Honeycomb Plugins Documentation

Release 0.1.1

Cymmetria

Feb 15, 2019

Contents

This is the plugin repository for Honeycomb, the honeypot framework by Cymmetria.

CHAPTER 1

Service API Reference

1.1 honeycomb.servicemanager.base_service module

Custom Service implementation from MazeRunner.

class honeycomb.servicemanager.base_service.**DockerService**(**args*, ***kwargs*)

Bases: *honeycomb.servicemanager.base_service.ServerCustomService*

Provides an ability to run a Docker container that will be monitored for events.

docker_image_name

Return docker image name.

docker_params

Return a dictionary of docker run parameters.

See also:

Docker run: https://docs.docker.com/engine/reference/run/

Returns Dictionary, e.g., `dict(ports={80: 80})`

get_lines()

Fetch log lines from the docker service.

Returns A blocking logs generator

on_server_shutdown()

Stop the container before shutting down.

on_server_start()

Service run loop function.

Run the desired docker container with parameters and start parsing the monitored file for alerts.

parse_line(*line*)

Parse line and return dictionary if its an alert, else None / {}.

read_lines(*file_path*, *empty_lines=False*, *signal_ready=True*)
Fetch lines from file.

In case the file handler changes (logrotate), reopen the file.

Parameters

- **file_path** – Path to file
- **empty_lines** – Return empty lines
- **signal_ready** – Report signal ready on start

class honeycomb.servicemanager.base_service.**ServerCustomService**(*alert_types: list*, *service_args: dict = {}*)

Bases: multiprocessing.context.Process

Custom Service Class.

This class provides a basic wrapper for honeycomb (and mazerunner) services.

add_alert_to_queue(*alert_dict*)
Log alert and send to integrations.

alert_types = None
List of alert types, parsed from config.json

alerts_queue = None

emit(***kwargs*)
Send alerts to logfile.

Parameters **kwargs** – Fields to pass to honeycomb.decoymanager.models.Alert

logger = <Logger honeycomb.servicemanager.base_service (DEBUG)>
Logger to be used by plugins and collected by main logger.

on_server_shutdown()
Shutdown function of the server.

Override this and take care to gracefully shut down your service (e.g., close files)

on_server_start()
Service run loop function.

The service manager will call this function in a new thread.

Note: Must call *signal_ready()* after finishing configuration

---

run()
Daemon entry point.

run_service()
Run the service and start an alert processing queue.

See also:

Use *on_server_start()* and *on_server_shutdown()* for starting and shutting down your service

service_args = None
Validated dictionary of service arguments (see: `honeycomb.utils.plugin_utils.parse_plugin_args()`)

signal_ready()
Signal the service manager this service is ready for incoming connections.

thread_server = None

CHAPTER 2

Integration API Reference

2.1 honeycomb.integrationmanager.integration_utils module

Honeycomb Integration Manager.

class honeycomb.integrationmanager.integration_utils.**BaseIntegration**(*integration_data*)
Bases: object

Base Output Integration Class.

format_output_data(*output_data*)
Process and format the output_data returned by *send_event()* before display.

This is currently only relevant for MazeRunner, if you don't return an output this should return output_data without change.

Parameters **output_data** – As returned by *send_event()*

Return type dict

Returns MazeRunner compatible UI output.

Raises **IntegrationOutputFormatError** – If there's a problem formatting the output data.

poll_for_updates(*integration_output_data*)
Poll external service for updates.

If service has enabled polling, this method will be called periodically and should act like *send_event()*

Parameters **integration_output_data** – Output data returned by previous *send_event()* or *poll_for_updates()*

Returns See *send_event()*

Raises **IntegrationPollEventError** – If there's a problem polling for updates.

send_event(*alert_dict*)
Send alert event to external integration.

Parameters **alert_dict** – A dictionary with all the alert fields.

Return type tuple(dict(output_data), object(output_file))

Raises

- **IntegrationSendEventError** – If there's a problem sending the event.
- **IntegrationMissingRequiredFieldError** – If a required field is missing.

