_solve**(*logger*, *options*, *ga_evaluator*)
Run genetic algorithm with given options.

Parameters

- **logger** (logging.Logger) – A logger to log with

- **options** (*dict*) – Options to respect.
- **ga_evaluator** (*evaluator.Evaluator*) – Evaluator to evaluate strategies with

Returns Hall of fame of individuals

Return type dict

evolve.**get_args**(*cmd*)
Sets up argparse and collects arguments.

Parameters **cmd** (*list*) – sys.argv or a list of args to parse

Returns Parsed arguments

Return type namespace

evolve.**get_unique_population_size**(*population*)
Computes number of unique individuals in a population.

Parameters **population** (*list*) – Population list

evolve.**initialize_population**(*logger*, *options*, *canary_id*, *disabled=None*)
Initializes the population from either random strategies or strategies located in a file.

Parameters

- **logger** (logging.Logger) – A logger to log with
- **options** (*dict*) – Options to respect in generating initial population. Options that can be specified as keys:

  ”load_from” (str, optional): File to load population from population_size (int): Size of population to initialize

  ”in-trees” (int): Number of trees to initialize in inbound forest of each individual

  ”out-trees” (int): Number of trees to initialize in outbound forest of each individual

  ”in-actions” (int): Number of actions to initialize in each inbound tree of each individual

  ”out-actions” (int): Number of actions to initialize in each outbound tree of each individual

  ”seed” (str): Strategy to seed this pool with
- **canary_id** (*str*) – ID of the canary strategy, used to associate each new strategy with the packets captured during the canary phase
- **disabled** (*list, optional*) – List of actions that are disabled

Returns New population of individuals

Return type list

evolve.**load_generation**(*logger*, *filename*)
Loads strategies from a file

Parameters

- **logger** (logger.Logger) – A logger to log with
- **filename** (*str*) – Filename of file containing newline separated strategies to read generation from

evolve.**mutate_individual**(*logger*, *ind*)
Simply calls the mutate function of the given individual.

Parameters

- **logger** (logging.Logger) – A logger to log with
- **ind** (actions.strategy.Strategy) – A strategy individual to mutate

Returns Mutated individual

Return type actions.strategy.Strategy

evolve.**mutation_crossover**(*logger*, *population*, *hall*, *options*)
Apply crossover and mutation on the offspring.

Hall is a copy of the hall of fame, used to accept or reject mutations.

Parameters

- **logger** (logging.Logger) – A logger to log with
- **population** (*list*) – Population of individuals
- **hall** (*dict*) – Current hall of fame
- **options** (*dict*) – Options to override settings. Accepted keys are: "crossover_pb" (float): probability of crossover "mutation_pb" (float): probability of mutation "allowed_retries" (int): number of times a strategy is allowed to exist in the hall of fame. "no_reject_empty" (bool): whether or not empty strategies should be rejected

Returns New population after mutation

Return type list

evolve.**print_results**(*hall_of_fame*, *logger*)
Prints hall of fame.

Parameters

- **hall_of_fame** (*dict*) – Hall of fame to print
- **logger** (logging.Logger) – A logger to log results with

evolve.**restrict_headers**(*logger*, *protos*, *filter_fields*, *disabled_fields*)
Restricts which protocols/fields can be accessed by the algorithm.

Parameters

- **logger** (logging.Logger) – A logger to log with
- **protos** (*str*) – Comma separated string of protocols that are allowed
- **filter_fields** (*str*) – Comma separated string of fields to allow
- **disabled_fields** (*str*) – Comma separated string of fields to disable

evolve.**run_collection_phase**(*logger*, *ga_evaluator*)
Individual mutation works best when it has seen real packets to base action and trigger values off of, instead of blindly fuzzing packets. Usually, the 0th generation is useless because it hasn't seen any real packets yet, and it bases everything off fuzzed data. To combat this, a canary phase is done instead.

In the canary phase, a single dummy individual is evaluated to capture packets. Once the packets are captured, they are associated with all of the initial population pool, so all of the individuals have some packets to base their data off of.

Since this phase by necessity requires the evaluator, this is only run if –no-eval is not specified.

