Kitty Documentation

Release 0.6.0

Cisco SAS team

January 26, 2016

Contents

Contents:

CHAPTER 1

Introduction

Sulley: Boo?
Boo: Kitty!

1.1 What is Kitty?

Kitty is an open-source modular and extensible fuzzing framework written in python, inspired by OpenRCE's Sulley and Michael Eddington's (and now Deja Vu Security's) Peach Fuzzer.

1.1.1 Goal

When we started writing Kitty, our goal was to help us fuzz unusual targets — meaning proprietary and esoteric protocols over non-TCP/IP communication channels — without writing everything from scratch each time. A generic and abstract framework that would include the common functionallity of every fuzzing process we could think of, and would allow the user to easily extend and use it to test their specific target.

1.1.2 Features

With this goal in mind, the following features were very important to us:

Modularity Each part of the fuzzer stands on its own. This means that you can use the same monitoring code for different applications, or the same payload generator (aka *Data Model*) for testing parsing of the same data that is received over different channels.

Extensibility If you need to test something "new", you will not need to change Kitty's core code. Most, if not all, features can be implemented in the user code. This includes monitoring, controlling and communicating with the fuzzed target.

Rich data modeling The data model core is rich and allows describing advanced data structures, including strings, hashes, lengths, conditions and many more. And, like most of the framework, it is designed to be extended even further as necessary.

Stateful Support for multi-stage fuzzing tests. Not only you can describe what the payload of an individual message will look like, you can also describe the order of messages, and even perform fuzzing on the sequence's order.

Client and Server fuzzing You can fuzz both servers and clients, assuming you have a matching stack. Sounds like a big requirement, but it isn't: it just means that you should have the means to communicate with the target, which you should have in most cases anyway.

Cross platform Runs on Linux, OS X and Windows. We don't judge ;-)

1.2 What it's not?

Well, Kitty is not a fuzzer. It also contains no implementation of specific protocol or communication channel. You can write your own fuzzer with it, and you can use Kitty-based code of others, but it's not an out-of-the-box fuzzer.

A good place to get (and add) implementations of Kitty models is Katnip.

1.2.1 Katnip

Kitty, as a framework, implements the fuzzer main loop, and provides syntax for modeling data and base classes for each of the elements that are used to create a full fuzzing session. However, specific implementations of classes are **not** part of the Kitty framework. This means that Kitty defines the interface and base class to perform data transactions with a target, but it doesn't provide implementations for data transmition over HTTP, TCP or UART.

Implementations of all sorts of classes can be found in the complimentary repository **Katnip** (link TBD), which has the sole porpuse of providing specific *implementaions*. As such, Katnip contains different implementations for targets (such as `TcpTarget` and `SslTarget`), controllers (such as `TelnetController` and `SshController`), monitors and templates. Use them when you write your fuzzer, as they will save you a lot of time, and may serve as a reference for your own implementations.

1.3 What's Next?

- Go over Kitty's structure
- Take a look at, and run, some examples (requires Katnip)
- Go over the tutorials
- Go fuzz something new :)

1.4 Contribution FAQ

(TBD links)

Found a bug? Open an issue

Have a fix? Great! please submit a pull request

Implemented an interesting controller/monitor/target? Please submit a pull request in the Katnip repository

Found an interesting bug using a Kitty-based fuzzer? We'd love to hear about it! please drop us a line

CHAPTER 2

Framework Structure

This document goes over the main modules that form a fuzzer and explains the relation between them. It also discusses the differences between client and server fuzzing.

2.1 Relation Between Modules

Note: Need to generate UML, look here... https://build-me-the-docs-please.readthedocs.org/en/latest/Using_Sphinx/UsingGraphicsAndDiagramsInSphinx.html

```
Fuzzer  +--- Model *--- Template *--- Field
        |
        +--- Target  +--- Controller
        |            |
        |            *--- Monitor
        |
        +--- Interface
```

2.2 Modules

2.2.1 Data Model

The first part of Kitty is the data model. It defines the structure of the messages that will be sent by the fuzzer. This includes a separation of the message into fields (such as header, length and payload), types of those fields (such as string, checksum and hex-encoded 32bit integer) and the relation between them (such as length, checksum and count). The data model also describes the order in which different messages are chained together to form a fuzzing session. This can be useful when trying to fuzz deeper parts of the system (for example, getting past an authentication stage in order to fuzz the parts of the system only reachable by authenticated users). The data model can also specify that the order in which messages are sent itself be fuzzed.

The data model is constructed using the classes defined in the **model** source folder (link TBD).

For more information, visit the data model documentation, reference and tutorials.

2.2.2 Target

Base Classes

1. `kitty.targets.client.ClientTarget` when fuzzing a client application
2. `kitty.targets.server.ServerTarget` when fuzzing a server application

API Reference kitty.targets package

The target module is in charge of everything that is related to the victim. Its responsibilities are:

1. When fuzzing a server — initiating the fuzzing session by sending a request to the server, and handling the response, when such response exists.
2. When fuzzing a client — triggering a fuzzing session by causing the client to initiate a request to the server. The server, with the help of the stack (see external dependencies TBD), will send a fuzzed response to the client. Note that in this case the target itself is not involved the client-server-fuzzer communication!
3. Managing the monitors and controllers (see below).

The sources for the target classes are located in the **targets** source folder. There are two target base classes, `ClientTarget` (in `targets/client.py`) and `ServerTarget` (in `targets/server.py`). These classes define the target APIs and should be subclassed when implementing a new target class.

A class should be written for every new type of target. For example, if you want to test a server application that communicates over a serial UART connection, you will need to create a new class that inherits from `ServerTarget` and is able to send data over the UART. However, many times it will only require the implementation of the send/receive functions and not much more. Some targets are already available in the **Katnip** (link TBD) repository, so you don't need to implement them yourself: for example, `TcpTarget` may be used to test HTTP servers and `SslTarget` may be used to test HTTPS servers.

The controller and the monitors (described below) are managed by the target, which queries and uses them as needed.

For each test the target generates a report which contains the reports of the controller and all monitors. This report is passed back to the fuzzer (described below) upon request.

2.2.3 Controller

Base Class `kitty.controllers.base.BaseController`

API Reference kitty.controllers package

The controller is in charge of preparing the victim for the test. It should make sure that the victim is in an appropriate state before the target initiates the transfer session. Sometimes it means doing nothing, other times it means starting or reseting a VM, killing a process or performing a hard reset to the victim hardware.

Since the controller is reponsible for the state of the victim, it is expected to perform basic monitoring as well, and report whether the victim is ready for the next test.

2.2.4 Monitor

Base Class `kitty.monitors.base.BaseMonitor`

API Reference kitty.monitors package

A monitor object monitors the behavior of the victim. It may monitor the network traffic, memory consumption, serial output or anything else.

Since there might be more than a single behavior to monitor, multiple monitors can be used when fuzzing a victim.

2.2.5 Fuzzer

Classes

1. `kitty.fuzzers.client.ClientFuzzer` when fuzzing a client target.
2. `kitty.fuzzers.server.ServerFuzzer` when fuzzing a server target.

API Reference kitty.fuzzers package

A fuzzer drives the whole fuzzing process. Its job is to obtain mutated payloads from the model, initiate the data transaction, receive the report from the target, and perform further processing, if needed. A fuzzer is the top level entity in our test runner, and should not be subclassed in most cases.

2.2.6 Interface

Base Class `kitty.interfaces.base.BaseInterface`

API Reference kitty.interfaces package

Interface is a user interface, which allows the user to monitor and check the fuzzer as it goes. The web interface should suffice in most cases.

CHAPTER 3

Tutorials

Here you can find some tutorials to help you get started with Kitty.

3.1 Complete Examples

3.1.1 Server vs. Client Fuzzing

Kitty is a framework for fuzzing various kinds of entities. It can fuzz **servers** — by acting like a web client to initiate requests to the victim server, or by acting like a USB host to send commands to the victim USB device; and it can also fuzz **clients**, by responding to requests made by them — acting as the web server to respond to requests from the victim browser, or as the USB device responding to commands from the victim USB host.

In Kitty's terminology, the **client** is the side initiating the data exchange, and the **server** is the side reacting to it. By this defintion, a TCP server would be considered a server, as it reacts to requests initiated by the fuzzer; but so would a PDF viewer, if it is started with a given file provided by the fuzzer. Similarly, an IRC client or a web browser would be considered clients, as they initiate requests to the fuzzer and then process its responses; but so would a USB-Host stack, which initiates the data exchange with a fuzzer disguised as the USB device. And in fact, the same entity could be a client in some contexts, and a server in others: a web browser would be considered a client in the context of conversations made to the fuzzer-server, as seen above; but would be considerd a server in the context of being run as a process by the fuzzer, with a command-line argument pointing it at a given (presumably-fuzzed) URL.

There is a big difference between server fuzzing and client fuzzing, and the need to support both was a main driver for the creation of Kitty. This section will list the conceptual, flow and implementation differences between those two modes.

Conceptual Differences

Session initiation Since the nature of a server is to handle and respond to requests, it is rather simple to fuzz it on our terms: we need to make sure it is up and ready to accept requests, and then start sending those requests to it. This means that the fuzzer is **active**, it initiates the data exchange and decides what data the server will handle, assuming that the server starts in the same state for each data exchange session.

On the other hand, when fuzzing a client, the mutated data is the response, not the request. The client is the one to initiate the data exchange, the fuzzer acts as a server, and so it is **passive** and cannot control the timing or even the occurence of the fuzzing session. In order to take control back, the fuzzer needs some way to cause the client to start the data exchange.

Communication Stack When a server is tested, it is the fuzzer, as we just saw, that initiates the communication. This means that the fuzzer can choose at exactly what layer to begin the communication.

So, for example, when testing a TCP-based protocol, the TCP stack may be used, and when testing an HTTP-based protocol, an HTTP stack may be used. Moreover, there usually are readily-available Python or C APIs for initiating the communication starting at any of these layers.

When testing a client, on the other hand, the fuzzer is only responding to requests from the client. As such, the fuzzer cannot easily choose at which layer to handle the communication — it must handle the request at whichever layer it was received. Thus, fuzzing a specific layer will very likely require hooking into the stack implementation in order to inject the mutated responses.

Flow Differences

The flow differences are derived from the conceptual differences, the flow for each scenario is described below.

[TBD - flow charts]

Implementation Differences

The implementation differences are derived from the flow differences and are listed in the table below.

Component	Server Mode	Client Mode
Stack	Unmodified in most cases	Hooked in most cases, runs as a separate process
Target	Responsible for sending and receiving data from victim, uses the stack	Responsible for triggering the data exchange in the victim (using the controller), data is handled directly by the stack
Controller	Brings the victim to initial state, monitors victim status	Same as in server mode, but additionally responsible for triggering the data exchange
Fuzzer	Actively polls the data model for more data and triggers the data exhange	Waits in a separate process for requests from the modified stack over IPC, provides stack with mutated data when needed

3.1.2 Server Fuzzing Tutorial

This tutorial will guide you through the steps that are taken to build a fuzzer for your target, we will build such a fuzzer for a tiny HTTP server. It will be a minimal implementation, just to show the basics.

- First, we need to define our *data model* to let Kitty know how our protocol looks like.
- Then, we need to define how will we communicate with our *target*.
- After that, we need to find a way to *control* our target and how to *monitor* it (TBD).
- And finally, we need to connect all those pieces together.

Data Model

We start with a simple example, of fuzzing a simple HTTP GET request. For simplicity, we will not look at the spec to see the format of each message.

A simple "GET" request may look like this:

```
GET /index.html HTTP/1.1
```

There are some obvious fields in this request:

1. Method - a string with the value "GET"

2. Path - a string with the value "/index.html"
3. Protocol - a string with the value "HTTP/1.1"

However, there are some other things in this message, which we ignored:

- (1.a) The space between Method and Path
- (2.a) The space between Path and Protocol
- 3. The double "new lines" ("") at the end of the request

Those are the delimiters, and we should not forget them.

Here's the translation of this structure to a Kitty model:

Data model, version 1

```
from kitty.model import *

http_get_v1 = Template(name='HTTP_GET_V1', fields=[
    String('GET', name='method'),           # 1. Method - a string with the value "GET"
    Delimiter(' ', name='space1'),          # 1.a The space between Method and Path
    String('/index.html', name='path'),     # 2. Path - a string with the value "/index.html"
    Delimiter(' ', name='space2'),          # 2.a. The space between Path and Protocol
    String('HTTP/1.1', name='protocol'),    # 3. Protocol - a string with the value "HTTP/1.1"
    Delimiter('\r\n\r\n', name='eom'),      # 4. The double "new lines" ("\r\n\r\n") at the end of t
])
```

We used three new objects here, all declared in **kitty/model/__init__.py**:

1. `Template`, which is the top most container of the low level data model, it encloses a full message. It received all of its enclosed fields as an array.
2. `String('GET', name='method')` creates a new `String` object with the default value 'GET', and names it 'method'.
3. `Delimiter(' ', name='space1')` creates a new `Delimiter` object with the default value ' ', and names it 'space1'.

Based on this model, Kitty will generate various mutations of the template, each mutation is constructed from a mutation of one of the fields, and the default values of the rest of them. When a field has no more mutations, it will return it to its default value, and move to the next field.

Even in this simple example, we refine our model even more. We can see that the **Protocol** field can be divided even more. We can split it to the following fields:

- (3.a) Protocol Name - a string with the value "HTTP"
- (3.b) The '/' after "HTTP"
- (3.c) Major Version - a number with the value 1
- (3.d) The '.' between 1 and 1
- (3.e) Minor Version - a number with the value 1

Now we can replace the `protocol` string field with 5 fields.