Returns A tuple where the first value is a dictionary with information to display in the UI, and the second is an optional file to be attached. If polling is enabled, the returned output_data will be passed to *poll_for_updates()*. If your integration returns nothing, you should return ({}, None).

test_connection(*integration_data*)

Perform a test to ensure the integration is configured correctly.

This could include testing authentication or performing a test query.

Parameters **integration_data** – Integration arguments.

Returns *success*

Return type tuple(bool(success), str(response))

CHAPTER 3

Honeycomb Commands Reference

3.1 Honeycomb

Honeycomb is a honeypot framework.

```
Honeycomb [OPTIONS] COMMAND [ARGS]...
```

Options

-H, --home <home>
Honeycomb home path [default: /home/docs/.config/honeycomb]

--iamroot
Force run as root (NOT RECOMMENDED!)

-c, --config <config>
Path to a honeycomb.yml file that provides instructions

-v, --verbose
Enable verbose logging

--version
Show the version and exit.

Environment variables

DEBUG

Provide a default for *--verbose*

3.1.1 integration

Honeycomb integration commands.

```
Honeycomb integration [OPTIONS] COMMAND [ARGS]...
```

configure

Configure an integration with default parameters.

You can still provide one-off integration arguments to `honeycomb.commands.service.run()` if required.

```
Honeycomb integration configure [OPTIONS] INTEGRATION [ARGS]...
```

Options

-e, --editable
Load integration directly from unspecified path without installing (mainly for dev)

-a, --show_args
Show available integration arguments

Arguments

INTEGRATION
Required argument

ARGS
Optional argument(s)

install

Install a honeycomb integration from the online library, local path or zipfile.

```
Honeycomb integration install [OPTIONS] [INTEGRATIONS]...
```

Arguments

INTEGRATIONS
Optional argument(s)

list

List integrations.

```
Honeycomb integration list [OPTIONS]
```

Options

-r, --remote
Include available integrations from online repository

show

Show detailed information about a package.

```
Honeycomb integration show [OPTIONS] INTEGRATION
```

Options

-r, --remote
Show information only from remote repository

Arguments

INTEGRATION
Required argument

test

Execute the integration's internal test method to verify it's working as intended.

```
Honeycomb integration test [OPTIONS] [INTEGRATIONS]...
```

Options

-e, --editable
Run integration directly from specified path (main for dev)

Arguments

INTEGRATIONS
Optional argument(s)

uninstall

Uninstall a integration.

```
Honeycomb integration uninstall [OPTIONS] [INTEGRATIONS]...
```

Options

-y, --yes
Don't ask for confirmation of uninstall deletions.

Arguments

INTEGRATIONS
Optional argument(s)

3.1.2 service

Honeycomb service commands.

```
Honeycomb service [OPTIONS] COMMAND [ARGS]...
```

install

Install a honeypot service from the online library, local path or zipfile.

```
Honeycomb service install [OPTIONS] [SERVICES]...
```

Arguments

SERVICES
Optional argument(s)

list

List services.

```
Honeycomb service list [OPTIONS]
```

Options

-r, --remote
Include available services from online repository

logs

Show logs of daemonized service.

```
Honeycomb service logs [OPTIONS] SERVICES...
```

Options

-n, --num <num>
Number of lines to read from end of file [default: 10]

-f, --follow
Follow log output

Arguments

SERVICES
Required argument(s)

run

Load and run a specific service.

```
Honeycomb service run [OPTIONS] SERVICE [ARGS]...
```

Options

-d, --daemon
Run service in daemon mode

-e, --editable
Load service directly from specified path without installing (mainly for dev)

-a, --show-args
Show available service arguments

-i, --integration <integration>
Enable an integration

Arguments

SERVICE
Required argument

ARGS
Optional argument(s)

show

Show detailed information about a package.

```
Honeycomb service show [OPTIONS] SERVICE
```

Options

-r, --remote
Show information only from remote repository

Arguments

SERVICE
Required argument

status

Show status of installed service(s).

```
Honeycomb service status [OPTIONS] [SERVICES]...
```

Options

-a, --show-all
Show status for all services

Arguments

SERVICES
Optional argument(s)

stop

Stop a running service daemon.

```
Honeycomb service stop [OPTIONS] SERVICE
```

Options

-e, --editable
Load service directly from specified path without installing (mainly for dev)

Arguments

SERVICE
Required argument

test

Execute the service's internal test method to verify it's working as intended.