Parameters

- **logger** (logging.Logger) – A logger to log with
- **ga_evaluator** (*evaluator.Evaluator*) – An evaluator object to evaluate with

Returns ID of the test 'canary' strategy evaluated to do initial collection

Return type str

evolve.**sel_random**(*individuals*, *k*)
Implementation credit to DEAP: https://github.com/DEAP/deap Select *k* individuals at random from the input *individuals* with replacement. The list returned contains references to the input *individuals*.

Parameters

- **individuals** (*list*) – A list of individuals to select from.
- **k** (*int*) – The number of individuals to select.

Returns A list of selected individuals.

Return type list

evolve.**selection_tournament**(*individuals*, *k*, *tournsize*, *fit_attr='fitness'*)
Implementation credit to DEAP: https://github.com/DEAP/deap Select the best individual among *tournsize* randomly chosen individuals, *k* times. The list returned contains references to the input *individuals*.

Parameters

- **individuals** (*list*) – A list of individuals to select from.
- **k** (*int*) – The number of individuals to select.
- **tournsize** (*int*) – The number of individuals participating in each tournament.
- **fit_attr** – The attribute of individuals to use as selection criterion (defaults to "fitness")

Returns A list of selected individuals.

Return type list

evolve.**setup_logger**(*log_level*)
Sets up the logger. This will log at the specified level to "ga.log" and at debug level to "ga_debug.log". Logs are stored in the trials/ directory under a run-specific folder. Example: trials/2020-01-01_01:00:00/logs/ga.log

Parameters **log_level** (*str*) – Log level to use in setting up the logger ("debug")

evolve.**write_generation**(*filename*, *population*)
Writes the population pool for a specific generation.

Parameters

- **filename** (*str*) – Name of file to write the generation out to
- **population** (*list*) – List of individuals to write out

evolve.**write_hall**(*filename*, *hall_of_fame*)
Writes hall of fame out to a file.

Parameters

- **filename** (*str*) – Filename to write results to
- **hall_of_fame** (*dict*) – Hall of fame of individuals

CHAPTER 21

geneva.actions.action

Geneva superclass object for defining a packet-level action.

class action.**Action**(*action_name*, *direction*)

Bases: object

Defines the superclass for a Geneva Action.

__init__(*action_name*, *direction*)

Initializes this action object.

Parameters

- **action_name** (*str*) – Name of this action (“duplicate”)
- **direction** (*str*) – Direction of this action (“out”, “both”, “in”)

applies(*direction*)

Returns whether this action applies to the given direction, as branching actions are not supported on inbound trees.

Parameters **direction** (*str*) – Direction to check if this action applies (“out”, “in”, “both”)

Returns whether or not this action can be used to a given direction

Return type bool

static get_actions(*direction*, *disabled=None*, *allow_terminal=True*)

Dynamically imports all of the Action classes in this directory.

Will only return terminal actions if terminal is set to True.

Parameters

- **direction** (*str*) – Limit imported actions to just those that can run to this direction (“out”, “in”, “both”)
- **disabled** (*list, optional*) – list of actions that are disabled
- **allow_terminal** (*bool*) – whether or not terminal actions (“drop”) should be imported

Returns Dictionary of imported actions

Return type dict

mutate(*environment_id=None*)
Mutates packet.

static parse_action(*str_action*, *direction*, *logger*)
Parses a string action into the action object.

Parameters

- **str_action** (*str*) – String representation of an action to parse
- **direction** (*str*) – Limit actions searched through to just those that can run to this direction ("out", "in", "both")
- **logger** (logging.Logger) – a logger to log with

Returns A parsed action object

Return type *action.Action*

frequency = 0

ident = 0

CHAPTER 22

geneva.actions.drop

class drop.**DropAction**(*environment_id=None*)

Bases: actions.action.Action

Geneva action to drop the given packet.

__init__(*environment_id=None*)

Initializes this drop action.

Parameters **environment_id**(*str, optional*) – Environment ID of the strategy we are a part of

run(*packet*, *logger*)

The drop action returns None for both it's left and right children, and does not pass the packet along for continued use.

frequency = 1

CHAPTER 23

geneva.actions.duplicate

class duplicate.**DuplicateAction**(*environment_id=None*)

Bases: actions.action.Action

Defines the DuplicateAction - returns two copies of the given packet.