Data model, version 2

```
from kitty.model import *

http_get_v2 = Template(name='HTTP_GET_V2', fields=[
    String('GET', name='method'),               # 1. Method - a string with the value "GET"
    Delimiter(' ', name='space1'),              # 1.a The space between Method and Path
```

```
    String('/index.html', name='path'),          # 2. Path - a string with the value "/index.html"
    Delimiter(' ', name='space2'),               # 2.a. The space between Path and Protocol
    String('HTTP', name='protocol name'),        # 3.a Protocol Name - a string with the value "HTTP"
    Delimiter('/', name='fws1'),                 # 3.b The '/' after "HTTP"
    Dword(1, name='major version',               # 3.c Major Version - a number with the value 1
          encoder=ENC_INT_DEC)                   # encode the major version as decimal number
    Delimiter('.', name='dot1'),                 # 3.d The '.' between 1 and 1
    Dword(1, name='major version',               # 3.e Minor Version - a number with the value 1
          encoder=ENC_INT_DEC)                   # encode the minor version as decimal number
    Delimiter('\r\n\r\n', name='eom')            # 4. The double "new lines" ("\r\n\r\n") at the end o
])
```

We just met two new objects:

1. `Dword(1, name='major version')` create a 32-bit integer field with default value 1 and name it 'major version'
2. `ENC_INT_DEC` is an *encoder* that encodes this int as a decimal number. An encoder only affects the representation of the number, not its data nor its mutations

`Dword` is part of a family of fields (`Byte`, `Word` and `Qword`) that provides convenient initialization to the basic field on `BitField`.

> The last example shows how we can treat a payload in different ways, and how it affects our data model. It is not always good to give too much details in the model. Sometimes too much details will make the fuzzer miss some weird cases, because it will always be "almost correct" and most of the times it will cause the fuzzing session to be very long. There is a balance that should be reached, and each implementor should find his own (this is a spiritual guide as well).

HTTP_GET_V2 is pretty detailed data model, but while all parts of the template that we want Kitty to send should be represented by fields, there are fields that we don't want Kitty to mutate. For Instance, the two new lines at the end of the request signals the server that the message has ended, and if they are not sent, the request will probably not be processed at all. Or, if we know there is a "GET" handler function in the target, we might want to always have "GET " at the start of our template.

The next example achieves both goals, but in two different ways:

Data model, version 3

```
from kitty.model import *

http_get_v2 = Template(name='HTTP_GET_V2', fields=[
    String('GET', name='method', fuzzable=False),   # 1. Method - a string with the value "GET"
    Delimiter(' ', name='space1', fuzzable=False),  # 1.a The space between Method and Path
    String('/index.html', name='path'),             # 2. Path - a string with the value "/index.html"
    Delimiter(' ', name='space2'),                  # 2.a. The space between Path and Protocol
    String('HTTP', name='protocol name'),           # 3.a Protocol Name - a string with the value "HT
    Delimiter('/', name='fws1'),                    # 3.b The '/' after "HTTP"
    Dword(1, name='major version',                  # 3.c Major Version - a number with the value 1
          encoder=ENC_INT_DEC)                      # encode the major version as decimal number
    Delimiter('.', name='dot1'),                    # 3.d The '.' between 1 and 1
    Dword(1, name='major version',                  # 3.e Minor Version - a number with the value 1
          encoder=ENC_INT_DEC)                      # encode the minor version as decimal number
    Static('\r\n\r\n', name='eom')                  # 4. The double "new lines" ("\r\n\r\n") at the e
])
```

The first method we used is setting the `fuzzable` parameter of a field to `False`, as we did for the first two fields, this method lets us preserve the structure of the model, and change it easily when we do want to mutate those fields:

```
String('GET', name='method', fuzzable=False),    # 1. Method - a string with the value "GET"
Delimiter(' ', name='space1', fuzzable=False),   # 1.a The space between Method and Path
```

The second method is by using a `Static` object, which is immutable, as we did with the last field, this method improves the readability if we have a long chunk of data in our template that will never change:

```
# 4. The double "new lines" ("\r\n\r\n") at the end of the request
Static('\r\n\r\n', name='eom')
```

Target

Now that we have a data model, we need to somehow pass it to our target. Since we are fuzzing an HTTP server implementation, we need to send our requests over TCP. There is already a target class to take care of TCP communication with the server - `kitty.targets.tcp.TcpTarget`, but we will build it here again, step by step, to learn from it.

When fuzzing a server, our target should inherit from `ServerTarget`. Except of two methods - `_send_to_target` and `_receive_from_target`, each method that you override should call its super.

Each fuzzing session goes through the following stages:

```
1. set up the environment
2. for each mutation:
    1. preform pre-test actions
    2. do transmition
    3. cleanup after the test
    4. provide a test report
3. tear down the environment
```

Each of those steps is reflected in the `ServerTarget` API:

Step	Corresponding API
set up the environment	`setup()`
perform pre-test actions	`pre_test(test_num)`
do transmission	`transmit(payload)` (calls `_send_to_target(payload)` and `_receive_from_target()`)
cleanup after test	`post_test(test_num)`
provide a test report	`get_report()`
tear down the environment	`teardown()`

Now let's implement those methods (the part we need):

class definition and constructor

```
'''
TcpTarget is an implementation of a TCP target
'''
import socket
from kitty.targets.server import ServerTarget


class TcpTarget(ServerTarget):
    '''
```

```
    TcpTarget is implementation of a TCP target for the ServerFuzzer
    '''

    def __init__(self, name, host, port, timeout=None, logger=None):
        '''
        :param name: name of the object
        :param host: hostname of the target (the TCP server)
        :param port: port of the target
        :param timeout: socket timeout (default: None)
        :param logger: logger for this object (default: None)
        '''
        ## Call ServerTarget constructor
        super(TcpTarget, self).__init__(name, logger)
        ## hostname of the target (the TCP server)
        self.host = host
        ## port of the target
        self.port = port
        if (host is None) or (port is None):
            raise ValueError('host and port may not be None')
        ## socket timeout (default: None)
        self.timeout = timeout
        ## the TCP socket
        self.socket = None
```

We create a socket at the beginning of each test, and close it at the end

pre_test and post_test

```
def pre_test(self, test_num):
'''
prepare to the test, create a socket
'''
    ## call the super (report preparation etc.)
    super(TcpTarget, self).pre_test(test_num)
    ## only create a socket if we don't have one
    if self.socket is None:
        sock = self._get_socket()
        ## set the timeout
        if self.timeout is not None:
            sock.settimeout(self.timeout)
        ## connect to socket
        sock.connect((self.host, self.port))
        ## our TCP socket
        self.socket = sock

def _get_socket(self):
    '''get a socket object'''
    ## Create a TCP socket
    return socket.socket(socket.AF_INET, socket.SOCK_STREAM)

def post_test(self, test_num):
    '''
    Called after a test is completed, perform cleanup etc.
    '''
    ## Call super, as it prepares the report
    super(TcpTarget, self).post_test(test_num)
    ## close socket
```

```
    if self.socket is not None:
        self.socket.close()
        ## set socket to none
        self.socket = None
```

Notice that we called the super in each overriden method. This is important, as the super class perform many tasks that are not target-specific.

The next step is to implement the sending and receiving. It's pretty straight forward, we call socket's `send` and `receive` methods. We don't call super in those methods, as the super is not implemented.

send_to_target + receive_from_target

```
def _send_to_target(self, data):
    self.socket.send(data)

def _receive_from_target(self):
    return self.socket.recv(10000)
```

That's it. We have a target that is able to perform TCP transmissions.

As the final stage of each test is providing a report, you can add fields to the report in your target at each of the methods above.

A basic fuzzer can already created with what we've seen so far. Dummy controller can supply supply the requirement of the base target class, and we don't have to use any monitor at all, but if we want to be able to not only crash the client, but to be able to detect the crash and restart it once it crashes, we need to implement a `Controller`

Controller

As described in the overview, and in the Controller Documentation, the controller makes sure that our victim is ready to be fuzzed, and if it can't it reports failure.

In our example, we have an HTTP server that we want to fuzz, for simplicity, we will run the server locally. We will do it by implementing `LocalProcessController` a class that inherits from `kitty.controllers.base.BaseController`.

The controller is controller by the `Target` and it follows pretty much the same stages as the target (excluding the transmission) Each fuzzing session goes through the following stages:

```
1. set up the environment
2. for each mutation:
    1. preform pre-test actions
    2. cleanup after the test
    3. provide a test report
3. tear down the environment
```

Each of those steps is reflected in the `ServerTarget` API:

Step	Corresponding API	Controllers role
set up the environment	`setup()`	preparations
perform pre-test actions	`pre_test(test_num)`	prepare the victim to the test (make sure its up)
cleanup after test	`post_test(test_num)`	check the status of the victim, shut it down if needed
provide a test report	`get_report()`	provide a report
tear down the environment	`teardown()`	perform a cleanup

class definition and constructor

```
from kitty.controllers.base import BaseController

class LocalProcessController(BaseController):
    '''
    LocalProcessController a process that was opened using subprocess.Popen.
    The process will be created for each test and killed at the end of the test
    '''

    def __init__(self, name, process_path, process_args, logger=None):
        '''
        :param name: name of the object
        :param process_path: path to the target executable
        :param process_args: arguments to pass to the process
        :param logger: logger for this object (default: None)
        '''
        super(ClientProcessController, self).__init__(name, logger)
        assert(process_path)
        assert(os.path.exists(process_path))
        self._process_path = process_path
        self._process_name = os.path.basename(process_path)
        self._process_args = process_args
        self._process = None
```

Our controller has nothing to do at the setup stage, so we don't override this method

Before a test starts, we need to make sure that the victim is up

pre_test

```
def pre_test(self, test_num):
    '''start the victim'''
    ## stop the process if it still runs for some reason
    if self._process:
        self._stop_process()
    cmd = [self._process_path] + self._process_args
    ## start the process
    self._process = Popen(cmd, stdout=PIPE, stderr=PIPE)
    ## add process information to the report
    self.report.add('process_name', self._process_name)
    self.report.add('process_path', self._process_path)
    self.report.add('process_args', self._process_args)
    self.report.add('process_id', self._process.pid)
```

When the test is over, we want to store the output of the process, as well as its exit code (if crashed):

post_test

```
def post_test(self):
    '''Called when test is done'''
    self._stop_process()
    ## Make sure process started by us
    assert(self._process)
    ## add process information to the report
```

```
    self.report.add('stdout', self._process.stdout.read())
    self.report.add('stderr', self._process.stderr.read())
    self.logger.debug('return code: %d', self._process.returncode)
    self.report.add('return_code', self._process.returncode)
    ## if the process crashed, we will have a different return code
    self.report.add('failed', self._process.returncode != 0)
    self._process = None
    ## call the super
    super(ClientProcessController, self).post_test()
```

When all fuzzing is over, we perform the `teardown`:

teardown

```
    def teardown(self):
        '''
Called at the end of the fuzzing session, override with victim teardown
'''
        self._stop_process()
        self._process = None
        super(ClientProcessController, self).teardown()
```

Finally, here is the implementation of the `_stop_process` method

_stop_process

```
def _stop_process(self):
    if self._is_victim_alive():
        self._process.terminate()
        time.sleep(0.5)
        if self._is_victim_alive():
            self._process.kill()
            time.sleep(0.5)
            if self._is_victim_alive():
                raise Exception('Failed to kill client process')

def _is_victim_alive(self):
    return self._process and (self._process.poll() is None)
```

3.1.3 Client Fuzzing Tutorial

One of the advanteges of kitty is its ability to fuzz client targets. Client targets are targets that we cannot fuzz by sending malformed requests, but by sending malformed responses.

As explained in the Server vs. Client Fuzzing, this is a big difference, for two main reasons. The first reason is that unlike server fuzzing, the communication is started by the target, and not by the fuzzer. The second reason is that in order to fuzz a client we usually need to hook some functions in the server stack.

First we will explain how to fuzz a client target with Kitty. After that, we will explain how to separate the code so the fuzzer will run in a *separate process than the stack*.

How Does It Work

Since in our case the communication is triggered by the target and handled by the stack, the fuzzer is passive. It triggers the client to start the communication from its own context (thread / process) and then does nothing until it is called by the stack to provide a response to the target request.

The stack needs to get a mutation of a response from the fuzzer. The fuzzer exposes the `get_mutation(name, data)` method to the stack to provide those mutations. The responsibility of the stack is to call `get_mutation` in request handlers. If `get_mutation` returns `None`, the stack handles the request apropriety, otherwise, it returns the result of `get_mutation` to the target.

Here's an example of (psuedo) client hook:

```
class StackImplementation:
    # ...
    def build_get_response(self, request_id):
        resp = self.fuzzer.get_mutation(stage='get_response', data={ 'request_id' : request_id })
        if resp:
            return resp
        # build valid response
        resp = ...
        return resp
```

`self.fuzzer` in the example above is the instance of `ClientFuzzer` that we passed to the stack. We call `get_mutation` with two arguments. The first, `get_response` is the name of the name of the scheme (request) that is used for this request, that we create in our data model. In `get_mutation` the fuzzer checks if it currently fuzzing this scheme, and if so, it will return a mutated response, otherwise it will return None. The second argument is a dictionary of data that should be inserted into the `DynamicFields` in the scheme, it is usually a data that is transaction dependant and is not known when building the scheme, for example, transaction or request id, such as in the example above.

Building the Fuzzer

We will list the different parts of the client fuzzer, in the last section we will give a *simple example* of such a fuzzer.

Target

The target for client fuzzing inherits from `kitty.target.client.ClientTarget` unlike in server fuzzing, it's major work is managing the controller and monitors, so you can often just instantiate `kitty.target.client.ClientTarget` directly.

Controller

The controller of the client target inherits from `kitty.controllers.client.ClientController`. The most important method in it is `trigger`. This method triggers the client to start the communication with the server stack. Since this method differ from target to target, it is not implemented in `ClientController` and must be implemented in a new class.