If there's no such method, honeycomb will attempt to connect to the port listed in config.json

```
Honeycomb service test [OPTIONS] [SERVICES]...
```

Options

-f, --force
Do not check if service is running before testing

-e, --editable
Run service directly from specified path (main for dev)

Arguments

SERVICES
Optional argument(s)

uninstall

Uninstall a service.

```
Honeycomb service uninstall [OPTIONS] [SERVICES]...
```

Options

-y, --yes
Don't ask for confirmation of uninstall deletions.

Arguments

SERVICES
Optional argument(s)

CHAPTER 4

Services

4.1 Banner

4.1.1 services.banner.banner_service module

Honeycomb Banner Service.

class services.banner.banner_service.**BannerRequestHandler**(*request*, *client_address*, *server*)

Bases: socketserver.StreamRequestHandler

Request handler for banner service.

alert = None

banner = None

handle()

Handle all requests by sending out our banner.

class services.banner.banner_service.**BannerService**(**args*, ***kwargs*)

Bases: base_service.ServerCustomService

Simple service that will print out banner and hang.

on_server_shutdown()

Stop banner service.

on_server_start()

Start banner service.

test()

Test service alerts and return a list of triggered event types.

services.banner.banner_service.**service_class**

alias of *services.banner.banner_service.BannerService*

4.2 Drupal

4.2.1 services.drupal.drupal_server module

A Drupal CMS server based on Python's HTTPServer.

class services.drupal.drupal_server.**DrupalServer**(*logger*, *alert*)
Bases: object

Drupal CMS honeypot.

start()
Start serving requests by starting the underlying HTTP server.

stop()
Stop serving requests.

class services.drupal.drupal_server.**HoneyHTTPRequestHandler**(**args*, *directory=None*, ***kwargs*)
Bases: http.server.SimpleHTTPRequestHandler, object

Filter requests to catch Drupalgeddon 2 exploit attempts.

do_GET()
Handle an HTTP GET request.

do_POST()
Handle an HTTP POST request.

log_error(*message*, **args*)
Log an error.

log_message(*level*, *message*, **args*)
Send message to logger with standard apache format.

log_request(*code='-'*, *size='-'*)
Log an incoming request.

verify(*query*)
Filter HTTP request to make sure it's not an exploit attempt.

version_string()
Return the web server name that we run on.

class services.drupal.drupal_server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
Bases: socketserver.ThreadingMixIn, http.server.HTTPServer

Extend both classes to have threading capabilities.

4.2.2 services.drupal.drupal_service module

4.3 HP Officejet

4.3.1 services.hp_officejet.hp_officejet_server module

HP OfficeJet Server Module.

class services.hp_officejet.hp_officejet_server.**PJLCommandHandler**(*request*, *client_address*, *server*)

Bases: socketserver.BaseRequestHandler

PJL Command Requesrt Handler.

alert(**args*, ***kwargs*)
Raise alert.

handle()
Handle a PJL request.

handle_command(*command*, *address*)
Handle PJL Command.

class services.hp_officejet.hp_officejet_server.**PJLServer**(*alert_callback*, *logger*)

Bases: socketserver.ThreadingMixIn, socketserver.TCPServer

PJL Server class.

start()
Start PJL Server.

stop()
Stop PJL Server.

4.3.2 services.hp_officejet.hp_officejet_service module

4.4 Intel AMT

4.4.1 services.intel_amt.intel_amt_service module

Intel AMT Honeycomb Service.

class services.intel_amt.intel_amt_service.**AMTServerHandler**(**args*, *directory=None*, ***kwargs*)

Bases: http.server.SimpleHTTPRequestHandler

Intel AMT Request Handler.

do_GET()
Handle a GET Request.

server_version = 'Intel(R) Active Management Technology 2.6.3'

translate_path(*path*)
Copy of translate_path but instead of start from current directory, change to the dir of the file.

version_string()
HTTP Server version header.

class services.intel_amt.intel_amt_service.**AMTService**(**args*, ***kwargs*)