__init__(*environment_id=None*)

Initializes this action object.

Parameters

- **action_name** (*str*) – Name of this action ("duplicate")
- **direction** (*str*) – Direction of this action ("out", "both", "in")

mutate(*environment_id=None*)

Swaps its left and right child

run(*packet*, *logger*)

The duplicate action duplicates the given packet and returns one copy for the left branch, and one for the right branch.

frequency = 3

CHAPTER 24

geneva.actions.fragment

class fragment.**FragmentAction**(*environment_id=None*, *correct_order=None*, *fragsize=-1*, *segment=True*, *overlap=0*)

Bases: actions.action.Action

Defines the FragmentAction for Geneva - fragments or segments the given packet.

__init__(*environment_id=None*, *correct_order=None*, *fragsize=-1*, *segment=True*, *overlap=0*)

Initializes a fragment action object.

Parameters

- **environment_id** (*str, optional*) – Environment ID of the strategy this object is a part of
- **correct_order** (*bool, optional*) – Whether or not the fragments/segments should be returned in the correct order
- **fragsize** (*int, optional*) – The index this packet should be cut. Defaults to -1, which cuts it in half.
- **segment** (*bool, optional*) – Whether we should perform fragmentation or segmentation
- **overlap** (*int, optional*) – How many bytes the fragments/segments should overlap

fragment(*original*, *fragsize*)

Fragments a packet into two, given the size of the first packet (0:fragsize) Always returns two packets

get_rand_order()

Randomly decides if the fragments should be reversed.

ip_fragment(*packet*, *logger*)

Perform IP fragmentation.

mutate(*environment_id=None*)

Mutates the fragment action - it either chooses a new segment offset, switches the packet order, and/or changes whether it segments or fragments.

parse (*string*, *logger*)
Parses a string representation of fragmentation. Nothing particularly special, but it does check for a the fragsize.

Note that the given logger is a DIFFERENT logger than the logger passed to the other functions, and they cannot be used interchangeably. This logger is attached to the main GA driver, and is run outside the evaluator. When the action is actually run, it's run within the evaluator, which by necessity must pass in a different logger.

run (*packet*, *logger*)
The fragment action fragments each given packet.

tcp_segment (*packet*, *logger*)
Segments a packet into two, given the size of the first packet (0:fragsize) Always returns two packets, since fragment is a branching action, so if we are unable to segment, it will duplicate the packet.

If overlap is specified, it will select n bytes from the second packet and append them to the first, and increment the sequence number accordingly

frequency = 2

CHAPTER 25

geneva.layers.layer

CHAPTER 26

geneva.layers.packet

CHAPTER 27

geneva.actions.sleep

class sleep.**SleepAction**(*time=1*, *environment_id=None*)

Bases: actions.action.Action

Defines the SleepAction - causes the engine to pause before sending a packet.

__init__(*time=1*, *environment_id=None*)

Initializes the sleep action.

Parameters

- **time** (*float*) – How much time the packet should delay before sending
- **environment_id** (*str, optional*) – Environment ID of the strategy this action is a part of

parse(*string*, *logger*)

Parses a string representation for this object.

run(*packet*, *logger*)

The sleep action simply passes along the packet it was given with an instruction for how long the engine should sleep before sending it.

frequency = 0

CHAPTER 28

geneva.actions.strategy

class strategy.**Strategy**(*in_actions*, *out_actions*, *environment_id=None*)

Bases: object

__init__(*in_actions*, *out_actions*, *environment_id=None*)

Initialize self. See help(type(self)) for accurate signature.

act_on_packet(*packet*, *logger*, *direction='out'*)

Runs the strategy on a given scapy packet.

init_from_scratch(*num_in_trees*, *num_out_trees*, *num_in_actions*, *num_out_actions*, *disabled=None*)

Initializes this individual by drawing random actions.

initialize(*logger*, *num_in_trees*, *num_out_trees*, *num_in_actions*, *num_out_actions*, *seed*, *disabled=None*)

Initializes a new strategy object randomly.

mutate(*logger*)

Top level mutation function for a strategy. Simply mutates the out and in trees.