The other methods are inherited from `kitty.controllers.base.BaseController` and may or may not be implemented in the new class, based on your needs.

Monitor

The monitors of the client target inherit from `kitty.monitors.base.BaseMonitor` there is nothing special in client monitors.

User Interface

The user interface of the client fuzzer inherit from `kitty.interfaces.base.BaseInterface` there is nothing special about client fuzzer user interface.

Fuzzer

When fuzzing a client, you should create a `kitty.fuzzers.client.ClientFuzzer` object, pass it to the stack, and then start it.

Fuzzer Building Example

fuzz_special_stack.py

```
import struct
from kitty.targets import ClientTarget
from kitty.controllers import ClientController
from kitty.interfaces import WebInterface
from kitty.fuzzers import ClientFuzzer
from kitty.model import GraphModel
from kitty.model import Template, Dynamic, String


################# Modified Stack #################
class MySpecialStack(object):
    # We only show the relevant methods
    def __init__(self):
        self.fuzzer = None
        self.names = {1: 'Lumpy', 2: 'Cuddles', 3: 'Flaky', 4: 'Petunya'}

    def set_fuzzer(self, fuzzer):
        self.fuzzer = fuzzer

    def handle_GetName(self, name_id):
        resp = self.fuzzer.get_mutation(stage='GetName response', data={'name_id': struct.pack('I', n
        if resp:
            return resp
        name = '' if name_id not in self.names else self.names[name_id]
        return struct.pack('I', name_id) + name

################# Data Model #################

get_name_response_template = Template(
    name='GetName response',
    fields=[
        Dynamic(key='name_id', default_value='\x00', name='name id'),
        String(value='admin', nane='name')
    ]
)
```

```
################ Controller Implementation #################
class MyClientController(ClientController):
    def __init__(self):
        super(MyClientController, self).__init__('MyClientController')

    def trigger(self):
        # trigger transaction start at the client
        pass


################ Actual fuzzer code #################
target = ClientTarget('Example Target')

controller = MyClientController()
target.set_controller(controller)

model = GraphModel()
model.connect(get_name_response_template)
fuzzer = ClientFuzzer()
fuzzer.set_model(model)
fuzzer.set_target(target)
fuzzer.set_interface(WebInterface())

my_stack = MySpecialStack()
my_stack.set_fuzzer(fuzzer)
fuzzer.start()
my_stack.start()
```

Remote Fuzzer

The are two big problems with the client fuzzer that we've shown in the previous section. The first problem is that it ties us to python2 implementations of the stack. This means that even if you have a stack that you can modify, if it's not written in python2 you will need to perform major changes to your code, or not use it at all. The second problem is that even when using python2, different threading models and signal handling may cause big issues with kitty, as it uses python threads and uses signal handlers.

To overcome those issue, we have created the `kitty.remote` package. It allows you to separate the stack process from the fuzzer process.

> Currently, we only support python2 and python3, using the same python modules (with `six`) support for other languages will be provided in the future.

The idea is pretty simple - on the stack side, we only add `RpcClient`. No data models, monitors, target or anything like that. On the fuzzer side, we create the fuzzer as before, with all its classes, and than wrap it with a `RpcServer`, which waits for requests from the agent.

The next example shows how we convert the *previous example* to use the remote package.

Python2/3 Remote Fuzzer

my_stack.py (python3)

```
from kitty.remote import RpcClient

################ Modified Stack #################
```

```
class MySpecialStack(object):
    # We only show the relevant methods
    def __init__(self):
        self.fuzzer = None
        self.names = {1: 'Lumpy', 2: 'Cuddles', 3: 'Flaky', 4: 'Petunya'}

    def set_fuzzer(self, fuzzer):
        self.fuzzer = fuzzer

    def handle_GetName(self, name_id):
        resp = self.fuzzer.get_mutation(stage='GetName response', data={'name_id': struct.pack('I', n
        if resp:
            return resp
        name = '' if name_id not in self.names else self.names[name_id]
        return struct.pack('I', name_id) + name

fuzzer = RpcClient(host='127.0.0.1', port=26010)

my_stack = MySpecialStack()
my_stack.set_fuzzer(fuzzer)

fuzzer.start()
my_stack.start()
```

my_stack_fuzzer.py (python2)

```
from kitty.targets import ClientTarget
from kitty.controllers import ClientController
from kitty.interfaces import WebInterface
from kitty.fuzzers import ClientFuzzer
from kitty.model import GraphModel
from kitty.model import Template, Dynamic, String
from kitty.remote import RpcServer


################# Data Model #################
get_name_response_template = Template(
    name='GetName response',
    fields=[
        Dynamic(key='name_id', default_value='\x00', name='name id'),
        String(value='admin', nane='name')
    ]
)


################# Controller Implementation #################
class MyClientController(ClientController):
    def __init__(self):
        super(MyClientController, self).__init__('MyClientController')

    def trigger(self):
        # trigger transaction start at the client
        pass

################# Actual fuzzer code #################
target = ClientTarget('Example Target')

controller = MyClientController()
```

```
target.set_controller(controller)

model = GraphModel()
model.connect(get_name_response_template)
fuzzer = ClientFuzzer()
fuzzer.set_model(model)
fuzzer.set_target(target)
fuzzer.set_interface(WebInterface())

remote = RpcServer(host='127.0.0.1', port=26010, impl=fuzzer)
remote.start()
```

3.1.4 Extending the Framework

Kitty comes with implementation for several targets and controllers, ready to be used. This page will explain how to create new classes to meet your needs.

The API of all classes is mainly state-oriented. This means that the functions are called upon state, and does not specify an action by their names. For example, the `BaseController` methods are `setup`, `teardown`, `pre_test` and `post_test`, not `prepare_victim`, `stop_victim`, `restart_victim` etc.

Controllers

All controllers inherit from `BaseController` (kitty/controllers/base.py), client controllers need to support triggering, so they have an extended API.

Server Controller

Base Class: `kitty.controllers.base.BaseController`

Methods `setup(self)`: Called at the beginning of the fuzzing session, this is the time for initial settings.

`teardown(self)`: Called at the end of the fuzzing session, this is the time for cleanup and shutdown of the victim.

`pre_test(self, test_number)`: Called before each test. should call super if overriden.

`post_test(self)`: Called after each test. should call super if overriden.

Members `report`: A `Report` object, add elements for it as needed

Client Controller

Base Class: **kitty.controllers.base.ClientController**

Methods `trigger(self)`: client transaction triggering

Monitor

Base Class: `kitty.monitros.base.BaseMonitor`

A monitor is running on a separate thread. A generic thread function with a loop is running by default, and calls `_monitor_func`, which should be overriden by any monitor implementation. Not that this function is called in a loop, so there is no need to perform a loop inside it.

Methods

`setup(self)`: >Called at the beginning of the fuzzing session, this is the time for initial settings. should call super if overriden.

`teardown(self)`: >Called at the end of the fuzzing session, this is the time for cleanup and shutdown of the monitor. should call super if overriden.

`pre_test(self, test_number)`: >Called before each test. should call super if overriden.

`post_test(self)`: >Called after each test. should call super if overriden.

`_monitor_func(self)`: >Called in a loop once the monitor is setup. Unless there are specific needs, there is no need to implement stop mechanism or endless loop inside this function, as they are implemented by its wrapper. See the implementation of SerialMonitor (kitty/monitors/serial.py) for example.

Members

`report`: >A `Report` object, add elements for it as needed

Target

Each target should inherit from `ServerTarget` or `ClientTarget`, both inherit from `BaseTarget`. All methods in `BaseTarget` are relevant to `ServerTarget` and `ClientTarget` as well.

Base Target

Base Class: **kitty.targets.base.BaseTarget**

CHAPTER 4

Data Model Overview

As described in the overview, Kitty is by default a generation-based fuzzer. This means that a model should be created to represent the data exchanged in the fuzzed protocol.

On the high level, we describe and fuzz different sequences of messages, this allows building fuzzing scenarios that are valid until a specific stage but also to fuzz the order of messages as well.

4.1 Low Level Model

At the low level, there is a signle payload, constructed from multiple fields. Those fields can be encoded, dependant on other fields, be renderened conditionaly and combined with other fields into larger logical units. The top unit of the logical units is Template, and Template is the only low-level interface to the high-level of the data model.

A full documentation of the low level data model can be found in the API reference.

4.2 High Level Model

The high level model describe how a sequence of messages (Templates) looks like to the fuzzer. There are two main models for this part, `GraphModel` and `StagedSequenceModel`. An additional model - `RandomSequenceModel` is a naive case of `StagedSequenceModel` and is more convenient, when the order of messages really doesn't matter.

During the fuzzing session, the fuzzer queries the data model for a sequence to transmit. The data model chooses the next sequence according to its strategy and perform internal mutations if needed.

CHAPTER 5

Kitty Tools

When installing Kitty using setup.py or pip, it will install several tools:

5.1 Template Tester

```
Usage:
    kitty-template-tester [--fast] [--tree] [--verbose] <FILE> ...

This tool mutates and renders templates in a file, making sure there are no
syntax issues in the templates.
It doesn't prove that the data model is correct, only checks that the it is
a valid model

Options:
    <FILE>      python file that contains templates in dictionaries, lists or globals
    --fast      only import, don't run all mutations
    --tree      print fields tree of the template instead of mutating it
    --verbose   print full call stack upon exception
```

5.2 CLI Web Client

```
Usage:
    kitty-web-client (info [-v]|pause|resume) [--host <hostname>] [--port <port>]
    kitty-web-client reports store <folder> [--host <hostname>] [--port <port>]
    kitty-web-client reports show <file> ...

Retrieve and parse kitty status and reports from a kitty web server

Options:
    -v --verbose            verbose information
    -h --host <hostname>    kitty web server host [default: localhost]
    -p --port <port>        kitty web server port [default: 26000]
```

CHAPTER 6

API Reference

6.1 Kitty API Reference

6.1.1 Subpackages

kitty.controllers package

The kitty.controllers package provide the basic controller classes. The controller role is to provide means to start/stop the victim and put it in the appropriate state for a test.

BaseController should not be instantiated. It should be extended when performing server fuzzing.

ClientController is used for client fuzzing. It should be extended with an implementation for the *trigger()* function to trigger the victim to start the communications.

EmptyController is used when no actual work should be done by the controller.

Submodules

kitty.controllers.base module The controller is in charge of preparing the victim for the test. It should make sure that the victim is in an appropriate state before the target initiates the transfer session. Sometimes it means doing nothing, other times it means starting or reseting a VM, killing a process or performing a hard reset to the victim hardware. Since the controller is reponsible for the state of the victim, it is expected to perform a basic monitoring as well, and report whether the victim is ready for the next test.

class kitty.controllers.base.**BaseController**(*name*, *logger=None*)

Bases: *kitty.core.kitty_object.KittyObject*

Base class for controllers. Defines basic variables and implements basic behavior.

__init__(*name*, *logger=None*)

Parameters

- **name** – name of the object
- **logger** – logger for the object (default: None)

get_report()

Return type *Report*

Returns a report about the victim since last call to pre_test

post_test()
Called when test is done. Call super if overriden.

pre_test(*test_number*)
Called before a test is started. Call super if overriden.

Parameters **test_number** – current test number

setup()
Called at the beginning of the fuzzing session. You should override it with the actual implementation of victim setup.

teardown()
Called at the end of the fuzzing session. You should override it with the actual implementation of victim teardown.

kitty.controllers.client module `ClientController` is a controller for victim in client mode, it inherits from *`BaseController`*, and implements one additional method: *`trigger()`*.

class kitty.controllers.client.**ClientController**(*name*, *logger=None*)
Bases: *kitty.controllers.base.BaseController*

Base class for client controllers.

__init__(*name*, *logger=None*)

Parameters

- **name** – name of the object
- **logger** – logger for the object (default: None)

trigger()
Trigger a data exchange from the tested client

kitty.controllers.empty module EmptyController does nothing, implements both client and server controller API

class kitty.controllers.empty.**EmptyController**(*name*, *logger=None*)
Bases: *kitty.controllers.client.ClientController*

EmptyController does nothing, implements both client and server controller API

__init__(*name*, *logger=None*)

Parameters

- **name** – name of the object
- **logger** – logger for the object (default: None)

post_test()
Called when test is done

Note: Call super if overriden

pre_test(*test_number*)
Called before a test is started

Note: Call super if overriden

trigger()
Trigger a data exchange from the tested client

kitty.core package

The classes in this module has very little to do with the fuzzing process, however, those classes and functions are used all over kitty.

exception kitty.core.**KittyException**
Bases: exceptions.Exception

Simple exception, used mainly to make tests better, and identify exceptions that were thrown by kitty directly.

kitty.core.**khash**(**args*)
hash arguments. khash handles None in the same way accross runs (which is good :))

Submodules

kitty.core.kassert module
kitty.core.kassert.**is_in**(*obj*, *it*)

Parameters

- **obj** – object to assert
- **it** – iterable of elements we assert obj is in

Raise an exception if obj is in an iterable

kitty.core.kassert.**is_int**(*obj*)

Parameters **obj** – object to assert

Raise an exception if obj is not an int type

kitty.core.kassert.**is_of_types**(*obj*, *types*)

Parameters

- **obj** – object to assert
- **types** – iterable of types, or a signle type

Raise an exception if obj is not an instance of types

kitty.core.kassert.**not_none**(*obj*)

Parameters **obj** – object to assert

Raise an exception if obj is not None

kitty.core.kitty_object module KittyObject is subclassed by most of Kitty's objects.

It provides logging, naming, and description of the object.

class kitty.core.kitty_object.**KittyObject**(*name*, *logger=None*)
Bases: object

Basic class to ease logging and description of objects.