Bases: base_service.ServerCustomService

Intel AMT Honeycomb Service.

on_server_shutdown()
Shut down gracefully.

on_server_start()
Initialize service.

test()
Trigger service alerts and return a list of triggered event types.

services.intel_amt.intel_amt_service.**service_class**
alias of *services.intel_amt.intel_amt_service.AMTService*

4.5 Micros

4.5.1 services.micros.micros_server module

Micros honeycomb server module.

class services.micros.micros_server.**MicrosHandler**(**args*, *directory=None*, ***kwargs*)
Bases: http.server.SimpleHTTPRequestHandler

Micros Request Handler.

alert_function = None

db_info = '0a10000000100018000000a07371334971355054746632 6b427353486170706c6963617469

do_GET()
Process GET requests.

Provide static content, replacing dynamic tokens.

do_POST()
Process POST request.

Examine the request to ensure it follows expected protocol answer predefined queries.

handle_one_request()
Handle a single HTTP request.

Overriden to not send 501 errors

listening_port = None

log_list = '0c2000000010002900000138726638505 06e7946747866727536657768747 4703a2f2f7363

log_message(*format*, **args*)
Log a request.

logger = None

micros_info = '0a10000000100018000000845556515070397 87a66697056536e4c756170706c6963617

poc_suf2 = '001dd1021cc1021ee102'

poc_suf_1_1 = '0A100000001000180000'

poc_suf_1_2 = '6170706C69636174696F6E2F6F637465742D73747265616D01E11E0200000360000003

poc_suf_1_3 = '00000006000000'

poc_suf_1_4 = '000000240024'

poc_suf_1_ses = '66497a3263516c56444c35305045356e'

protocol_version = 'HTTP/1.1'

send_file(*filepath*)
Send a file from the mock filesystem.

setup()
Set up request handler.

version_string()
HTTP Server version header.

4.5.2 services.micros.micros_service module

4.6 Mirai Worm Monitor

4.6.1 services.mirai_worm_monitor.custom_pool module

Mirai Worm Gevent Pool.

class services.mirai_worm_monitor.custom_pool.**CustomPool**(*logger*, *size=0*, *greenlet_class=None*)
Bases: gevent.pool.Pool

An extension of the gevent pool.

If this pool becomes full, it drops the oldest connections instead of waiting for them to end.

add(*greenlet*)
Add the greenlet to the pool.

log_pool_info()
Debug log pool info.

remove_connection(*to_del_source*)
Remove connection from pool.

4.6.2 services.mirai_worm_monitor.mirai_worm_monitor_service module

4.7 Simple HTTP

4.7.1 services.simple_http.simple_http_service module

Simple HTTP Honeycomb Service.

class services.simple_http.simple_http_service.**HoneyHTTPRequestHandler**(**args*, *directory=None*, ***kwargs*)
Bases: http.server.SimpleHTTPRequestHandler, object

Simple HTTP Request Handler.

log_error(*msg*, **args*)
Log an error.

log_message(*level*, *msg*, **args*)
Send message to logger with standard apache format.

log_request(*code='-'*, *size='-'*)
Log a request.

send_head(**args*, ***kwargs*)
Handle every request by raising an alert.

server_version = 'nginx'

version_string()
HTTP Server version header.

class services.simple_http.simple_http_service.**SimpleHTTPService**(**args*, ***kwargs*)

Bases: base_service.ServerCustomService

Simple HTTP Honeycomb Service.

alert(*request*)
Raise an alert.

httpd = None

on_server_shutdown()
Shut down gracefully.

on_server_start()
Initialize Service.

test()
Test service alerts and return a list of triggered event types.

class services.simple_http.simple_http_service.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

Bases: socketserver.ThreadingMixIn, http.server.HTTPServer

Threading HTTP Server stub class.

services.simple_http.simple_http_service.**service_class**
alias of *services.simple_http.simple_http_service.SimpleHTTPService*

4.8 WebLogic

4.8.1 services.weblogic.weblogic_server module

Oracle WebLogic Honeycomb Module.

class services.weblogic.weblogic_server.**WebLogicHandler**(**args*, *directory=None*, ***kwargs*)

Bases: http.server.SimpleHTTPRequestHandler

Oracle WebLogic Request Handler.