mutate_dir(*trees*, *direction*, *logger*)

Mutates a list of trees. Requires the direction the tree operates on (in or out).

pretty_print()

pretty_str_forest(*forest*)

Returns a string representation of a given forest (inbound or outbound)

run_on_packet(*packet*, *logger*, *direction*)

Runs the strategy on a given packet given the forest direction.

str_forest(*forest*)

Returns a string representation of a given forest (inbound or outbound)

strategy.**do_mate**(*forest1*, *forest2*)

Performs mating between two given forests (lists of trees). With 80% probability, a random tree from each forest are mated, otherwise, a random tree is swapped between them.

strategy.**mate**(*ind1*, *ind2*, *indpb*)
Executes a uniform crossover that modify in place the two individuals. The attributes are swapped according to the *indpb* probability.

strategy.**swap_one**(*forest1*, *forest2*)
Swaps a random tree from forest1 and forest2.

It picks a random element within forest1 and a random element within forest2, chooses a random index within each forest, and inserts the random element

CHAPTER 29

geneva.actions.tamper

TamperAction

One of the four packet-level primitives supported by Geneva. Responsible for any packet-level modifications (particularly header modifications). It supports the following primitives: - no operation: it returns the packet given - replace: it changes a packet field to a fixed value - corrupt: it changes a packet field to a randomly generated value each time it is run - add: adds a given value to the value in a field - compress: performs DNS decompression on the packet (if applicable)

class tamper.**TamperAction**(*environment_id=None*, *field=None*, *tamper_type=None*, *tamper_value=None*, *tamper_proto='TCP'*)

Bases: actions.action.Action

Defines the TamperAction for Geneva.

__init__(*environment_id=None*, *field=None*, *tamper_type=None*, *tamper_value=None*, *tamper_proto='TCP'*)

Creates a tamper object.

Parameters

- **environment_id** (*str, optional*) – environment_id of a previously run strategy, used to find packet captures
- **field** (*str, optional*) – field that the object will tamper. If not set, all the parameters are chosen randomly
- **tamper_type** (*str, optional*) – primitive this tamper will use ("corrupt")
- **tamper_value** (*str, optional*) – value to tamper to
- **tamper_proto** (*str, optional*) – protocol we are tampering

mutate(*environment_id=None*)

Mutate can switch between the tamper type, field.

parse(*string*, *logger*)

Parse out a given string representation of this action and initialize this action to those parameters.

Note that the given logger is a DIFFERENT logger than the logger passed to the other functions, and they cannot be used interchangeably. This logger is attached to the main GA driver, and is run outside the evaluator. When the action is actually run, it's run within the evaluator, which by necessity must pass in a different logger.

run (*packet*, *logger*)
The tamper action runs its tamper procedure on the given packet, and returns the edited packet down the left branch.

Nothing is returned to the right branch.

tamper (*packet*, *logger*)
Edits a given packet according to the action settings.

frequency = 5

CHAPTER 30

geneva.actions.trace

class trace.**TraceAction**(*start_ttl=1*, *end_ttl=64*, *environment_id=None*)

Bases: actions.action.Action

The Trace Action is used to TTL probe/traceroute a censor. When the action fires, it sends the captured packet with increasing ttls within a certain range

TraceAction is an experimental action that is never used in actual evolution

__init__(*start_ttl=1*, *end_ttl=64*, *environment_id=None*)

Initializes the trace action.

Parameters

- **start_ttl** (*int*) – Starting TTL to use
- **end_ttl** (*int*) – TTL to end with
- **environment_id** (*str, optional*) – Environment ID associated with the strategy we are a part of

parse(*string*, *logger*)

Parses a string representation for this object.

run(*packet*, *logger*)

The trace action sends the captured packet repeatedly with increasing ttl probes defined between the range of start_ttl to end_ttl. This is an experimental action, and is not used for training.

frequency = 0

CHAPTER 31

geneva.actions.tree

Defines an action tree. Action trees are comprised of a trigger and a tree of actions.

exception tree.**ActionTreeParseError**
Bases: Exception

Exception thrown when an action tree is malformed or cannot be parsed.

class tree.**ActionTree**(*direction*, *trigger=None*)
Bases: object

Defines an ActionTree for the Geneva system.