__init__(*name*, *logger=None*)

Parameters **name** – name of the object

get_description()

Return type str

Returns the description of the object. by default only prints the object type.

classmethod get_logger()

Returns the class logger

get_name()

Return type str

Returns object's name

not_implemented(*func_name*)
log access to unimplemented method and raise error

Parameters **func_name** – name of unimplemented function.

Raise NotImplementedError detailing the function the is not implemented.

kitty.core.threading_utils module Threading Utils

class kitty.core.threading_utils.**FuncThread**(*func*, **args*)
Bases: threading.Thread

FuncThread is a thread wrapper to create thread from a function

__init__(*func*, **args*)

Parameters

- **func** – function to be called in this thread
- **args** – arguments for the function

run()
run the the function in this thread's context

class kitty.core.threading_utils.**LoopFuncThread**(*func*, **args*)
Bases: threading.Thread

FuncThread is a thread wrapper to create thread from a function

__init__(*func*, **args*)

Parameters

- **func** – function to be call in a loop in this thread
- **args** – arguments for the function

run()
run the the function in a loop until stoped

set_func_stop_event(*func_stop_event*)

Parameters **func_stop_event** (Event) – event to signal stop to the _func

stop()
stop the thread, return after thread stopped

kitty.data package

This package contains class for managing data related to the fuzzing session.

Submodules

kitty.data.data_manager module This module is usde to store the fuzzing session related data. It provides both means of communications between the fuzzer and the user interface, and persistent storage of the fuzzing session results.

class kitty.data.data_manager.**DataManager**(*dbname*)
Bases: threading.Thread

Manages data on a dedicated thread. All calls to it should be done by submitting DataManagerTask

Example

```
dataman = DataManager('fuzz_session.sqlite`)
dataman.start()
def get_session_info(manager):
    return manager.get_session_info_manager().get_session_info()
get_info_task = DataManagerTask(get_session_info)
dataman.submit_task(get_info_task)
session_info = get_info_task.get_results()
```

__init__(*dbname*)

Parameters **dbname** – database name for storing the data

close()
close the database connection

get_reports_manager()

Return type *ReportsTable*

Returns reports manager

get_session_info_manager()

Return type *SessionInfoTable*

Returns session info manager

get_test_info()

Returns test info

open()
open the database

run()
thread function

set_test_info(*test_info*)
set the test info

Parameters **test_info** (*dict*) – the test information to be set

submit_task(*task*)
submit a task to the data manager, to be proccessed in the DataManager context

Parameters **task** (*DataManagerTask*) – task to perform

class kitty.data.data_manager.**DataManagerTask**(*task*)
Bases: object

Task to be performed in the *DataManager* context

__init__(*task*)

Parameters **task** (function(*DataManager*) -> object) – task to be performed

execute(*dataman*)
run the task

Parameters **dataman** (*DataManager*) – the executing data manager

get_results()

Returns result from running the task

class kitty.data.data_manager.**ReportsTable**(*connection*, *cursor*)
Bases: *kitty.data.data_manager.Table*

Table for storing the reports

__init__(*connection*, *cursor*)

Parameters

- **connection** – the database connection
- **cursor** – the cursor for the database

get(*test_id*)
get report by the test id

Parameters **test_id** – test id

Returns Report object

get_report_test_ids()

Returns ids of test reports

store(*report*, *test_id*)

Parameters

- **report** – the report to store
- **test_id** – the id of the test reported

Returns report id

class kitty.data.data_manager.**SessionInfo**(*orig=None*)
Bases: object

session information manager

__init__(*orig=None*)

Parameters **orig** – SessionInfo object to copy (default: None)

as_dict()

Returns dictionary with the object fields

copy(*orig*)

Parameters **orig** – SessionInfo object to copy

Returns True if changed, false otherwise

fields = ['start_time', 'start_index', 'end_index', 'current_index', 'failure_count', 'kitty_version', 'data_model_hash']

classmethod **from_dict**(*info_d*)

Parameters **info_d** – the info dictionary

Return type *SessionInfo*

Returns object that corresponds to the info dictionary

i = ('data_model_hash', 'INT')

class kitty.data.data_manager.**SessionInfoTable**(*connection*, *cursor*)
Bases: *kitty.data.data_manager.Table*

Table for storing the session info

__init__(*connection*, *cursor*)

Parameters

- **connection** – the database connection
- **cursor** – the cursor for the database

get_session_info()

Return type *SessionInfo*

Returns current session info

read_info()

Return type *SessionInfo*

Returns current session info

set_session_info(*info*)

Parameters **info** (*SessionInfo*) – info to set

class kitty.data.data_manager.**Table**(*connection*, *cursor*)
Bases: object

Base class for data manager tables

__init__(*connection*, *cursor*)

Parameters

- **connection** – the database connection
- **cursor** – the cursor for the database

insert(*fields*, *values*)
insert new db entry

Parameters

- **fields** – list of fields to insert
- **values** – list of values to insert

Returns row id of the new row

row_to_dict(*row*)
translate a row of the current table to dictionary

Parameters **row** – a row of the current table (selected with *)

Returns dictionary of all fields

select (*to_select*, *where=None*, *sql_params=[]*)
select db entries

Parameters

- **to_select** – string of fields to select
- **where** – where clause (default: None)
- **sql_params** – params for the where clause

update (*field_dict*, *where_clause=None*)
update db entry

Parameters

- **field_dict** – dictionary of fields and values
- **where_clause** – where clause for the update

kitty.data.report module

class kitty.data.report.**Report** (*name*, *default_failed=False*)
Bases: object

This class represent a report for a single test. This report may contain subreports from nested entities.

Example In this example, the report, generated by the controller, indicates failure.

```
report = Report('Controller')
report.add('generation time', 0)
report.failed('target does not respont')
```

__init__ (*name*, *default_failed=False*)

Parameters

- **name** – name of the report (or the issuer)
- **default_failed** – is the default status of the report failed (default: False)

add (*key*, *value*)
Add an entry to the report

Parameters

- **key** – entry's key
- **value** – the actual value

Example

```
my_report.add('retriy count', 3)
```

clear ()
Set the report to its defaults. This will clear the report, keeping only the name and setting the failure status to the default.

failed (*reason=None*)
Set the failure status to True, and set the failure reason

Parameters **reason** – failure reason (default: None)

classmethod **from_dict**(*d*, *encoding='base64'*)
Construct a `Report` object from dictionary.

Parameters

- **d** (*dictionary*) – dictionary representing the report
- **encoding** – encoding of strings in the dictionary (default: 'base64')

Returns Report object

get(*key*)
Get a value for a given key

Parameters **key** – entry's key

Returns corresponding value

get_name()

Returns the name of the report

is_failed()

Returns True if the report or any sub report indicates failure.

success()
Set the failure status to False.

to_dict(*encoding='base64'*)
Return a dictionary version of the report

Parameters **encoding** – required encoding for the string values (default: 'base64')

Return type dictionary

Returns dictionary representation of the report

kitty.fuzzers package

The kitty.fuzzers module provides fuzzer classes. In most cases, there is no need to extend those classes, and they may be used as is.

BaseFuzzer should not be instantiated, and only serves as a common parent for *ClientFuzzer* and *ServerFuzzer*.

ClientFuzzer should be used when the fuzzer provides payloads to some server stack in a client fuzzing session.

ServerFuzzer should be used when the fuzzer instantiates the communication, in cases such as fuzzing a server of some sort or when writing payloads to files.

Submodules

kitty.fuzzers.base module
class kitty.fuzzers.base.**BaseFuzzer**(*name=''*, *logger=None*, *option_line=None*)
Bases: *kitty.core.kitty_object.KittyObject*

Common members and logic for client and server fuzzers. This class should not be instantiated, only subclassed.

__init__(*name=''*, *logger=None*, *option_line=None*)

Parameters

- **name** – name of the object
- **logger** – logger for the object (default: None)
- **option_line** – cmd line options to the fuzzer (dafult: None)

set_delay_between_tests(*delay_secs*)
Set duration between tests

Parameters **delay_secs** – delay between tests (in seconds)

set_delay_duration(*delay_duration*)
Deprecated since version use: *set_delay_between_tests()*

set_interface(*interface*)

Parameters **interface** – user interface

set_max_failures(*max_failures*)

Parameters **max_failures** – maximum failures before stopping the fuzzing session

set_model(*model*)
Set the model to fuzz

Parameters **model** (*BaseModel* or a subclass) – Model object to fuzz

set_range(*start_index=0*, *end_index=None*)
Set range of tests to run

Parameters

- **start_index** – index to start at (default=0)
- **end_index** – index to end at(default=None)

set_session_file(*filename*)
Set session file name, to keep state between runs

Parameters **filename** – session file name

set_signal_handler()
Replace the signal handler with self._exit_now

set_store_all_reports(*store_all_reports*)

Parameters **store_all_reports** – should all reports be stored

set_target(*target*)

Parameters **target** – target object

start()
start the fuzzing session

stop()
stop the fuzzing session

unset_signal_handler()
Set the default signal handler

kitty.fuzzers.client module

class kitty.fuzzers.client.**ClientFuzzer**(*name='ClientFuzzer'*, *logger=None*, *option_line=None*)
Bases: *kitty.fuzzers.base.BaseFuzzer*

ClientFuzzer is designed for fuzzing clients. It does not preform an active fuzzing, but rather returns a mutation of a response when in the right state. It is designed to be a module that is integrated into different stacks. See its usage example in the file examples/client_fuzzer_example.py which designed to fuzz a browser.

STAGE_ANY = '****************'**

__init__(*name='ClientFuzzer'*, *logger=None*, *option_line=None*)

Parameters

- **name** – name of the object
- **logger** – logger for the object (default: None)
- **option_line** – cmd line options to the fuzzer

get_mutation(*stage*, *data*)
Get the next mutation, if in the correct stage

Parameters

- **stage** – current stage of the stack
- **data** – a dictionary of items to pass to the model

Returns mutated payload if in apropriate stage, None otherwise

stop()
Stop the fuzzing session

kitty.fuzzers.server module

class kitty.fuzzers.server.**ServerFuzzer**(*name='ServerFuzzer'*, *logger=None*, *option_line=None*)
Bases: *kitty.fuzzers.base.BaseFuzzer*

ServerFuzzer is a class that is designed to fuzz servers. It does not create the mutations, as those are created by the Session object. The idea is to go through every path in the model, execute all requsets in the path, and mutating the last request.

__init__(*name='ServerFuzzer'*, *logger=None*, *option_line=None*)

Parameters

- **name** – name of the object
- **logger** – logger for the object (default: None)
- **option_line** – cmd line options to the fuzzer

kitty.interfaces package

This package provides User Interface classes

BaseInterface is not well-maintained, and should not be used.

WebInterface starts a web server that provides information about the fuzzing state, as well as reports.

Submodules

kitty.interfaces.base module

class kitty.interfaces.base.**BaseInterface**(*name='BaseInterface'*, *logger=None*)
Bases: *kitty.core.kitty_object.KittyObject*

User interface API

__init__(*name='BaseInterface'*, *logger=None*)

Parameters

- **name** – name of the object
- **logger** – logger for the object (default: None)

failure_detected()
handle failure detection

finished()
handle finished

is_paused()

Returns whether current state is paused

pause()
pause the fuzzer

progress()
handle progress

resume()
resume the fuzzer

set_continue_event(*event*)

Parameters **event** – used to control pause/continue

set_data_provider(*data*)

Parameters **data** – the data provider

start()
start the monitor

stop()
stop the monitor

class kitty.interfaces.base.**EmptyInterface**(*name='EmptyInterface'*, *logger=None*)
Bases: *kitty.interfaces.base.BaseInterface*

This interface may be used when there is no need for user interface

__init__(*name='EmptyInterface'*, *logger=None*)

Parameters

- **name** – name of the object
- **logger** – logger for the object (default: None)

failure_detected()
handle failure detection

finished()
handle finished

progress()
handle progress

kitty.interfaces.web module

class kitty.interfaces.web.**WebInterface**(*host='127.0.0.1'*, *port=26000*)

Bases: *kitty.interfaces.base.EmptyInterface*

Web UI for the fuzzer

__init__(*host='127.0.0.1'*, *port=26000*)

Parameters

- **host** – listening address
- **port** – listening port

get_description()

Returns description string (with listening host:port)

kitty.model package

This package contains the entire reference for data and sequence models used by Kitty.

Subpackages

kitty.model.high_level package This package contains the high level data model, which represents sequences and transition between messages.

Submodules

kitty.model.high_level.base module

class kitty.model.high_level.base.**BaseModel**(*name='BaseModel'*)

Bases: *kitty.core.kitty_object.KittyObject*

This class defines the API that is required to be implemented by any top- level model

---

Note: This class should not be instantiated directly.

__init__(*name='BaseModel'*)

Parameters **name** – name of the model (default: BaseModel)

current_index()

Returns current mutation index

get_model_info()

Return type dict

Returns model information

get_sequence()

Return type [Connection]

Returns Sequence of current case

get_sequence_str()

Returns string representation of the sequence

get_test_info()

Return type dict

Returns test information

hash()

Returns a hash of the model object (used for notifying change in the model)

last_index()

Returns the last valid mutation index

mutate()
Mutate to next state

Returns True if mutated, False if not

num_mutations()

Returns number of mutations in the model

skip(*count*)

Parameters **count** – number of cases to skip

Returns number of cases skipped

class kitty.model.high_level.base.**Connection**(*src*, *dst*, *callback*)
Bases: object

A connection between to messages,

__init__(*src*, *dst*, *callback*)

Parameters

- **src** – the source node
- **dst** – the destination node
- **callback** – the function to call after sending src and before sending dst

kitty.model.high_level.graph module Model with a graph structure, all paths in the graph will be fuzzed. The last node in each path will be mutated until exhaustion.

class kitty.model.high_level.graph.**GraphModel**(*name='GraphModel'*)
Bases: *kitty.model.high_level.base.BaseModel*

The GraphModel is built of a simple digraph, where the nodes are templates, and on each edge there's a callback function. It will provide sequences of edges from this graph, always starting from the root (dummy) node, where the last template in the sequence (the destination of the last edge) is mutated. As such the main target of the GraphModel is to fuzz the handling of the fields in a message.

