Table of Contents

DEVELOPER MANUAL

IronCurtain Development System

Development system of IronCurtain mainly contains three parts:

1. Core Proxy
2. Admin Panel
3. Database

These three parts can run on independent machines. You need to change only some configuration files mentioned in user manual for this.
Core Proxy and the Admin Panel parts share the same database.

IronCurtain Core Proxy

IronCurtain Core Proxy is developed on Eclipse SDK.
Also, gedit, kate were used.

The proxy listens for incoming connections from client. When a new connection comes, it checks the authentication status of that ip address. If no user is authenticated then the user is directed to the login page.Then the request of the user is processed and rules that matched are applied.

IronCurtain operates using plugins and rules. A plugin defines a general template for a rule and provides mechanisms for the rules' application. A plugin is a python file which contains definition of the plugins actions, along with a variable called Args that is used to generate the user interface for adding and editing the rules. After a plugin is installed into IronCurtain, admin users can add new rules that take advantage of the functionalities provided by the newly added plugin.

The communication with the database is handled by Sql Object. This library provides an abstraction layer to the database. Accessing database is as easy as creating class instances and calling functions.

IronCurtain Admin Panel

IronCurtain Admin Panel is developed using TurboGears framework.It is a web application runs separately from the core proxy. It connects to the core proxy by using sockets. Actions on the admin panel are sent to the core proxy using this channel. Also, SQLObject is used for updates on database.

The admin panel allows uploading of new plugin files. It generates user interfaces for the existing plugins, where an admin can add new rules for that plugin.

IronCurtain Database

The customer has the flexibility of choosing their own database engine for IronCurtain's database. SQLObject library provides easy integration with any existing database engines. The changes specified in user manual in two config files are sufficient for changing your database.

Developing Plug-ins

A plugin is a python files that has specific constructs for IronCurtain.

Firs of all every plugin must import the predefined constants for answers, error handling etc.

```
from constants import *
```

The plugin should define a class, named the same as the plugin's file. The class should include some minimal functions and variables. (Desc, Args, ApplyTime, __init__(), act_on_res(), act_on_req()) Their definitions are as follows.

The plugin should include its description. This description will be shown to in the Admin Panel and it can be defined in a string named "Desc"

```
Desc = "Changes request or respose headers"
```

There are three options a plugin may be invoked. It may be called just after HTTP headers are received(before HTTP body) called "BEFORE_BODY". The other option is to be called after 4KB of HTTP body has been received. This is mostly used for image previewing, altering and blocking without downloading all of the file and indicated by constant "AFTER_4KB". Last option is that the plugin is called after all of the HTTP body is received, that is the constant "AFTER_BODY"

```
ApplyTime = AFTER_BODY
```

If a plugin needs fields other than the defaults provided by IronCurtain (name, description, applied users/groups, applied URLs, not applied URLs, applied Tags) it should define a list of tuples named "Args". Each tuple defines a new field to be generated on the user interface. The first part of a tuple is the internal name of the field. Second part of the tuple is the text displayed to the user in the user interface. Third part of the tuple defines the type of the input element; it can be "string" for a plain text box and it can be "enum" for a combo box. If it is a combo box, there should be a fourth part in the tuple, that should be another tuple listing the options in the combo box. If there is no need for extra fields, the Args list should be defined and left empty. An example can be seen below:

```
Args = [("Action", "Action", "enum", ("Replace", "Delete")),
       ("HeaderName", "Header Name", "string"),
       ("HeaderValue", "Header Value", "string"),
       ("ReqRes", "Request/Response", "enum",
                  ("Request", "Response", "Both") )
       ]
```

Access to the fields defined in "Args" is done by the "params" parameter of `__init__(self, params)` Every field can be accessed like `params["HeaderName"]`

When rules are created using this plugin, the user will input data into the fields defined in the "Args". For correct operation, the input should be validated. Input validation is achieved using `__init__()` function and exceptions. The plugin should check the inputs for validity and throw a `ParamError` exception, and also should provide a string to be displayed to user. An example;

```
try:
    parsedImgWidth = int(params["ImgWidth"])
       if (parsedImgWidth <= 0):
           raise ParamError, PARAM_CONTENT_ERROR +
                  "Image Width should be positive"
except ValueError:
    raise ParamError, PARAM_TYPE_MISMATCH +
           "Image Width should be integer"
```

When a request or response occurs, and it matches all conditions of a rule (user, group, appliedURL, applied tag) a corresponding function is called. act_on_req for requests and act_on_res for responses. The parameters to these functions contain whole HTTPRequest or HTTPResponse objects correspondingly. The HTTPResponse may be incomplete, depending on the "ApplyTime" parameter.

```
def act_on_req(self, cReq):
def act_on_res(self, cRes):
```

These functions should return one of these results, defined in the imported constants module;

`(ACTION_BLOCK, message)` : This tuple is returned to block access to site completely. The "message" is displayed to the user and is meaningful only in this case.