__init__(*direction*, *trigger=None*)
Creates this action tree.

Parameters

- **direction** (*str*) – Direction this tree is facing ("out", "in")
- **trigger** (actions.trigger.Trigger) – Trigger to use with this tree

add_action(*new_action*)
Adds an action to this action tree.

check(*packet*, *logger*)
Checks if this action tree should run on this packet.

choose_one()
Picks a random element in the tree.

contains(*action*)
Checks if an action is contained in the tree.

count_leaves()
Counts the number of leaves.

do_parse(*node*, *string*, *logger*)
Handles the preorder recursive parsing.

do_run(*node*, *packet*, *logger*)
Handles recursively running a packet down the tree.

get_parent(*node*)
Returns the parent of the given node and direction of the child.

get_rand_action(*direction*, *request=None*, *allow_terminal=True*, *disabled=None*)
Retrieves and initializes a random action that can run in the given direction.

get_slots()
Returns the number of locations a new action could be added.

initialize(*num_actions*, *environment_id*, *allow_terminal=True*, *disabled=None*)
Sets up this action tree with a given number of random actions. Note that the returned action trees may have less actions than num_actions if terminal actions are used.

mate(*other_tree*)
Mates this tree with another tree.

mutate()
Mutates this action tree with respect to a given direction.

parse(*string*, *logger*)
Parses a string representation of an action tree into this object.

preorder(*node*)
Yields a preorder traversal of the tree.

pretty_print(*visual=False*)
Pretty prints the tree.

pretty_print_help(*root*, *visual=False*, *parent=None*)

Pretty prints the tree.

- root is the highest-level node you wish to start printing
- [visual] controls whether a png should be created, by default, this is false.
- [parent] is an optional parameter specifying the parent of a given node, should only be used by this function.

Returns the root with its children as an anytree node.

remove_action(*action*)
Removes a given action from the tree.

remove_one()
Removes a random leaf from the tree.

run(*packet*, *logger*)
Runs a packet through the action tree.

string_repr(*node*)
Yields a preorder traversal of the tree to create a string representation.

swap(*my_donation*, *other_tree*, *other_donation*)
Swaps a node in this tree with a node in another tree.

CHAPTER 32

geneva.actions.trigger

class trigger.**Trigger**(*trigger_type*, *trigger_field*, *trigger_proto*, *trigger_value=0*, *environment_id=None*, *gas=None*)

Bases: object

__init__(*trigger_type*, *trigger_field*, *trigger_proto*, *trigger_value=0*, *environment_id=None*, *gas=None*)

Params:

- trigger_type: the type of trigger. Only "field" (matching based on a field value) is currently supported.
- trigger_field: the field the trigger should check in a packet to trigger on
- trigger_proto: the protocol the trigger should look in to retrieve the trigger field
- trigger_value: the value in the trigger_field that, upon a match, will cause the trigger to fire
- environment_id: environment_id the current trigger is running under. Used to retrieve previously saved packets
- gas: how many times this trigger can fire before it stops triggering. gas=None disables gas (unlimited triggers.)
- has_wildcard: represents if the trigger will match a specific value, or any value containing trigger_value

add_gas(*gas*)

Adds gas to this trigger, gas is an integer

disable_gas()

Disables the use of gas.

enable_gas()

Sets gas to 0

static get_gas()

Returns a random value for gas for this trigger.

static get_rand_trigger(*environment_id*, *real_packet_probability*)
Creates a random trigger.

is_applicable(*packet*, *logger*)
Checks if this trigger is applicable to a given packet.

mutate(*environment_id*, *real_packet_probability=0.5*)
Mutates this trigger object by picking a new protocol, field, and value.

static parse(*string*)
Given a string representation of a trigger, define a new Trigger represented by this string.

set_gas(*gas*)
Sets the gas to the specified value

CHAPTER 33

geneva.actions.utils

exception utils.**SkipStrategyException**(*msg*, *fitness*)
Bases: Exception

Raised to signal that this strategy evaluation should be cut off.

__init__(*msg*, *fitness*)
Creates the exception with the fitness to pass back

class utils.**CustomAdapter**(*logger*, *extras*)
Bases: logging.LoggerAdapter

Used for demo mode, to change sensitive IP addresses where necessary. Can be used (mostly) like a regular logger.