Assuming we have the templates A, B, C and D and we want to fuzz all templates, but we know that in order to make an impact in fuzzing template D, we first need to send A, then B or C, and only then D, e.g. we have the following template graph:

Which translate to the following sequences (* - mutated template):

```
A*
A -> B*
A -> B -> D*
A -> C*
A -> C -> D*
```

Such a model will be written in Kitty like this:

Example

```
model = GraphModel('MyGraphModel')
model.connect(A)
model.connect(A, B)
model.connect(B, D)
model.connect(A, C)
model.connect(C, D)
```

Note: Make sure there are no loops in your model!!

The callback argument of connect allows monitoring and maintainance during a fuzzing test.

Example

```
def log_if_empty_response(fuzzer, edge, response):
    if not response:
        logger.error('Got an empty response for request %s' % edge.src.get_name())

model.connect(A, B, log_if_empty_response)
```

__init__(*name='GraphModel'*)

Parameters **name** – name for this model

check_loops_in_grpah(*current=None*, *visited=[]*)

Parameters

- **current** – current node to check if visited
- **visited** – list of visited fields

Raise KittyException if loop found

connect(*src*, *dst=None*, *callback=None*)

Parameters

- **src** – source node, if dst is None it will be destination node and the root (dummy) node will be source
- **dst** – destination node (default: None)
- **callback** (*func(fuzzer, edge, response) -> None*) – a function to be called after the response for src received and before dst is sent

get_model_info()

get_test_info()

hash()

skip(*count*)

kitty.model.high_level.random_sequence module

class kitty.model.high_level.random_sequence.**RandomSequenceModel**(*name='RandomSequenceModel'*, *seed=None*, *callback_generator=None*, *num_mutations=1000*, *max_sequence=10*)

Bases: *kitty.model.high_level.staged_sequence.StagedSequenceModel*

This class provides random sequences of templates based on the selection strategy. It is like *StagedSequenceModel* with a signle stage.

__init__(*name='RandomSequenceModel'*, *seed=None*, *callback_generator=None*, *num_mutations=1000*, *max_sequence=10*)

Parameters

- **name** – name of the model object (default: 'RandomSequenceModel')
- **seed** (*int*) – RNG seed (default: None)
- **callback_generator** (*func(from_template, to_template) -> func(fuzzer, edge, response) -> None*) – a function that returns callback functions (default: None)
- **num_mutations** – number of mutations to perform (defualt: 1000)
- **max_sequence** – maximum sequence length (default: 10)

add_template(*template*)

Add template that might be used in generated sequences

Parameters **template** – template to add to the model

kitty.model.high_level.staged_sequence module Generate random sequences of messages for a session. Currently, perform no special operation for mutations on the nodes.

class kitty.model.high_level.staged_sequence.**Stage**(*name*, *selection_strategy='1'*, *seed=None*)

Bases: *kitty.core.kitty_object.KittyObject*

Stage supports 4 different sequence length strategy. For all of them, the order of templates will be random. The available strategies are:

•for random length - 'random'

•for exact length - '12'

•for length in range - '1-3'

•for all - 'all'

__init__(*name*, *selection_strategy='1'*, *seed=None*)

Parameters

- **name** – name of the Stage
- **selection_strategy** – strategy for selecting amount of template in each mutation
- **seed** – RNG seed (default: None)

add_template(*template*)

Parameters **template** – template to add to the stage

get_sequence_templates()

Returns templates of current sequence mutation

get_templates()

Returns list of the stage's templates

hash()

mutate()

class kitty.model.high_level.staged_sequence.**StagedSequenceModel**(*name='StagedSequenceModel'*, *callback_generator=None*, *num_mutations=1000*)

Bases: *kitty.model.high_level.base.BaseModel*

The StagedSequenceModel provides sequences that are constructed from multiple stages of sequences. Each such stage provides its part of the sequence based on its strategy. The main goal of the Staged sequence mode is to find flaws in the state handling of a stateful target.

Example Assuming we have the templates [CreateA .. CreateZ, UseA .. UseZ and DeleteA .. DeleteZ]. A valid sequence is **CreateA -> UseA -> DeleteA**, so we want to test cases like **CreateA -> UseB -> UseA -> DeleteC**, or **CreateA -> DeleteA -> UseA**. And let's say the common thing to all sequences is that we always want to start by sending one or more of the Create messages.

We can do that with StagedSequenceModel:

```
stage1 = Stage('create', selection_strategy='1-5')
stage1.add_Template(CreateA)
# ...
stage1.add_Template(CreateZ)
stage2 = Stage('use and delete', selection_strategy='3-20')
stage2.add_Template(UseA)
stage2.add_Template(DeleteA)
# ...
stage2.add_Template(UseZ)
stage2.add_Template(DeleteZ)

model = StagedSequenceModel('use and delete mess', num_mutations=10000)
model.add_stage(stage1)
model.add_stage(stage2)
```

Each time it is asked, it will provide a sequence that starts with 1-5 random "Create" templates, followed by 3-20 random Use and Delete messages. None of those templates will be mutated, as we try to fuzz the sequence itself, not the message structure.

Since we don't know the what will be the order of templates in the sequences, we can't just provide a callback as we do in GraphModel. The solution in our case is to provide the model with a callback generator, which receives the from_template and to_template and returns a callback function as described in GraphModel in runtime.

__init__(*name='StagedSequenceModel'*, *callback_generator=None*, *num_mutations=1000*)

Parameters

- **name** – name of the model object (default: 'StagedSequenceModel')

- **callback_generator** (*func(from_template, to_template) -> func(fuzzer, edge, response) -> None*) – a function that returns callback functions
- **num_mutations** – number of mutations to perform

add_stage(*stage*)
add a stage. the order of stage is preserved

Parameters **stage** (Stage) – the stage to add

get_model_info()

Returns dictionary of information about this model

get_test_info()

Returns dictionary of information about the current test

hash()

kitty.model.low_level package This package contains the low level data model, which represents the structure of specific messages in the fuzzed protocol.

Submodules

kitty.model.low_level.aliases module Functions that provide simpler field creation API and name aliasing for convenience

Integer Aliases Since integers the are binary encoded are used in many places, there are many aliases for them. Some are similar to common type conventions, others are simply meant to make the template code smaller. Below is a (hopefully) full list of aliases. However, the safest place to check, as always, is the source.

Bytes	Signed	Endianess	Aliases
1	No		U8, UInt8, BE8, LE8, Byte
2	No	big, but can be changed	U16, UInt16, Word
2		big	BE16, WordBE
2		little	LE16, WordLE
2	Yes	big, but can be changed	S16, SInt16
4	No	big, but can be changed	U32, UInt32, Dword
4		big	BE32, DwordBE
4		little	LE32, DwordLE
4	Yes	big, but can be changed	S32, SInt32
8	No	big, but can be changed	U64, UInt64, Qword
8		big	BE64, QwordBE
8		little	LE64, QwordLE
8	Yes	big, but can be changed	S64, SInt64

Hash Aliases There are multiple functions that return frequently used Hash instances:

- *Md5*
- *Sha1*
- *Sha224*
- *Sha256*
- *Sha384*

- *Sha512*.

Their prototype is the same as *Hash*, excluding the *algorithm* argument.

Compare Aliases The *Compare* aliases create a Compare object with a specific comp_type. See table below:

comp_type	Aliases
'=='	Equal
'!='	NotEqual
'>'	Greater
'>='	GreaterEqual, AtLeast
'<'	Lesser
'<='	LesserEqual, AtMost

kitty.model.low_level.aliases.**AtLeast**(*field*, *comp_value*)
Condition applies if the value of the field is greater than or equals to the comp_value

kitty.model.low_level.aliases.**AtMost**(*field*, *comp_value*)
Condition applies if the value of the field is lesser than or equals to the comp_value

kitty.model.low_level.aliases.**BE16**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**BE32**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**BE64**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**BE8**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**Byte**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**Dword**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**DwordBE**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**DwordLE**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**Equal**(*field*, *comp_value*)
Condition applies if the value of the field is equal to the comp_value

kitty.model.low_level.aliases.**Greater**(*field*, *comp_value*)
Condition applies if the value of the field is greater than the comp_value

kitty.model.low_level.aliases.**GreaterEqual**(*field*, *comp_value*)
Condition applies if the value of the field is greater than or equals to the comp_value

kitty.model.low_level.aliases.**LE16**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**LE32**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**LE64**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**LE8**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**Lesser**(*field*, *comp_value*)
Condition applies if the value of the field is lesser than the comp_value

kitty.model.low_level.aliases.**LesserEqual**(*field*, *comp_value*)
Condition applies if the value of the field is lesser than or equals to the comp_value

kitty.model.low_level.aliases.**Md5**(*depends_on*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

kitty.model.low_level.aliases.**NotEqual**(*field*, *comp_value*)
Condition applies if the value of the field is not equal to the comp_value

kitty.model.low_level.aliases.**Qword**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**QwordBE**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**QwordLE**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**S16**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**S32**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**S64**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**S8**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**SInt16**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**SInt32**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**SInt64**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**SInt8**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**Sha1**(*depends_on*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

kitty.model.low_level.aliases.**Sha224**(*depends_on*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

kitty.model.low_level.aliases.**Sha256**(*depends_on*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

kitty.model.low_level.aliases.**Sha384**(*depends_on*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

kitty.model.low_level.aliases.**Sha512**(*depends_on*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

kitty.model.low_level.aliases.**SizeInBytes**(*sized_field*, *length*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=False*, *name=None*)

kitty.model.low_level.aliases.**U16**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**U32**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**U64**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**U8**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**UInt16**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**UInt32**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**UInt64**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**UInt8**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**Word**(*value*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**WordBE**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.aliases.**WordLE**(*value*, *min_value=None*, *max_value=None*, *fuzzable=True*, *name=None*)

kitty.model.low_level.condition module conditon object has one mandatory function - applies, which receives a Container as the single argument.

Conditions are used by the *If* and *IfNot* fields to decide wether to render their content or not. In many cases the decision is made based on a value of a specific field, but you can create whatever condition you want. In future versions, they will probably be used to make other decisions, not only basic rendering decisions.

class kitty.model.low_level.condition.**Compare**(*field*, *comp_type*, *comp_value*)

Bases: *kitty.model.low_level.condition.FieldContidion*

Condition applies if the comparison between the value of the field and the comp_value evaluates to True

Note: There are some functions for creating specific *Compare* conditions that you can see below.

__init__ (*field*, *comp_type*, *comp_value*)

Parameters

- **field** – (name of) field that should meet the condition
- **comp_type** – how to compare the values. One of ('>', '<', '>=', '<=', '==', '!=')
- **comp_value** – value to compare the field to

Example

```
Template([
    String(['kitty'], name='name'),
    If(Compare('name', '!=', 'kitty'), [
        String('123')
    ])
])
```

hash()

class kitty.model.low_level.condition.**Condition**
Bases: object

applies (*container*)
All subclasses must implement the *applies(self, container)* method

Parameters **container** (Container) – the caller

Returns True if condition applies, False otherwise

hash()

class kitty.model.low_level.condition.**FieldContidion** (*field*)
Bases: *kitty.model.low_level.condition.Condition*

Base class for field-based conditions (if field meets condition return true)

__init__ (*field*)

Parameters **field** (*BaseField* or str) – (name of, or) field that should meet the condition

applies (*container*)
Subclasses should not override *applies*, but instead they should override *_applies*, which has the same syntax as *applies*. In the *_applies* method the condition is guaranteed to have a reference to the desired field, as *self._field*.

