# Zorp Getting Started

**Balázs Scheidler**

**Attila Szalay**

**Zorp Getting Started**

by Balázs Scheidler and Attila Szalay

# Table of Contents

# List of Examples

# Chapter 1. Zorp achitecture

This chapter contains an overview of current firewall technologies, and gives you a basic insight into Zorp architecture.

## 1.1. Current technologies

In this section we try to give you an overview of current firewall technologies and trends.

- Bastion host

  Bastion host is a protected server or workstation, having two connections simoultaneously: 1) with a protected network, and 2) the internet. There is no direct way between the networks. If a client on the protected network wants to access a service on the Internet, he has to enter the bastion host. Bastion hosts are built in case of special needs of security.

  Today they are not at all wide-spread, because their installation, usage and preventive maintenance are difficult, demanding much expertise.

- Packet filtering firewall

  Packet filtering firewalls - as their name suggests - filters network traffic on the packet level. This means that the decision about the packet's further processing is made based on the information available in the packet header (IP, UDP and TCP headers).

  This information is not enough in environments with higher-than-low risk, because the packet contents are not checked.

- Stateful packet filtering

  Stateful packet filtering was designed to eliminate the disadvantages of simple packet filters. These firewalls try to track associated packets (TCP connections for example) and make decisions on stream contents in addition to packet headers.

The problem with stateful packet inspection is that interpreting a TCP stream in the same way as a client does cannot be done, since some clients interpret some parts of the protocol differently.

- Proxy firewall

  Proxy firewalls don't forward packets. They accept a given connection, and connect to the server end on their own. They read protocol requests, interpret them and in case a given request is allowable they send it on to the server. The real difference between proxy firewalls and packet filtering is that the two connections (client->proxy, proxy->server) or completely independent of each other, and that greater detail of the stream contents can be analyzed.

  It is much easier to change stream contents, while an SPF can only insert or remove some bytes to/from the stream, a proxy firewall can easily change it completely. (For instance convert POP3 requests to imap and vice versa)

- Modular proxy firewall

  A proxy firewall becomes modular when proxies can be connected in any way. This means that if we have a main protocol having some subprotocol (think of an SSH connection with a forwarded POP3 stream, or a HTTP protocol embedded in SSL, or PPP within telnet) we can attach a proxy to the embedded part.

  Making proxies modular allows many protocols to correctly analyse, which weren't possible before.

  For example, many corporate firewalls disallow using SSH, because of its portforwarding feature, using an SSH tunnel one can easily subvert corporate policies and open up the whole protected network to an outsider.

  This is a disadvantage, but completely disallowing SSH has some serious drawbacks too. We lose the on-wire encryption provided by SSH, which is a must for protocols sending passwords in the clear.

  Zorp was built from the ground up to be modular, proxies can be stacked within each other in case the parent protocol permits some kind of embedded protocol/data. The ssh proxy to be implemented will allow a POP3 proxy to be attached to the tunneled TCP connection. This way you can control which features of ssh is allowed, and although the possibility to create a tunnel through the firewall is closed, the on-wire encryption provided by SSH is not.

## 1.2. Base concepts in Zorp

Zorp is a component-based, object-oriented, event-driven and modular proxy firewall suite, which makes it possible to finetune proxy decisions (with its built in script language), to fully analyze complex protocols (like SSH with several forwarded TCP connections), and to utilize outband authentication techniques (unlike common practices where proxy authentication had to be hacked into the protocol).

Zorp is basically made up the following four cooperating building blocks:

- decision logic

  Implemented in Python and is primarily responsible to say A or B for questions. These questions are asked by the low level proxy modules to know how to treat their peers.

  It is implemented as hooks an administrator can override, or whose result is defined through parameters the administrator can set, or change.

- connecting proxy modules

  Since Zorp is modular, modules must be connected in a way meaningful to the current application.

- application level gateways, proxies

  These are implemented in native programming language. They work closely with the decision logic, so the details of the protocol can be controlled by the administrator.

- authentication modules

  Zorp was designed to support user authentication from the ground up. Current practices of user authentication on firewalls require a special extension to the protocol, which is used to pass authentication credentials. Instead of this Zorp supports outband user authentication, where a channel other than the current protocol channel is used to pass this information.

### 1.2.1. Component based

Component based development has several advantages over traditional program development: reusability, smaller code size, higher level languages, language independence etc.

Zorp is also using components, but it doesn't depend on large component systems like OMG CORBA or Microsoft DCOM. Using those on a firewall would be questionable, because of security reasons.

A component in zorp is an interface, a set of functions and variables, which hides details in the implementation. Changing the implementation doesn't effect interoperability with other interfaces.

The four most important components in Zorp are:

- ZorpListen

  Implements a listener which accepts connections, and calls the given callback if a connection is accepted. A connection is represented by a ZorpStream interface.