__init__(*logger*, *extras*)
Initialize the adapter with a logger and a dict-like object which provides contextual information. This constructor signature allows easy stacking of LoggerAdapters, if so desired.

You can effectively pass keyword arguments as shown in the following example:

adapter = LoggerAdapter(someLogger, dict(p1=v1, p2="v2"))

critical(*msg*, **args*, ***kwargs*)
Print a critical message, uses logger.critical.

debug(*msg*, **args*, ***kwargs*)
Print a debug message, uses logger.debug.

error(*msg*, **args*, ***kwargs*)
Print an error message, uses logger.error.

get_ip(*ip*)
Lookup the assigned random IP for a given real IP. If no random IP exists, a new one is created and a message is logged indicating it.

info(*msg*, **args*, ***kwargs*)
Print an info message, uses logger.info.

process(*msg*, *args*, *kwargs*)
Modify the log message to replace any instance of an IP in msg or args with its assigned random IP.

warning(*msg*, **args*, ***kwargs*)
Print a warning message, uses logger.warning.

regex = re.compile('\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}')

class utils.**Logger**(*log_dir*, *logger_name*, *log_name*, *environment_id*, *log_level='DEBUG'*)
Bases: object

Logging class context manager, as a thin wrapper around the logging class to help handle closing open file descriptors.

__init__(*log_dir*, *logger_name*, *log_name*, *environment_id*, *log_level='DEBUG'*)
Initialize self. See help(type(self)) for accurate signature.

utils.**build_command**(*args*)
Given a dictionary of arguments, build it back into a command line string.

utils.**close_logger**(*logger*)
Closes open file handles for a given logger.

utils.**get_console_log_level**()
returns log level of console handler

utils.**get_from_fuzzed_or_real_packet**(*environment_id*, *real_packet_probability*, *enable_options=True*, *enable_load=True*)
Retrieves a protocol, field, and value from a fuzzed or real packet, depending on the given probability and if given packets is not None.

utils.**get_id**()
Returns a random ID

utils.**get_interface**()
Chooses an interface on the machine to use for socket testing.

utils.**get_logger**(*basepath*, *log_dir*, *logger_name*, *log_name*, *environment_id*, *log_level='DEBUG'*, *file_log_level='DEBUG'*, *demo_mode=False*)
Configures and returns a logger.

utils.**get_plugins**()
Iterates over this current directory to retrieve plugins.

utils.**get_worker**(*name*, *logger*)
Returns information dictionary about a worker given its name.

utils.**import_plugin**(*plugin*, *side*)
Imports given plugin. :param - plugin: plugin to import (e.g. “http”) :param - side: which side of the connection should be imported (“client” or “server”)

utils.**parse**(*requested_trees*, *logger*)
Parses a string representation of a solution into its object form.

utils.**punish_complexity**(*fitness*, *logger*, *ind*)
Reduces fitness based on number of actions - optimizes for simplicity.

utils.**punish_fitness**(*fitness*, *logger*, *eng*)
Adjusts fitness based on additional optimizer functions.

utils.**punish_unused**(*fitness*, *logger*, *ind*)
Punishes strategy for each action that was not run.

utils.**read_packets**(*environment_id*)
Reads the pcap file associated with the last evaluation of this strategy. Returns a list of Geneva Packet objects.

utils.**setup_dirs**(*output_dir*)
Sets up Geneva folder structure.

utils.**string_to_protocol**(*protocol*)
Converts string representations of scapy protocol objects to their actual objects. For example, "TCP" to the scapy TCP object.

utils.**write_fitness**(*fitness*, *output_path*, *eid*)
Writes fitness to disk.

CHAPTER 34

geneva.plugins.dns

Client

Run by the evaluator, tries to make a GET request to a given server

class plugins.dns.client.**DNSClient**(*args*)
Bases: plugins.plugin_client.ClientPlugin

Defines the DNS client.