Parameters **container** (*Container*) – the caller

Returns True if condition applies, False otherwise

hash()

invalidate()

class kitty.model.low_level.condition.**FieldMutating** (*field*)
Bases: *kitty.model.low_level.condition.FieldContidion*

Condition applies if the field is currently mutating

class kitty.model.low_level.condition.**InList** (*field*, *value_list*)
Bases: *kitty.model.low_level.condition.ListCondition*

Condition applies if the value of the field appears in the value list

Example

```
Template([
    Group(['1', '2', '3'], name='numbers'),
    If(InList('numbers', ['1', '5', '7']), [
        String('123')
    ])
])
```

class kitty.model.low_level.condition.**ListCondition**(*field*, *value_list*)

Bases: *kitty.model.low_level.condition.FieldContidion*

Base class for comparison between field and list. can't be instantiated

__init__(*field*, *value_list*)

Parameters

- **field** – (name of) field that should meet the condition
- **value_list** – list of values that should be compared to the field

hash()

kitty.model.low_level.container module Containers are fields that group multiple fields into a single logical unit, they all inherit from `Container`, which inherits from `BaseField`.

class kitty.model.low_level.container.**Container**(*fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.field.BaseField*

A logical unit to group multiple fields together

__init__(*fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **fields** (*field or iterable of fields*) – enclosed field(s) (default: [])
- **encoder** (`BitsEncoder`) – encoder for the container (default: ENC_BITS_DEFAULT)
- **fuzzable** – is container fuzzable (default: True)
- **name** – (unique) name of the container (default: None)

Example

```
Container([
    String('header_name'),
    Delimiter('='),
    String('header_value')
])
```

append_fields(*new_fields*)

Add fields to the container

Parameters **new_fields** – fields to append

copy()

Returns a copy of the container

get_info()
Get info regarding the current fuzzed enclosed node

Returns info dictionary

get_tree(*depth=0*)
Get a string representation of the field tree

Parameters **depth** – current depth in the tree (default:0)

Returns string representing the field tree

hash()

num_mutations()

Returns number of mutations in the container

pop()
Remove a the top container from the container stack

push(*field*)
Add a field to the container, if the field is a Container itself, it should be poped() when done pushing into it

Parameters **field** – BaseField to push

render()

Returns rendered value of the container

replace_fields(*new_fields*)
Remove all fields from the container and add new fields

Parameters **new_fields** – fields to add to the container

reset()
Reset the state of the container and its internal fields

resolve_field(*field*)
Resolve a field from name

Parameters **field** – name of the field to resolve

Return type *BaseField*

Returns the resolved field

Raises KittyException if field could not be resolved

scan_for_field(*field_key*)
Scan for a field in the container and its enclosed fields

Parameters **field_key** – name of field to look for

Returns field with name that matches field_key, None if not found

set_session_data(*session_data*)
Set session data in the container enclosed fields

Parameters **session_data** – dictionary of session data

class kitty.model.low_level.container.**ForEach**(*mutated_field*, *fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)
Bases: *kitty.model.low_level.container.Container*

Perform all mutations of enclosed fields for each mutation of mutated_field

__init__(*mutated_field*, *fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **mutated_field** – (name of) field to perform mutations for each of its mutations
- **fields** – enclosed field(s) (default: [])
- **encoder** (BitsEncoder) – encoder for the container (default: ENC_BITS_DEFAULT)
- **fuzzable** – is container fuzzable (default: True)
- **name** – (unique) name of the container (default: None)

Example

```
Template([
    Group(['a', 'b', 'c'], name='letters'),
    ForEach('letters', [
        Group(['1', '2', '3'])
    ])
])
# results in the mutations: a1, a2, a3, b1, b2, b3, c1, c2, c3
```

hash()

reset(*reset_mutated=True*)
reset the state of the container and its internal fields

Parameters **reset_mutated** – should reset the mutated field too (default: True)

class kitty.model.low_level.container.**If**(*condition*, *fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.container.Container*

Render only if condition evalutes to True

__init__(*condition*, *fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **condition** (*an object that has a function applies(self, Container) -> Boolean*) – condition to evaluate
- **fields** – enclosed field(s) (default: [])
- **encoder** (BitsEncoder) – encoder for the container (default: ENC_BITS_DEFAULT)
- **fuzzable** – is container fuzzable (default: True)
- **name** – (unique) name of the container (default: None)

Example

```
Template([
    Group(['a', 'b', 'c'], name='letters'),
    If(ConditionCompare('letters', '==', 'a'), [
        Static('dvil')
    ])
```

```
    ])
    # results in the mutations: advil, b, c
```

copy()
Copy the container, put an invalidated copy of the condition in the new container

hash()

render()
Only render if condition applies

class kitty.model.low_level.container.**IfNot**(*condition*, *fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)
Bases: *kitty.model.low_level.container.Container*

Render only if condition evalutes to False

__init__(*condition*, *fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **condition** (*an object that has a function applies(self, Container) -> Boolean*) – condition to evaluate
- **fields** – enclosed field(s) (default: [])
- **encoder** (BitsEncoder) – encoder for the container (default: ENC_BITS_DEFAULT)
- **fuzzable** – is container fuzzable (default: True)
- **name** – (unique) name of the container (default: None)

Example

```
Template([
    Group(['a', 'b', 'c'], name='letters'),
    IfNot(ConditionCompare('letters', '==', 'a'), [
        Static('ar')
    ])
])
# results in the mutations: a, bar, car
```

copy()
Copy the container, put an invalidated copy of the condition in the new container

hash()

render()
Only render if condition applies

class kitty.model.low_level.container.**Meta**(*fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)
Bases: *kitty.model.low_level.container.Container*

Don't render enclosed fields

Example

```
            Container([
                Static('no sp'),
                Meta([Static(' ')]),
                Static('ace')
            ])
            # will render to: 'no space'
```

render()

Returns empty Bits

class kitty.model.low_level.container.**OneOf**(*fields=[]*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.container.Container*

Render a single field from the fields (also mutates only one field each time)

render()

Render only the mutated field (or the first one if not in mutation)

class kitty.model.low_level.container.**Pad**(*pad_length*, *pad_data='x00'*, *fields=[]*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.container.Container*

Pad the rendered value of the enclosed fields

__init__(*pad_length*, *pad_data='\x00'*, *fields=[]*, *fuzzable=True*, *name=None*)

Parameters

- **pad_length** – length to pad up to (in bits)
- **pad_data** – data to pad with (default: '')
- **fields** – enclosed field(s) (default: [])
- **fuzzable** – is fuzzable (default: True)
- **name** – (unique) name of the template (default: None)

hash()

render()

Returns enclosed fields with padding

class kitty.model.low_level.container.**Repeat**(*fields=[]*, *min_times=1*, *max_times=1*, *step=1*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.container.Container*

Repeat the enclosed fields. When not mutated, the repeat count is min_times

__init__(*fields=[]*, *min_times=1*, *max_times=1*, *step=1*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **fields** – enclosed field(s) (default: [])
- **min_times** – minimum number of repetitions (default: 1)
- **max_times** – maximum number of repetitions (default: 1)
- **step** – how many repetitions to add each mutation (default: 1)

- **encoder** (BitsEncoder) – encoder for the container (default: ENC_BITS_DEFAULT)
- **fuzzable** – is container fuzzable (default: True)
- **name** – (unique) name of the container (default: None)

Examples

```
Repeat([Static('a')], min_times=5, fuzzable=False)
# will render to: 'aaaaa'
Repeat([Static('a')], min_times=5, max_times=10, step=5)
# will render to: 'aaaaa', 'aaaaaaaaaa'
```

hash()

render()

class kitty.model.low_level.container.**TakeFrom**(*fields=[]*, *min_elements=1*, *max_elements=None*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.container.OneOf*

Render to only part of the enclosed fields, performing all mutations on them

__init__(*fields=[]*, *min_elements=1*, *max_elements=None*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **fields** (*field or iterable of fields*) – enclosed field(s) (default: [])
- **min_elements** – minimum number of elements in the sub set
- **max_elements** – maximum number of elements in the sub set
- **encoder** (BitsEncoder) – encoder for the container (default: ENC_BITS_DEFAULT)
- **fuzzable** – is container fuzzable (default: True)
- **name** – (unique) name of the container (default: None)

hash()

render()

reset()

class kitty.model.low_level.container.**Template**(*fields=[]*, *encoder=<kitty.model.low_level.encoder.ByteAlignedBitsEncoder object>*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.container.Container*

Top most container of a message, serves a the only interface to the high level model

__init__(*fields=[]*, *encoder=<kitty.model.low_level.encoder.ByteAlignedBitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **fields** – enclosed field(s) (default: [])
- **encoder** (BitsEncoder) – encoder for the container (default: ENC_BITS_BYTE_ALIGNED)

- **fuzzable** – is fuzzable (default: True)
- **name** – (unique) name of the template (default: None)

Example

```
Template([
    Group(['a', 'b', 'c']),
    Group(['1', '2', '3'])
])

# the mutations are: a1, b1, c1, a1, a2, a3
```

copy()
We might want to change it in the future, but for now...

get_info()

class kitty.model.low_level.container.**Trunc**(*max_size*, *fields=[]*, *fuzzable=True*, *name=None*)
Bases: *kitty.model.low_level.container.Container*

Truncate the size of the enclosed fields

__init__(*max_size*, *fields=[]*, *fuzzable=True*, *name=None*)

Parameters

- **max_size** – maximum size of the container (in bits)
- **fields** – enclosed field(s) (default: [])
- **fuzzable** – is fuzzable (default: True)
- **name** – (unique) name of the template (default: None)

hash()

render()

kitty.model.low_level.encoder module Encoders are used for encoding fields and containers. The encoders are passed as an argument to the fields/container, during the field rendering, the encoder's *encode* method is called.

There are three families of encoders:

Bits Encoders Used to encode fields/containers that their value is of type *Bits* (Container, ForEach etc.)

String Encoders Used to encode fields that their value is of type *str* (String, Delimiter, RandomBytes etc.)

BitField Encoders Used to encode fields that inherit from BitField or contain BitField (UInt8, Size, Checksum etc.) Those encoders are also refferred to as Int Encoders.

class kitty.model.low_level.encoder.**BitFieldAsciiEncoder**(*fmt*)
Bases: *kitty.model.low_level.encoder.BitFieldEncoder*

Encode int as ascii

__init__(*fmt*)

Parameters **fmt** – format for encoding (from BitFieldAsciiEncoder.formats)

encode(*value*, *length*, *signed*)

formats = ['%d', '%x', '%X', '%#x', '%#X']

class kitty.model.low_level.encoder.**BitFieldBinEncoder**(*mode*)
Bases: *kitty.model.low_level.encoder.BitFieldEncoder*

Encode int as binary

__init__(*mode*)

Parameters **mode** (*str*) – mode of binary encoding. 'le' for little endian, 'be' for big endian, '' for non-byte aligned

encode(*value*, *length*, *signed*)

Parameters

- **value** – value to encode
- **length** – length of value in bits
- **signed** – is value signed

class kitty.model.low_level.encoder.**BitFieldEncoder**
Bases: object

Base encoder class for BitField values

Singleton Name	Encoding	Class
ENC_INT_BIN	Encode as binary bits	BitFieldBinEncoder
ENC_INT_LE	Encode as a little endian binary bits	BitFieldBinEncoder
ENC_INT_BE	Encode as a big endian binary bits	
ENC_INT_DEC	Encode as a decimal value	BitFieldAsciiEncoder
ENC_INT_HEX	Encode as a hex value	
ENC_INT_HEX_UPPER	Encode as an upper case hex value	
ENC_INT_DEFAULT	Same as ENC_INT_BIN	

encode(*value*, *length*, *signed*)

Parameters

- **value** (int) – value to encode
- **length** (int) – length of value in bits
- **signed** (boolean) – is value signed

class kitty.model.low_level.encoder.**BitFieldMultiByteEncoder**(*mode='be'*)
Bases: *kitty.model.low_level.encoder.BitFieldEncoder*

Encode int as multi-byte (used in WBXML format)

__init__(*mode='be'*)

Parameters **mode** (*str*) – mode of binary encoding. 'le' for little endian, 'be' for big endian, '' for non-byte aligned

encode(*value*, *length*, *signed*)

Parameters

- **value** – value to encode
- **length** – length of value in bits
- **signed** – is value signed

class kitty.model.low_level.encoder.**BitsEncoder**
Bases: object

Base encoder class for Bits values

The Bits encoders *encode* function receives a *Bits* object as an argument and returns an encoded *Bits* object.

Singleton Name	Encoding	Class
ENC_BITS_NONE	None, returns the same value received	BitsEncoder
ENC_BITS_BYTE_ALIGNED	Appends bits to the received object to make it byte aligned	ByteAligned-BitsEncoder
ENC_BITS_REVERSE	Reverse the order of bits	ReverseBitsEncoder
ENC_BITS_BASE64	Encode a Byte aligned bits in base64	StrEncoderWrapper
ENC_BITS_BASE64_NO_NL	Encode a Byte aligned bits in base64, but removes the new line from the end	
ENC_BITS_UTF8	Encode a Byte aligned bits in UTF-8	
ENC_BITS_HEX	Encode a Byte aligned bits in hex	
ENC_BITS_DEFAULT	Same as ENC_BITS_NONE	

encode(*value*)

Parameters **value** (*Bits*) – value to encode

class kitty.model.low_level.encoder.**BitsFuncEncoder**(*func*)
Bases: *kitty.model.low_level.encoder.StrEncoder*

Encode bits using a given function

__init__(*func*)

Parameters **func** – encoder function(Bits)->Bits

encode(*value*)

class kitty.model.low_level.encoder.**ByteAlignedBitsEncoder**
Bases: *kitty.model.low_level.encoder.BitsEncoder*

Stuff bits for byte alignment

encode(*value*)

Parameters **value** – value to encode

class kitty.model.low_level.encoder.**ReverseBitsEncoder**
Bases: *kitty.model.low_level.encoder.BitsEncoder*

Reverse the order of bits

encode(*value*)

Parameters **value** – value to encode

class kitty.model.low_level.encoder.**StrBase64NoNewLineEncoder**
Bases: *kitty.model.low_level.encoder.StrEncoder*

Encode the string as base64, but without the new line at the end

encode(*value*)

Parameters **value** – value to encode

class kitty.model.low_level.encoder.**StrEncodeEncoder**(*encoding*)
Bases: *kitty.model.low_level.encoder.StrEncoder*

Encode the string using str.encode function

__init__(*encoding*)

Parameters **encoding** (str) – encoding to be used, should be a valid argument for str.encode

encode(*value*)

Parameters **value** – value to encode

class kitty.model.low_level.encoder.**StrEncoder**
Bases: object

Base encoder class for str values The String encoders *encode* function receives a *str* object as an argument and returns an encoded *Bits* object.

Singleton Name	Encoding	Class
ENC_STR_UTF8	Encode the str in UTF-8	StrEncodeEncoder
ENC_STR_HEX	Encode the str in hex	
ENC_STR_BASE64	Encode the str in base64	
ENC_STR_BASE64_NO_NL	Encode the str in base64 but remove the new line from the end	StrBase64NoNewLineEncoder
ENC_STR_DEFAULT	Do nothing, just convert the str to Bits object	StrEncoder

encode(*value*)

Parameters **value** (str) – value to encode

class kitty.model.low_level.encoder.**StrEncoderWrapper**(*encoder*)
Bases: *kitty.model.low_level.encoder.ByteAlignedBitsEncoder*

Encode the data using str.encode function

__init__(*encoder*)

Parameters **encoding** (StrEncoder) – encoder to wrap

encode(*value*)

Parameters **value** – value to encode

class kitty.model.low_level.encoder.**StrFuncEncoder**(*func*)
Bases: *kitty.model.low_level.encoder.StrEncoder*

Encode string using a given function

__init__(*func*)

Parameters **func** – encoder function(str)->str

encode(*value*)

class kitty.model.low_level.encoder.**StrNullTerminatedEncoder**
Bases: *kitty.model.low_level.encoder.StrEncoder*

Encode the string as c-string, with null at the end

encode(*value*)

Parameters **value** – value to encode

kitty.model.low_level.field module This module is the "Heart" of the data model. It contains all the basic building blocks for a Template. Each "field" type is a discrete component in the full Template.

class kitty.model.low_level.field.**BaseField**(*value*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)
Bases: *kitty.core.kitty_object.KittyObject*

Basic type for all fields and containers, it contains the common logic. This class should never be used directly.