EXPLOIT_STRING = b'</void>'

GENERIC_RESPONSE = '<?xml version=\'1.0\' encoding=\'UTF-8\'?><S:Envelope xmlns:S="http

PATCHED_RESPONSE = '<?xml version=\'1.0\' encoding=\'UTF-8\'?><S:Envelope xmlns:S="http

alert_function = None

basepath = '/home/docs/checkouts/readthedocs.org/user_builds/honeycomb-plugins/checkou

do_POST()
Handle a POST request, looking for exploit attempts.

handle_one_request()
Handle a single HTTP request.

Overriden to not send 501 errors

log_message(*format*, **args*)
Log request.

logger = None

protocol_version = 'HTTP/1.1'

send_file(*filename*, *status_code=200*)
Send file from mock filesystem.

send_head()
Return a file object that do_HEAD/GET will use.

do_GET/HEAD are already implemented by SimpleHTTPRequestHandler.

setup()
Set up request handler.

version_string()
HTTP Server version header.

4.8.2 services.weblogic.weblogic_service module

4.9 Xerox

4.9.1 services.xerox.common_strings module

String consts for Xerox service.

4.9.2 services.xerox.pjl_server module

4.9.3 services.xerox.web_server module

4.9.4 services.xerox.xerox_servers module

4.9.5 services.xerox.xerox_service module

CHAPTER 5

Integrations

5.1 Cuckoo

5.1.1 integrations.cuckoo.integration module

Honeycomb Cuckoo Integration.

class integrations.cuckoo.integration.**CuckooIntegration**(*integration_data*)
Bases: integrationmanager.integration_utils.BaseIntegration

CuckooIntegration.

format_output_data(*output_data*)
format_output_data.

get_instance_base_url(*api=True*)
get_instance_base_url.

poll_for_updates(*integration_output_data*)
poll_for_updates.

send_event(*required_alert_fields*)
send_event.

test_connection(*data*)
test_connection.

integrations.cuckoo.integration.**IntegrationActionsClass**
alias of *integrations.cuckoo.integration.CuckooIntegration*

5.2 JSON File

5.2.1 integrations.json_file.integration module

Honeycomb JSON integration.

integrations.json_file.integration.**IntegrationActionsClass**
alias of *integrations.json_file.integration.JsonIntegration*

class integrations.json_file.integration.**JsonIntegration**(*integration_data*)
Bases: integrationmanager.integration_utils.BaseIntegration

Honeycomb JSON integration.

format_output_data(*output_data*)
No special formatting needed.

send_event(*alert_fields*)
Write event to JSON file.

5.3 MISP

5.3.1 integrations.misp.integration module

Honeycomb MISP integration.

integrations.misp.integration.**IntegrationActionsClass**
alias of *integrations.misp.integration.MISPIntegration*

class integrations.misp.integration.**MISPIntegration**(*integration_data*)
Bases: integrationmanager.integration_utils.BaseIntegration

Honeycomb MISP integration.

format_output_data()
No special formatting needed.

misp = None
MISP instance.

misp_dict = {'MD5': [('add_hashes', 'md5')], 'additional_fields': '', 'domain': ['
A list of methods to call on event. Methods are tuples of (method_name, value_kwarg).

send_event(*alert_dict*)
Send MISP event.

PyMISP parameters are passed directly to requests. The *ssl* parameter can be either True/False to control requests.Session.verify, but can also be a path to CA cert file

See also:

http://docs.python-requests.org/en/master/user/advanced/#ssl-cert-verification

test_connection(*data*)
Test connectivity to MISP and fetch details about server.