__init__(*args*)
Initializes the DNS client.

dns_test(*to_lookup*, *dns_server*, *output_dir*, *environment_id*, *logger*, *timeout=3*, *use_tcp=False*)
Makes a DNS query to a given censored domain.

static get_args(*command*)
Defines required args for this plugin

run(*args*, *logger*, *engine=None*)
Try to make a forbidden DNS query.

name = 'dns'

Code influenced from: - https://github.com/emileaben/scapy-dns-ninja/blob/master/dns-ninja-server.py - https://thepacketgeek.com/scapy-p-09-scapy-and-dns/

class plugins.dns.server.**DNSServer**(*args*, *logger=None*)
Bases: plugins.plugin_server.ServerPlugin

Purpose: Handle incoming DNS queries and respond with resource records defined in a zone configuration file (if exists for that domain) or respond with the answer given by a DNS resolver

Features: - Loads zone configuration files (–zones-dir) - Forwards DNS requests to a DNS resolver for domains that it does not know the answer to (–dns-resolver) - DNS forwarding can be disabled with (–no-forwarding) - Can act as the authority server for all DNS responses

Zones: - Support for A, MX, NS, TXT and CNAME - Other records may be automatically supported through the default action (no special case) - Only the first string per TXT record will be retrieved to avoid duplicated quotes

Logging: - Logs are created for each run and saved in the directory specified (–log-dir) - Logs can be disabled with (–no-log)

Python Test: tests/test_dns_server.py

__init__(*args*, *logger=None*)
Initializes the DNS Server.

build_dns_response(*packet*)
Build the DNS response packet using one of the following methods: 1) Load the resource record(s) from a manually configured DNS zone file (if exists) OTHERWISE, if enabled: 2) Send a DNS query to a DNS resolver and copy the DNS resource records

build_response_packet(*listener_packet*, *raw_socket=True*)
Build the DNS response packet - If raw_socket is enabled include the Network and Transport Layer

forward_dns_query(*packet: scapy.layers.inet.IP*)
Forwards the DNS query to a real DNS resolver and returns the DNS response

get_args()
Sets up argparse and collects arguments.

get_dns_query_info(*packet: scapy.layers.inet.IP*)
Extract information from the DNS query

get_resource_records(*domain_name*, *question_name*, *question_type*)
Gets the appropriate resource record loaded earlier from the zone file

load_zones()
Loads the DNS Zones in the zones directory specified (zones_dir)

process_packet_netfilter(*listener_packet*)
Callback function for each packet received by netfilter

run(*args*, *logger*)
Starts the DNS Service

stop()
Stops this server.

name = 'dns'

netfilter_queue = 'netfilterqueue'

socket_TCP = 'socket_TCP'

socket_UDP = 'socket_UDP'

plugins.dns.server.**main**(*args*)
Run the DNS server

DNS Plugin driver

Overrides the default evaluator plugin handling so we can check for legit IPs for UDP tests.

class plugins.dns.plugin.**DNSPluginRunner**(*args*)
Bases: plugins.plugin.Plugin

Defines the DNS plugin runner.

__init__(*args*)
Marks this plugin as enabled

check_legit_ip(*ip*, *logger*, *domain='facebook'*)
Helper method to check if the given IP address is serving web content.

static get_args(*command*)
Defines required global args for this plugin

start(*args*, *evaluator*, *environment*, *ind*, *logger*)
Runs the plugins

name = 'dns'

CHAPTER 35

geneva.plugins.echo

Client

Run by the evaluator, echo's data back and forth to the server

class plugins.echo.client.**EchoClient**(*args*)
Bases: plugins.plugin_client.ClientPlugin

Defines the Echo client.

__init__(*args*)
Initializes the echo client.

static get_args(*command*)
Defines required args for this plugin

run(*args*, *logger*, *engine=None*)
Try to make a forbidden GET request to the server.

name = 'echo'

class plugins.echo.server.**EchoServer**(*args*)
Bases: plugins.plugin_server.ServerPlugin

Defines the Echo client.

__init__(*args*)
Initializes the Echo client.

static get_args(*command*)
Defines arguments for this plugin

get_request(*control_socket*)
Get a request from the socket.

run(*args*, *logger*)
Initializes the Echo server.

stop()
Stops this server.

```
    name = 'echo'
```

CHAPTER 36

geneva.plugins.http

Run by the evaluator, tries to make a GET request to a given server

class plugins.http.client.**HTTPClient**(*args*)
Bases: plugins.plugin_client.ClientPlugin

Defines the HTTP client.