- ZorpConnect

  Connects to the given address, and calls the given callback if the connection is established. The result connection can also be represented by a ZorpStream interface.

- ZorpStream

  This is a full-duplex stream with read and write operations, and are used by Proxies to transmit data.

- ZorpProxy

  A ZorpProxy receives two ZorpStreams representing the client and the server side. ZorpProxy is responsible to connect these with an appropriate protocol.

  Proxy modules (application gateways) implement this interface, and are loaded dynamically.

This spartaic interface provided by Zorp is enriched with utility classes implemented in Python. These utility classes accept and make connections, and start the appropriate proxy module for a connection. Since the core of zorp doesn't

even know about things like access control, these are also implemented in Python. (deciding if a given connection is allowed or not)

### 1.2.2. Object oriented

Although building a system from components is also an OOP approach, the real power is the language where decisions can be specified. The language - Python - is an easy to learn, clean, and powerful object oriented language.

The administrator can either define his own reusable set of classes, or can use the ones provided by us.

### 1.2.3. Event driven

Event driven in the context of zorp doesn't exactly mean the thing you are used to in connection with GUI programming. Event driven means that proxies send events to the decision layer, and the result controls the behaviour of the proxy.

For example the FTP proxy module sends an event whenever the transmission of a file is requested. The administrator then can hook into the processing of this event, and decide if the given file transfer is allowed, or not. (for instance by checking the current directory).

We are not however limited to yes/no answers. With the following means of communication we can create quite complex control mechanisms:

1. Events

   Events are sent by the underlying Proxy to the event handler class implemented in Python. It may have one or more parameters, and returns a value modifying proxy behaviour. Handling an event means implementing an object method in Python.

2. Attributes

   Attributes represent the internal state of the proxy which it decides to make available to the policy layer.

3. Callbacks

   Callbacks are similar to events with the difference that they are called by the policy layer, and implemented by the proxy (contrary the event is called by the proxy and implemented in the policy)

## 1.2.4. Modular

Each proxy module can be an up-level proxy, where it is directly connected to the server and the client, or can be attached as a subprotocol proxy to a parent proxy. This is the case when a multiplexer protocol (like SSH), must be fully analyzed. This is called stacking a proxy into another.

# Chapter 2. Parts of Zorp

In this chapter we'll have a quick overview on the parts a typical Zorp installation has

## 2.1. Configuration

The configuration and firewall policy is stored in a Python module found in `/etc/zorp/policy.py` by default.

## 2.2. Python utility classes

As we wrote earlier the spartaic interface provided by the Zorp core is covered by utility classes implemented in Python. These classes are stored in the Zorp python package, found in the `/usr/share/zorp/pylib` directory. You have to make sure that the PYTHONPATH is correctly set up to point to this directory.

## 2.3. Proxy modules

Shared objects containing proxy modules can be found in the `/usr/lib/zorp` directory.

## 2.4. Zorp binary

The zorp binary itself can also be found in the `/usr/lib/zorp` directory, but prior to running it you have to set some environment variables (done by the zorpctl script)

## 2.5. Instances description

You can run several Zorp instances on the same firewall. their startup parameters are stored in the `/etc/zorp/instances.conf` file.

# Chapter 3. Creating policies

The configuration and decision policy used by Zorp is made up by standard Python statements, which use the services provided by the Zorp core and utility classes. This chapter contains a step by step introduction to the policy structure with a small overview of the Python language.

## 3.1. Reading the policy

The policy file is read in two steps. First the file `policy.boot` file is read, and if it's processed successfully, the file provided by the administrator `/etc/zorp/policy.py` is processed.

`policy.boot` file is used internally, you normally won't need to touch it. It is used to initialize the Zorp - policy interface.

`/etc/zorp/policy.py` is the local firewall policy, and contains valid Python statements describing the configuration of your firewall.

## 3.2. Python overview

Python is an object oriented, interpreted language known from its clear syntax and consistent design. This section is not meant as an exhaustive Python reference, just as a first glance introduction you may need to create policies. If you want to know more about Python, consult the Python documentation.

### 3.2.1. Modules and Packages

Chunks of python code (variables, functions and classes) can be organized into modules. To use something from a module, you'll need to import the module with the import keyword.

**Example 3-1. Sample import statement**

```
import HttpProxy
```

The import statement above will create a reference to the HttpProxy module. You can access parts of a module with the dot operator: `HttpProxy.HTTP_POLICY`.

Modules can be grouped into a package, which in itself is a module of modules. So accessing a submodule in a package can be done `Zorp.HttpProxy` provided you have a package named Zorp, and it has a subpackage named HttpProxy.

Zorp classes are provided in a package named Zorp. It has several submodules, for details consult the docstrings in Python files in `/usr/share/zorp/pylib/Zorp/*.py`

To avoid having to use full references to variables, you can use named imports. The syntax for named imports is demonstrated in the following example.