Parameters are passed directly to PyMISP which in turn passes them to requests. The *ssl* parameter can be either True/False to control verify_ssl but can also be a path to CA cert file .. seealso:: http://docs.python-requests.org/en/master/user/advanced/#ssl-cert-verification

5.4 S3

5.4.1 integrations.s3.integration module

5.5 Syslog

5.5.1 integrations.syslog.integration module

Honeycomb Syslog integration.

class integrations.syslog.integration.**CEFCustomString**(*field_name: str*, *field_label: str*, *field_label_text: str*)

Bases: *integrations.syslog.integration.CEFField*

Custom CEF Field.

class integrations.syslog.integration.**CEFField**(*field_name: str*)

Bases: object

Generic CEF Field.

integrations.syslog.integration.**IntegrationActionsClass**

alias of *integrations.syslog.integration.SyslogIntegration*

class integrations.syslog.integration.**MySysLogHandler**(*address*, *facility=1*, *socktype=<SocketKind.SOCK_DGRAM: 2>*, *ssl_enabled=False*)

Bases: logging.handlers.SysLogHandler

Custom Syslog logging handler that includes CEFEvent.

For some reason python SysLogHandler appends x00 byte to every record sent, This fixes it and replaces it with n.

close()

Close the socket.

emit(*record*)

Emit a record.

The record is formatted, and then sent to the syslog server. If exception information is present, it is NOT sent to the server.

class integrations.syslog.integration.**SyslogIntegration**(*integration_data*)

Bases: integrationmanager.integration_utils.BaseIntegration

Honeycomb Syslog integration.

format_output_data(*output_data*)

No special formatting required.

get_formatted_alert_as_cef(*result_fields*)

Format message as CEFEvent.

get_formatted_alert_as_syslog(*result_fields*)

Convert alert to syslog record.

send_event(*required_alert_fields*)

Send syslog event.

CHAPTER 6

Writing your first plugin

Using simple_http as an example to accompany this guide, we will describe the 4 steps necessary to write a plugin. Feel free to use the provided config.json as a base for your own, and modify fields as required. It is recommended, for the sake of organization, that you create a new directory and follow this guide inside your specific plugin's directory.

Note: If you're looking for the full documentation for Honeycomb API look at *base_service* and *integration_utils*

CHAPTER 7

Plugin configuration - config.json

The config.json file describes the possible parameters your service can receive, and alerts it can emit. simple_http's config.json looks like this:

Listing 1: config.json

```
{
    "event_types": [
        {
            "name": "simple_http",
            "label": "HTTP Server Interaction",
            "fields": ["originating_ip", "originating_port", "request"],
            "policy": "Alert"
        }
    ],
    "service": {
        "allow_many": false,
        "supported_os_families": "All",
        "ports": [
        ],
        "name": "simple_http",
        "label": "Simple HTTP Server",
        "description": "Simple HTTP Server that alerts on every request",
        "conflicts_with": []
    },
    "parameters": [
      {
          "type": "integer",
          "value": "port",
          "label": "Listening Port",
          "required": true
      },
      {
          "type": "text",
          "value": "version",
```

(continues on next page)

(continued from previous page)

```
            "label": "Server version (header)",
            "required": true,
            "default": "nginx"
        },
        {
            "type": "boolean",
            "value": "threading",
            "label": "Enable threading support",
            "required": true,
            "default": false
        }
    ]
}
```

The *event_types* field describes alerts. This is the most important part of the configuration, as it's the way Honeycomb detects and logs suspicious events. There can be multiple alerts for each honeypot, as long as each alert is described by this structure.

Let's break down the structure:

name This is the internal identifier of the alert. Your python script should emit an alert matching *name* in order for it to be recognized and formatted.

label Human-readable name of the alert. This is the description of the alert.

fields An alert can take any number of parameters and output them when it triggers. This describes the parameters it takes.

policy This can be "Alert" or "Mute", for future use.

Next, we'll look at the *service* field. It describes the service generally and is used to avoid conflicts between honeypots that run simultaneously:

allow_many Allow multiple instances of this honeypot?

supported_os_families This prevents OS-specific honeypots from being installed on the wrong system. Current valid values are "Linux", "Windows", "Darwin", and "All".

ports Any ports this honeypot uses. For simple_http, you would expect port 80, but the service actually takes its port as a parameter.

name Internal service name.

label Human readable name.

description Full fledged description of the service.

conflicts_with Specific honeypots that this one conflicts with for whatever reason. You don't have to fill this field in, but if you know of conflicts you should.