__init__(*args*)
Initializes the HTTP client.

static get_args(*command*)
Defines required args for this plugin

run(*args*, *logger*, *engine=None*)
Try to make a forbidden GET request to the server.

name = 'http'

class plugins.http.server.**HTTPServer**(*args*)
Bases: plugins.plugin_server.ServerPlugin

Defines the HTTP client.

__init__(*args*)
Initializes the HTTP client.

static get_args(*command*)
Defines arguments for this plugin

run(*args*, *logger*)
Initializes the HTTP server.

stop()
Stops this server.

name = 'http'

HTTP Plugin driver

Overrides the default evaluator plugin handling so we can negotiate a clear port and track jailed sites to avoid residual censorship.

class plugins.http.plugin.**HTTPPluginRunner**(*args*)
Bases: plugins.plugin.Plugin

Defines the HTTP plugin runner.

__init__(*args*)
Marks this plugin as enabled

static get_args(*command*)
Defines required global args for all plugins

negotiate_clear_port(*args*, *evaluator*, *environment*, *logger*)
Since residual censorship might be affecting our IP/port combo that was randomly chosen, this method is to find a port on which no residual censorship is present. This is done simply by picking a port, running the server, running curl to confirm it's accessible, and then returning that port.

start(*args*, *evaluator*, *environment*, *ind*, *logger*)
Runs the plugins

name = 'http'

class plugins.http.plugin.**TestServer**(*requested_site*, *evaluator*, *environment*, *logger*)
Bases: object

Context manager to retrieve a test server from the external server pool.

__init__(*requested_site*, *evaluator*, *environment*, *logger*)
Initialize self. See help(type(self)) for accurate signature.

plugins.http.plugin.**check_censorship**(*site*, *evaluator*, *environment*, *logger*)
Make a request to the given site to test if it is censored. Used to test a site for residual censorship before using it.

CHAPTER 37

geneva.plugins.plugin

class plugin.**Plugin**

Bases: object

Defines superclass for application plugins.

override_evaluation = False

CHAPTER 38

geneva.plugins.plugin_client

class plugin_client.**ClientPlugin**
Bases: plugins.plugin.Plugin

Defines superclass for each application plugin.

__init__()
Initialize self. See help(type(self)) for accurate signature.

static get_args(*command*)
Defines required global args for all plugins

start(*args*, *logger*)
Runs this plugin.

wait_for_censor(*serverip*, *port*, *environment_id*, *log_dir*)
Sends control packets to the censor for up to 20 seconds until it's ready.

plugin_client.**main**(*command*)
Used to invoke the plugin client from the command line.

CHAPTER 39

geneva.plugins.plugin_server

class plugin_server.**ServerPlugin**
Bases: plugins.plugin.Plugin

Defines superclass for each application plugin.

__init__()
Initialize self. See help(type(self)) for accurate signature.

static get_args(*command*)
Defines required global args for all plugins

punish_fitness(*fitness*, *logger*)
Punish fitness.

start(*args*, *logger*)
Runs this plugin.

start_thread(*args*, *logger*)
Calls the given run function, designed to be run in a separate process.

stop()
Terminates the given process.

wait_for_server(*args*, *logger*)
Waits for server to startup - returns when the server port is bound to by the server.

wait_for_shutdown(*args*, *logger*)
Checks for the <eid>.server_shutdown flag to shutdown this server.

write_startup_file(*args*, *logger*)
Writes a flag file to disk to signal to the evaluator it has started up

plugin_server.**main**(*command*)
Used to invoke the server plugin from the command line.

CHAPTER 40

geneva.plugins.sni

Runs an SNI request, confirms the connection was not torn down

class plugins.sni.client.**SNIClient**(*args*)

Bases: plugins.plugin_client.ClientPlugin

Defines the SNI client.

__init__(*args*)

Initializes the sni client.

static get_args(*command*)

Defines required args for this plugin

run(*args*, *logger*, *engine=None*)

Try to make a forbidden SNI request to the server.

name = 'sni'

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W