__init__(*value*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **value** – default value
- **encoder** (BaseEncoder) – encoder for the field
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

copy()

Returns a copy of the field

get_current_value()

Returns current value

get_info()

Return type dictionary

Returns field information

hash()

mutate()
Mutate the field

Return type boolean

Returns True if field the mutated

num_mutations()

Returns number of mutation in this field

render()
Render the current value of the field

Return type Bits

Returns rendered value

reset()
Reset the field to its default state

resolve_field(*field*)
Resolve a field from name

Parameters **field** – name of the field to resolve

Return type *BaseField*

Returns the resolved field

Raises KittyException if field could not be resolved

scan_for_field(*field_name*)
Scan for field field with given name

Parameters **field_name** – field name to look for

Returns None

set_current_value(*value*)
Sets the current value of the field

Parameters **value** – value to set

Returns rendered value

skip(*count*)
Skip up to [count] cases, default behavior is to just mutate [count] times

Count number of cases to skip

Return type int

Returns number of cases skipped

class kitty.model.low_level.field.**BitField**(*value*, *length*, *signed=False*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)
Bases: kitty.model.low_level.field._LibraryField

Represents a fixed-length sequence of bits, the mutations target common integer related vulnerabilities

Note: Since BitField is frequently used in binary format, multiple aliases were created for it. See aliases.py for more details.

__init__(*value*, *length*, *signed=False*, *min_value=None*, *max_value=None*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*int*) – default value
- **length** (*positive int*) – length of field in bits
- **signed** – are the values signed (default: False)
- **min_value** – minimal allowed value (default: None)
- **max_value** – maximal allowed value (default: None)
- **encoder** (BitFieldEncoder) – encoder for the field
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

Examples

```
BitField(123, length=15, signed=True, max_value=1000)
UInt8(17, encoder=ENC_INT_DEC)
```

hash()

lib = None

class kitty.model.low_level.field.**Calculated**(*depends_on*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)
Bases: *kitty.model.low_level.field.BaseField*

A base type for fields that are calculated based on other fields

__init__(*depends_on*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **depends_on** – (name of) field we depend on
- **encoder** (BitsEncoder) – encoder for the field
- **fuzzable** – is container fuzzable
- **name** – (unique) name of the container

hash()

render()
Render the current value into a bitstring.Bits object

Return type bitstring.Bits

Returns the rendered field

class kitty.model.low_level.field.**CalculatedBits**(*depends_on*, *func*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)
Bases: *kitty.model.low_level.field.Calculated*

field that depends on the rendered value of a field, and rendered into Bits() object

__init__(*depends_on*, *func*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **depends_on** – (name of) field we depend on
- **func** – function for processing of the dependant data. func(Bits)->Bits
- **encoder** (BitsEncoder) – encoder for the field
- **fuzzable** – is container fuzzable
- **name** – (unique) name of the container

class kitty.model.low_level.field.**CalculatedInt**(*depends_on*, *bit_field*, *calc_func*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=False*, *name=None*)
Bases: *kitty.model.low_level.field.Calculated*

field that depends on the rendered value of another field and is rendered to (int, length, signed) tuple

__init__(*depends_on*, *bit_field*, *calc_func*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=False*, *name=None*)

Parameters

- **depends_on** – (name of) field we depend on
- **bit_field** – a BitField to be used for holding the value
- **calc_func** – function to calculate the value of the field. func(bits) -> int
- **encoder** (BitsEncoder) – encoder for the field (default: ENC_BITS_DEFAULT)
- **fuzzable** – is container fuzzable
- **name** – (unique) name of the container

reset()

scan_for_field(*field_name*)
If the field name is the internal field - return it

class kitty.model.low_level.field.**CalculatedStr**(*depends_on*, *func*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

Bases: *kitty.model.low_level.field.Calculated*

field that depends on the rendered value of a byte-aligned field and rendered to a byte aligned Bits() object

__init__(*depends_on*, *func*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

Parameters

- **depends_on** – (name of) field we depend on
- **func** – function for processing of the dependant data. func(str)->str
- **encoder** (StrEncoder) – encoder for the field (default: ENC_STR_DEFAULT)
- **fuzzable** – is container fuzzable
- **name** – (unique) name of the container

class kitty.model.low_level.field.**Checksum**(*depends_on*, *length*, *algorithm='crc32'*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=False*, *name=None*)

Bases: *kitty.model.low_level.field.CalculatedInt*

Checksum of another container.

__init__(*depends_on*, *length*, *algorithm='crc32'*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=False*, *name=None*)

Parameters

- **depends_on** – (name of) field to be checksummed
- **length** – length of the checksum field (in bits)
- **algorithm** – checksum algorithm name (from Checksum._algos) or a function to calculate the value of the field. func(Bits) -> int
- **encoder** (BitFieldEncoder) – encoder for the field (default: ENC_INT_DEFAULT)
- **fuzzable** – is field fuzzable (default: False)
- **name** – (unique) name of the field (default: None)

class kitty.model.low_level.field.**Clone**(*depends_on*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=False*, *name=None*)

Bases: *kitty.model.low_level.field.CalculatedBits*

rendered the same as the field it depends on

__init__(*depends_on*, *encoder=<kitty.model.low_level.encoder.BitsEncoder object>*, *fuzzable=False*, *name=None*)

Parameters

- **depends_on** – (name of) field we depend on
- **encoder** (BitsEncoder) – encoder for the field (default: ENC_BITS_DEFAULT)
- **fuzzable** – is container fuzzable
- **name** – (unique) name of the container

class kitty.model.low_level.field.**Delimiter**(*value*, *max_size=None*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.field.String*

Represent a text delimiter, the mutations target common delimiter-related vulnerabilities

__init__(*value*, *max_size=None*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – default value
- **max_size** – maximal size of the string before encoding (default: None)
- **encoder** (StrEncoder) – encoder for the field (default: ENC_STR_DEFAULT)
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

Example

```
Delimiter('=', max_size=30, encoder=ENC_STR_BASE64)
```

lib = None

class kitty.model.low_level.field.**Dynamic**(*key*, *default_value*, *length=None*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

Bases: *kitty.model.low_level.field.BaseField*

A field that gets its value from the fuzzer at runtime

__init__(*key*, *default_value*, *length=None*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

Parameters

- **key** (*str*) – key for the data in the session_data dictionary
- **default_value** (*str*) – default value of the field
- **length** – length of the field in bytes. must be set if fuzzable=True (default: None)
- **encoder** (StrEncoder) – encoder for the field (default: ENC_STR_DEFAULT)
- **fuzzable** – is field fuzzable (default: False)
- **name** – name of the object (default: None)

Examples

```
Dynamic(key='session id', default_value='{}')
Dynamic(key='session id', default_value='{}', length=4, fuzzable=True)
```

hash()

render()

set_session_data(*session_data*)

skip(*count*)

class kitty.model.low_level.field.**Group**(*values*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=True*, *name=None*)

Bases: kitty.model.low_level.field._LibraryField

A field with fixed set of possible mutations

__init__(*values*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **values** (*list of strings*) – possible values for the field
- **encoder** (StrEncoder) – encoder for the field (default: ENC_STR_DEFAULT)
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

Example

```
Group(['GET', 'PUT', 'POST'], name='http methods')
```

hash()

lib = None

class kitty.model.low_level.field.**Hash**(*depends_on*, *algorithm*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

Bases: *kitty.model.low_level.field.CalculatedStr*

Hash of a field.

Note: To make it more convenient, there are multiple aliases for various hashes. Take a look at *aliases*.

__init__(*depends_on*, *algorithm*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=False*, *name=None*)

Parameters

- **depends_on** – (name of) field to be hashed
- **algorithm** – hash algorithm name (from Hash._algos) or a function to calculate the value of the field. func(str) -> str
- **encoder** (StrEncoder) – encoder for the field (default: ENC_STR_DEFAULT)
- **fuzzable** – is field fuzzable (default: False)
- **name** – (unique) name of the field (default: None)

class kitty.model.low_level.field.**RandomBytes**(*value*, *min_length*, *max_length*, *seed=1234*, *num_mutations=25*, *step=None*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.field.BaseField*

A random sequence of bytes The length of the sequence is between *min_length* and *max_length*, and decided either randomally (if *step* is *None*) or starts from *min_length* and inreased by *step* bytes (if *step* has a value).

__init__(*value*, *min_length*, *max_length*, *seed=1234*, *num_mutations=25*, *step=None*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – default value
- **min_length** – minimal length of the field (in bytes)
- **max_length** – maximal length of the field (in bytes)

- **seed** – seed for the random number generator, to allow consistency between runs (default: 1234)
- **num_mutations** – number of mutations to perform (if step is None) (default:25)
- **step** (*int*) – step between lengths of each mutation (default: None)
- **encoder** (StrEncoder) – encoder for the field (default: ENC_STR_DEFAULT)
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

Examples

```
RandomBytes(value='1234', min_length=0, max_length=75, step=15)
RandomBytes(value='1234', min_length=0, max_length=75, num_mutations=80)
```

hash()

reset()

class kitty.model.low_level.field.**Size**(*sized_field*, *length*, *calc_func=<function <lambda>>*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=False*, *name=None*)

Bases: *kitty.model.low_level.field.CalculatedInt*

Size of another container. Calculated in each render call

Note: In most cases you can use the function *SizeInBytes()* instead, which receives the same arguments except of *calc_func*

__init__(*sized_field*, *length*, *calc_func=<function <lambda>>*, *encoder=<kitty.model.low_level.encoder.BitFieldBinEncoder object>*, *fuzzable=False*, *name=None*)

Parameters

- **sized_field** – (name of) field to be sized
- **length** – length of the size field (in bits)
- **calc_func** – function to calculate the value of the field. func(bits) -> int (default: length in bytes)
- **encoder** (BitFieldEncoder) – encoder for the field (default: ENC_INT_DEFAULT)
- **fuzzable** – is field fuzzable (default: False)
- **name** – (unique) name of the field (default: None)

class kitty.model.low_level.field.**Static**(*value*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *name=None*)

Bases: *kitty.model.low_level.field.BaseField*

A static field does not mutate. It is used for constant parts of the model

__init__(*value*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *name=None*)

Parameters

- **value** (*str*) – default value
- **encoder** (StrEncoder) – encoder for the field (default: ENC_STR_DEFAULT)

- **name** – name of the object (default: None)

Example

```
Static('this will never change')
```

class kitty.model.low_level.field.**String**(*value*, *max_size=None*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=True*, *name=None*)

Bases: kitty.model.low_level.field._LibraryField

Represent a string, the mutation target common string-related vulnerabilities

__init__(*value*, *max_size=None*, *encoder=<kitty.model.low_level.encoder.StrEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – default value
- **max_size** – maximal size of the string before encoding (default: None)
- **encoder** (StrEncoder) – encoder for the field
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

Example

```
String('this is the default value', max_size=5)
```

hash()

lib = None

kitty.model.low_level.field.**gen_power_list**(*val*, *min_power=0*, *max_power=10*)

kitty.model.low_level.mutated_field module Fields to perform mutation fuzzing

In some cases, you might not know the details and structure of the fuzzed protocol (or might just be too lazy) but you do have some examples of valid messages. In these cases, it makes sense to perform mutation fuzzing. The strategy of mutation fuzzing is to take some valid messages and mutate them in various ways. Kitty supports the following strategies described below. The last one, *MutableField*, combines all strategies, with reasonable parameters, together.

Currently all strategies are inspired by this article: http://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html

class kitty.model.low_level.mutated_field.**BitFlip**(*value*, *num_bits=1*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.field.BaseField*

Perform bit-flip mutations of N sequential bits on the value

Example

```
BitFlip('\x01', 3)
Results in: '\xe1', '\x71', '\x39', '\x1d', '\x0f', '\x06'
```

__init__(*value*, *num_bits=1*, *fuzzable=True*, *name=None*)

Parameters

- **value** – value to mutate (str)

- **num_bits** – number of consequtive bits to flip (invert)
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

get_info()

hash()

class kitty.model.low_level.mutated_field.**BitFlips**(*value, bits_range=[1, 2, 3, 4], fuzzable=True, name=None*)

Bases: *kitty.model.low_level.container.OneOf*

Perform bit-flip mutations of (N..) sequential bits on the value

__init__(*value, bits_range=[1, 2, 3, 4], fuzzable=True, name=None*)

Parameters

- **value** (*str*) – value to mutate
- **bits_range** – range of number of consequtive bits to flip (default: range(1, 5))
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

Example

```
BitFlips('\x00', (3, 5))
Results in: '\xe0', '\x70', '\x38', '\x1c', '\x0e', '\x07' - 3 bits flipped each time
            '\xf8', '\x7c', '\x3e', '\x1f' - 5 bits flipped each time
```

class kitty.model.low_level.mutated_field.**BlockDuplicate**(*value, block_size, num_dups=2, fuzzable=True, name=None*)

Bases: *kitty.model.low_level.mutated_field.BlockOperation*

Duplicate a block of bytes from the default value, each mutation moving one byte forward.