`(ACTION_FORWARD, message)` : This tuple is returned to indicate the content is OK for this plugin. T

`(ACTION_ANSWER, message)` : This tuple is returned to indicate the content is changed by this plugin.

Other than these functions, a plugin developer can define arbitrary number of helper functions and classes.

Folder and File System

Auxiliary files, plugins and scripts have their own directories. Main structure is as following, and the detailed explanations follow;

IronCurtain/
IC-Plugins/
IC-Admin-Panel/

IronCurtain/ -- Core proxy resides in this directory
fetch-sm-cvs.sh --script that downloads spider monkey java script engine's sources

IronCurtain/data --this directory contains auxiliary files for plugins

ads.txt -- Regular expressions for ad sites
popup.txt -- Regular expressions for pop-up links
images/ -- Contains images used by plugins

IronCurtain/util/ -- Contains scripts

blacklister.sh -- a script to add black sites to database
blacklister.py -- helper for blacklister
rootcreate.py -- script to create an initial root user for the Admin Panel

IronCurtain/src/ -- Source files

Auth.py --manages login process of the users
config.py --contains setting for the core proxy
constants.py --commonly used constants in Core Proxy and Plugins
core.py --handles two way traffic in the proxy, and applies the previously defined rules to this traffic. Each request generates a new thread for serving.
HttpReply.py --contains the class to represent and operate on HTTP Replies.
HttpRequest.py --contains the class to represent and operate on HTTP Requests.
ICLog.py --contains the class to write the logs into the database. It works in its seperate thread.
JSFilter.py --executes the JavaScript code found in the webpages. This is used for catching dynamically generated content on the browser.
model.py --contains the schema of the database to be used in the SqlObject Library.
PlugCore.py --main plugin engine. handles adding and removing plugins and rules.
ProxyListener.py --uses asynchat library of python to communicate with the admin panel. It executes the commands sent by the admin panel. (i.e. adding or updating a rule)
start-ironcurtain.py -- Starts the IronCurtain Proxy Module
plugins/ -- Plugin sources files. Initially it is empty. Newly added plugins will be stored here.
js/ -- Javascript engine files

IC-Plugins/ -- Bundled IronCurtain Plugins
BandwidthLimiter.py --it enables to set quotas to each user. It blocks the user access if quota is exceeded.
ChangeHeader.py -- inspects Http Headers and acts according to the defined rules (i.e. changing/removing a header)
ContentBlocker.py --inspects pages' content and classifies its content according to topics. It is the main classification plugin that we are using. It is first trained with more than 400 sites. It has the following categories (developer may add new categories): Arts & Humanities, Business & Economy, Computers & Internet, Education, Entertainment, Government, Health, News & Media, Recreation & Sports, Science, Social Science, Society & Culture.
GifDeanimate.py -- takes the first frames of the animated gif images and displays only the first frame. Effectively it disables the animation
ImageBlocker -- blocks all the images in webpages. Replaces them with transparent gifs of the same size, thus it keeps the layout of the webpage.

ImageSize.py -- Blocks image files according to the user defined width and height of the image and also according to the format of the image

ModifyContent.py --modifies the content of the webpage. It may directly block the page,or remove or replace keywords. It may affect on user defined tags.

ModifyJavascript.py --modifies the dynamic content generated by javascript on the browser.After using Java Script Engine ,spider monkey., the plugin acts on the final content that the user will see.

UrlBlocker.py --blocks the access to a site with given URL completely.

IC-Admin-Panel/ -- the main directory of the Admin Panel

start-icadminpanel.py -- Starts the IronCurtain Web Admin Panel Module

icadminpanel/ -- Contains the admin panel source files

icconfig.py --has settings for connection to the core proxy

controllers.py -- creates dynamically generated web pages for admin panel. Each page in the panel has corresponding function in this file.

model.py --contains the schema of the database to be used in the SqlObject Library.

ProxyConn.py --opens connection to the core proxy and sends the commands generated in the admin panel

config/ -- Contains the config files

static/ -- appearance and function of admin panel. Css files for the style of the Html pages. Images used in the admin panel. Javascript files that provide client side functionality to the admin panel.

templates/ -- Includes Turbogears's kids template files for GUI. This allows easy creation of webpages from python code.