And finally, the *parameters* field describes optional and non-optional parameters that your service can receive. Each parameter is described as follows:

type The json type of the parameter.

value Parameter name.

label Parameter description.

required Set to *true* if parameter is mandatory, or *false* if optional.

default Default value.

CHAPTER 8

Honeypot logic

8.1 Filename

Create a python file and name it *(honeypot_name)_service.py*. For example: *simple_http_service.py*.

8.2 Imports

Add the following import at the top of your service module:

```
from base_service import ServerCustomService
```

8.3 Plugin logic

Create your plugin by defining a class that inherits from *base_service.ServerCustomService*, for example:

```
class SimpleHTTPService(ServerCustomService):
```

We will address most of *ServerCustomService*'s API but make sure to also review its documentation for additional help. For example, it contains its own logger which is configured to record logs accross the framework.

8.4 Entry and exit

Your entry point will be the *on_server_start()* method. If you need an exit and cleanup point, that's *on_server_shutdown()*.

Listing 1: SimpleHTTPService.on_server_start

```
    def on_server_start(self):
        """Initialize Service."""
        os.chdir(os.path.join(os.path.dirname(__file__), "www"))
        requestHandler = HoneyHTTPRequestHandler
        requestHandler.alert = self.alert
        requestHandler.logger = self.logger
        requestHandler.server_version = self.service_args.get("version", DEFAULT_
↪SERVER_VERSION)

        port = self.service_args.get("port", DEFAULT_PORT)
        threading = self.service_args.get("threading", False)
        if threading:
            self.httpd = ThreadingHTTPServer(("", port), requestHandler)
        else:
            self.httpd = HTTPServer(("", port), requestHandler)

        self.signal_ready()
        self.logger.info("Starting {}Simple HTTP service on port: {}".format(
↪"Threading " if threading else "", port))
        self.httpd.serve_forever()
```

Listing 2: SimpleHTTPService.on_server_shutdown

```
    def on_server_shutdown(self):
        """Shut down gracefully."""
        if self.httpd:
            self.httpd.shutdown()
            self.logger.info("Simple HTTP service stopped")
            self.httpd = None
```

Note: *on_server_start()* **must** call *signal_ready()* to let the framework know it has successfully initialized and started working.

In simple_http, once we call *on_server_shutdown()*, execution flows into an infinite loop and so we must call *on_server_shutdown()* beforehand.

8.5 Parameters

If your service receives parameters, you can access them via service_args, supplying it with the *parameter value* from before. For example, in simple_http:

```
port = self.service_args.get('port', DEFAULT_PORT)
```

8.6 Connecting the plugin

Your *__main__* should consist of only one line:

```
service_class = (your_plugin_class_name)
```

For example, in simple_http:

```
service_class = SimpleHTTPService
```

8.7 Reporting alerts

The last vital stage in writing a useful plugin for Honeycomb is making it actually trigger alerts in case something bad happens. For this, *add_alert_to_queue()* is your method of choice. Supply it with a single parameter, a dictionary containing all the fields described in the alert as defined in your config.json, and *event_name* should contain the alert name. For example, simple_http defined one alert called *simple_http*, containing three fields: "originating_ip", "originating_port", and "request". A matching alert may look like this:

```
self.add_alert_to_queue({
    "event_type" : "simple_http",
    "originating_ip" : client.ip,
    "originating_port" : client.port,
    "request" : request.content
})
```

8.8 Test your service

It is recommended you override the `test()` method in your plugin class that returns triggers your alerts and returns a list to verify. The framework will automatically execute your test method and make sure all the listed alerts have been triggered successfully.

8.9 External Requirements

If your service depends on external modules, you can add them to a requirements.txt and the framework will install them in a virtual environment that will be loaded with you run the service.

It is recommended that you take simple_http as a skeleton of a service and modify it as necessary for your first honeypot. To install your new honeypot, 'honeycomb service install (directoryname)' on the chosen plugin directory, followed by 'honeycomb service run (pluginname)'. For more commands, read http://honeycomb.cymmetria.com/en/latest/cli.html#honeycomb-service.

Have fun!

Python Module Index

h

i

s

Index

Symbols

A

B

C

D

E

F

G

J

L

M

O

T

V

W