__init__(*value, block_size, num_dups=2, fuzzable=True, name=None*)

Parameters

- **value** (*str*) – value to mutate
- **block_size** – number of consequtive bytes to duplicate
- **num_dups** – number of times to duplicate the block (default: 1)
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

hash()

class kitty.model.low_level.mutated_field.**BlockDuplicates**(*value, block_size, num_dups_range=(2, 5, 10, 50, 200), fuzzable=True, name=None*)

Bases: *kitty.model.low_level.container.OneOf*

Perform block duplication with multiple number of duplications

__init__(*value, block_size, num_dups_range=(2, 5, 10, 50, 200), fuzzable=True, name=None*)

class kitty.model.low_level.mutated_field.**BlockOperation**(*value*, *block_size*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.field.BaseField*

Base class for performing block-level mutations

__init__(*value*, *block_size*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – value to mutate
- **block_size** – number of consequtive bytes to operate on
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

hash()

class kitty.model.low_level.mutated_field.**BlockRemove**(*value*, *block_size*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.mutated_field.BlockOperation*

Remove a block of bytes from the default value, each mutation moving one byte forward.

__init__(*value*, *block_size*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – value to mutate
- **block_size** – number of consequtive bytes to remove
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

class kitty.model.low_level.mutated_field.**BlockSet**(*value*, *block_size*, *set_chr*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.mutated_field.BlockOperation*

Set a block of bytes from the default value to a specific value, each mutation moving one byte forward.

__init__(*value*, *block_size*, *set_chr*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – value to mutate
- **block_size** – number of consequtive bytes to duplicate
- **set_chr** – char to set in the blocks
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

class kitty.model.low_level.mutated_field.**ByteFlip**(*value*, *num_bytes=1*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.field.BaseField*

Flip number of sequential bytes in the message, each mutation moving one byte forward.

Example

```
            ByteFlip('\x00\x00\x00\x00', 2)
            # results in:
            '\xff\xff\x00\x00'
            '\x00\xff\xff\x00'
            '\x00\x00\xff\xff'
```

__init__(*value*, *num_bytes=1*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – value to mutate
- **num_bytes** – number of consequtive bytes to flip (invert)
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

get_info()

hash()

class kitty.model.low_level.mutated_field.**ByteFlips**(*value*, *bytes_range=(1, 2, 4)*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.container.OneOf*

Perform byte-flip mutations of (N..) sequential bytes on the value

__init__(*value*, *bytes_range=(1, 2, 4)*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – value to mutate
- **bytes_range** – range of number of consequtive bytes to flip (default: (1, 2, 4))
- **fuzzable** – is field fuzzable (default: True)
- **name** – name of the object (default: None)

Example

```
            ByteFlips('\x00\x00\x00\x00', (2,3))
            Results in:
            '\xff\xff\x00\x00', '\x00\xff\xff\x00', '\x00\x00\xff\xff'  # 2 bytes flipped each ti
            '\xff\xff\xff\x00', '\x00\xff\xff\xff'  # 3 bytes flipped each time
```

class kitty.model.low_level.mutated_field.**MutableField**(*value*, *encoder=<kitty.model.low_level.encoder.ByteAlignedBitsE object>*, *fuzzable=True*, *name=None*)

Bases: *kitty.model.low_level.container.OneOf*

Container to perform mutation fuzzing on a value ByteFlips, BitFlips and block operations

__init__(*value*, *encoder=<kitty.model.low_level.encoder.ByteAlignedBitsEncoder object>*, *fuzzable=True*, *name=None*)

Parameters

- **value** (*str*) – value to mutate
- **encoder** (BitsEncoder) – encoder for the container (default: ENC_BITS_BYTE_ALIGNED)
- **fuzzable** – is fuzzable (default: True)

- **name** – (unique) name of the template (default: None)

kitty.monitors package

The kitty.monitors package provides the basic monitor class BaseMonitor should not be instantiated, only extended. By default, it runs a separate thread and calls _monitor_func in a loop. If a non-threaded monitor is required, one should re-implement multiple parts of the BaseMonitor class.

Submodules

kitty.monitors.base module This module defines BaseMonitor - the base (abstract) monitor class

class kitty.monitors.base.**BaseMonitor**(*name*, *logger=None*)
Bases: *kitty.core.kitty_object.KittyObject*

Base (abstract) monitor class

__init__(*name*, *logger=None*)

Parameters

- **name** – name of the monitor
- **logger** – logger for the monitor (default: None)

get_report()

Returns the monitor's report

post_test()
Called when a test is completed, prepare the report etc.

pre_test(*test_number*)
Called when a test is started

Parameters **test_number** – current test number

setup()
Make sure the monitor is ready for fuzzing

teardown()
cleanup the monitor data and

kitty.remote package

kitty.remote The remote package provides RPC mechanism for kitty Currently, all RPC communication is done over TCP

Submodules

kitty.remote.actor module
class kitty.remote.actor.**RemoteActor**(*host*, *port*)
Bases: *kitty.remote.rpc.RpcClient*

get_report()
need to wrap get_report, since we need to parse the report
class kitty.remote.actor.**RemoteActorServer**(*host*, *port*, *impl*)
Bases: *kitty.remote.rpc.RpcServer*

get_report()

kitty.remote.rpc module RPC implementation, based on jsonrpc https://json-rpc.readthedocs.org/

class kitty.remote.rpc.**RpcClient**(*host*, *port*)
Bases: object

__init__(*host*, *port*)

Parameters **url** – URL of the RPC server

get_unique_msg_id()

Returns a unique message id

class kitty.remote.rpc.**RpcHandler**(*request*, *client_address*, *server*)
Bases: BaseHTTPServer.BaseHTTPRequestHandler

do_POST()
Handle POST requests

error_response(*code*, *msg*)
Send an error response

Parameters

- **code** – error code
- **msg** – error message

send_result(*additional_dict*)
Send a result to the RPC client

Parameters **additional_dict** – the dictionary with the response

valid_response(*result*)
Send a valid response with the result

Parameters **result** – the result of the call

class kitty.remote.rpc.**RpcHttpServer**(*server_address*, *handler*, *impl*, *meta*)
Bases: BaseHTTPServer.HTTPServer

__init__(*server_address*, *handler*, *impl*, *meta*)

Parameters

- **server_address** – address of the server
- **handler** – handler for requests
- **impl** – reference to the implementation object

class kitty.remote.rpc.**RpcServer**(*host*, *port*, *impl*)
Bases: object

__init__(*host*, *port*, *impl*)

Parameters

- **host** – listening address
- **port** – listening port
- **impl** – implementation class

start()
Serving loop

kitty.remote.rpc.**decode_data**(*data*)
Decode data - list, dict, string, bool or int (and nested)

Parameters

- **data** – data to decode
- **encoding** – encoding to use (default: 'hex')

Returns decoded object of the same type

kitty.remote.rpc.**decode_string**(*data*, *encoding='hex'*)
Decode string

Parameters

- **data** – string to decode
- **encoding** – encoding to use (default: 'hex')

Returns decoded string

kitty.remote.rpc.**encode_data**(*data*)
Encode data - list, dict, string, bool or int (and nested)

Parameters

- **data** – data to encode
- **encoding** – encoding to use (default: 'hex')

Returns encoded object of the same type

kitty.remote.rpc.**encode_string**(*data*, *encoding='hex'*)
Encode string

Parameters

- **data** – string to encode
- **encoding** – encoding to use (default: 'hex')

Returns encoded string

kitty.targets package

The kitty.target package provides basic target classes. The target classes should be extended in most cases.

BaseTarget should not be instantiated, and only serves as a common parent for *ClientTarget*, *EmptyTarget* and *ServerTarget*.

ClientTarget should be used when fuzzing a client. In most cases it should not be extended, as the special functionality (triggering the victim) is done by its *ClientController*

EmptyTarget can be used in server-like fuzzing when no communication should be done.

ServerTarget should be used when fuzzing a server. In most cases it should be extended to provide the appropriate communication means with the server.

Submodules

kitty.targets.base module This module defines BaseTarget - the basic target

class kitty.targets.base.**BaseTarget**(*name='BaseTarget'*, *logger=None*)
Bases: *kitty.core.kitty_object.KittyObject*

BaseTarget contains the common logic and behaviour of all target.

__init__(*name='BaseTarget'*, *logger=None*)

add_monitor(*monitor*)
Add a monitor

get_report()

get_session_data()
Session related data dictionary to be used by data model.

Returns dictionary (str, bytes)

post_test(*test_num*)
Called when test is completed, a report should be prepared now

pre_test(*test_num*)
Called when a test is started

set_controller(*controller*)
Set a controller

set_fuzzer(*fuzzer*)

setup()
Make sure the target is ready for fuzzing, including monitors and controllers

teardown()
Clean up the target once all tests are completed

kitty.targets.client module

class kitty.targets.client.**ClientTarget**(*name*, *logger=None*, *mutation_server_timeout=3*)
Bases: *kitty.targets.base.BaseTarget*

This class represents a target when fuzzing a client.

__init__(*name*, *logger=None*, *mutation_server_timeout=3*)

Parameters

- **name** – name of the target
- **logger** – logger for this object (default: None)
- **mutation_server_timeout** – timeout for receiving mutation request from the server stack

set_mutation_server_timeout(*mutation_server_timeout*)
Set timeout for receiving mutation request from the server stack.

Parameters **mutation_server_timeout** – timeout for receiving mutation request from the server stack

set_post_fuzz_delay(*post_fuzz_delay*)
Set how long to wait before moving to the next mutation after each test.

Parameters **post_fuzz_delay** – time to wait (in seconds)

signal_mutated()
Called once a mutation was provided to the server stack.

trigger()
Trigger the target (e.g. the victim application) to start communication with the fuzzer.

kitty.targets.empty module

class kitty.targets.empty.**EmptyTarget**(*name*, *logger=None*)
Bases: *kitty.targets.server.ServerTarget*

Target that does nothing. Weird, but sometimes it is required.

__init__(*name*, *logger=None*)

Parameters

- **name** – name of the target
- **logger** – logger for this object (default: None)

kitty.targets.server module

class kitty.targets.server.**ServerTarget**(*name*, *logger=None*, *expect_response=False*)
Bases: *kitty.targets.base.BaseTarget*

This class represents a target when fuzzing a server. Its main job, beside using the Controller and Monitors, is to send and receive data from/to the target.

__init__(*name*, *logger=None*, *expect_response=False*)

Parameters

- **name** – name of the target
- **logger** – logger for this object (default: None)
- **expect_response** – should wait for response from the victim (default: False)

post_test(*test_num*)
Called after each test

Parameters **test_num** – the test number

pre_test(*test_num*)
Called before each test

Parameters **test_num** – the test number

set_expect_response(*expect_response*)

Parameters **expect_response** – should wait for response from the victim (default: False)

transmit(*payload*)
Transmit single payload, and receive response, if expected. The actual implementation of the send/receive should be in `_send_to_target` and `_receive_from_target`.

Parameters **payload** (*str*) – payload to send

Return type str

Returns the response (if received)

CHAPTER 7

Controllers

The controller is in charge of preparing the victim for the test. It should make sure that the victim is in an appropriate state before the target initiates the transfer session. Sometimes it means doing nothing, other times it means starting or reseting a VM, killing a process or performing a hard reset to the victim hardware. Since the controller is reponsible for the state of the victim, it is expected to perform a basic monitoring as well, and report whether the victim is ready for the next test.

7.1 Core Classes

7.1.1 BaseController

`kitty.controllers.base.BaseController(kitty.core.kitty_object.KittyObject)`

`setup(self)`

Called at the beginning of the fuzzing session, override with victim setup

`teardown(self)`

Called at the end of the fuzzing session, override with victim teardown

`pre_test(self, test_number)`

Called before a test is started Call super if overriden

`post_test(self)`

Called when test is done Call super if overriden

`get_report(self)`

Returns a report about the victim since last call to pre_test

7.1.2 ClientController

`kitty.controllers.client.ClientController(kitty.controllers.base.BaseController)`

ClientController is a controller for victim in client mode

`trigger(self)`

Trigger a data exchange from the tested client

7.1.3 EmptyController

`kitty.controllers.empty.EmptyController(kitty.controllers.client.ClientController)`
EmptyController does nothing, implements both client and server controller API

7.2 Implementation Classes

Implemented controllers for different victim types.

7.2.1 ClientGDBController

`kitty.controllers.client_gdb.ClientGDBController(kitty.controllers.client.ClientController)`

ClientGDBController runs a client target in gdb to allow further monitoring and crash detection.

Requires pygdb

```
__init__(self, name, gdb_path, process_path, process_args, max_run_time,
logger=None)
```

7.2.2 ClientUSBController

`kitty.controllers.client_usb.ClientUSBController(kitty.controllers.client.ClientController)`

ClientUSBController is a controller that triggers USB device connection by switching its Vbus signal. It is done by controlling EL7156 from arduino. The arduino is loaded with firmata, which allows remote control over serial from the PC, using pyduino.

```
__init__(self, name, controller_port, connect_delay, logger=None)
```

- `controller_port`: tty port of the controller
- `connect_delay`: delay between disconnecting and reconnecting the USB, in seconds

7.2.3 ClientProcessController

`kitty.controllers.client_process.ClientProcessController(kitty.controllers.client.ClientCon`

Starts the client with `subprocess.Popen`, collects *stdout* and *stderr*.

```
__init__(self, name, process_path, process_args, logger=None)
```

7.2.4 TcpSystemController

`kitty.controllers.tcp_system.TcpSystemController(kitty.controllers.base.BaseController)`

this controller controls a process on a remote machine by sending tcp commands over the network to a local agent on the remote machine to execute using popen

```
__init__(self, name, logger, proc_name, host, port)
```

- `proc_name`: name of victim process
- `host`: hostname of agent
- `port`: port of agent

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

k

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W