**Example 3-2. Named imports**

```
from Zorp.HttpProxy import HttpProxy
```

This way the identifier HttpProxy will be imported to the local namespace and instead of writing Zorp.HttpProxy.HttpProxy, you could simply write HttpProxy.

We usually use the second form when importing identifiers. It has the benefits that we can exactly see what parts of the imported module are used.

## 3.2.2. Statements

In python, statements are not terminated with a semicolon. Line termination ends a statement, though this can be changed by using a '\' at the end of the line.

Compound statements like if and while use indentation to mark the nested commands, like in the following example:

**Example 3-3. Nested commands**

```
if self.request_url == "http://www.balabit.hu/":
    return Z_ACCEPT
```

```
return Z_REJECT
```

The beginning of the block is marked by the colon at the end of the first line, the block is assumed to be ended when the indentation steps back one level.

## 3.2.3. Variables

All variables in Python are dynamically typed, which means that the type of a variable is determined at runtime when a value is assigned to that variable.

Variables doesn't have to be explicitly declared, though reading undefined variables result in an exception.

## 3.2.4. Functions

Functions can be defined using the `def` keyword.

**Example 3-4. Sample function declaration**

```
def filterURL(self, method, url, version):
    return Z_ACCEPT
```

During the execution of the filterURL function, the local variables self, method, url and version will be defined.

Calling the function above can have two formats:

- `filterURL(self, 'GET', 'http://www.balabit.hu/', 'http/1.1')`

- ```
  filterURL(self, url='http://www.balabit.hu/', \
            version='http/1.1', method='GET')
  ```

The two invocations above will result the same arguments to be passed to the filterURL function.
  ```

### 3.2.5. Classes

Classes in Python are similar to classes in other languages. They contain attributes (state) and functions (methods) to manipulate the state information.

**Example 3-5. A sample class in Python**

```
class MyHttp(HttpProxy):

    def config(self):
        HttpProxy.config(self)
        self.transparent_mode = TRUE
```

As you can see the self parameter is explicit, unlike C++ where the ’this’ pointer is implicitly passed to member functions.

Base classes can be specified in parentheses just after the name of the class. Instances of a class can be created by "calling" the class with constructor parameters passed as arguments. Constructor of a class is named __init__, destructor is called __del__.

## 3.3. Overview of Zorp classes

Writing policies usually means passing parameters to predefined Zorp objects, so you’ll need a little understanding what is what in Zorp.

- Listener

  Listener is responsible to listen on a network interface and start a service if a connection is accepted.

- Service

  Service is a class encapsulating a firewall service offered to clients. A service consists of a unique name (used in log messages and access control), a proxy class to instantiate when the service is started, a chainer responsible for

connecting to the server, and an authentication mechanism used to authenticate a user. (currently disabled)

- Chainer

  Chainer is responsible for determining the server address to connect to. There are several different Chainers defined each implementing a different method for defining the server address. TransparentChainer connects to the original destination (primarily used on transparent proxies), DirectedChainer connects to a predefined and fixed address (primarily used to direct traffic to an externally unaddressable host). For more information on chainers consult the `Chainer.py` source file.

- Proxy class

  Each proxy in Zorp defines a class in Python from which you can derive your customized classes, these are referred to as proxy classes. By deriving your class from a particular low level class, it determines the protocol that is usable on channels handled by your class.

- Zone

  Zone and its derivatives are the basis in connection level access control in Zorp.

## 3.4. Creating an example policy

In this section we define a sample policy to be running at a fictitious company. Our fictitious company has the following network infrastructure:

- intranet with non routable IP addresses (192.168.1.0/24)
- DMZ with non routable IP address providing HTTP and FTP service to outside hosts. (192.168.0.0/24)
- Internet access with leased line, external IP address 10.9.8.7 (I know it’s also nonroutable, it’s just for the sake of the example).

The leased lines will be connected to the external interface of our firewall, our intranet and DMZ will be connected to our firewall with dedicated interfaces.

FIXME: figure

## 3.4.1. Basis of access control: Zones

You'll need to describe your firewall's environment by defining zones that surrounds your firewall.

**Example 3-6. Zone definition**

```
InetZone('intranet', '192.168.1.0/24',
         inbound_services=["*"],
         outbound_services=["*"])

InetZone('DMZ', '192.168.0.0/24',
         inbound_services=["*"],
         outbound_services=["*"])

InetZone('internet', '0.0.0.0/0',
         inbound_services=["*"],
         outbound_services=["*"])
```

In the example above, we defined 3 zones: intranet with address range 192.168.1.0/24, DMZ with address range 192.168.0.0/24, and internet 0.0.0.0/0.

For now we allow all inbound, and outbound services using the asterisks (an asterisk matches all services), if you want to allow specific services you need to use their full name.

## 3.4.2. Providing an init() function

`init()` is called by the Zorp core after the policy file has been parsed. It is the responsibility of this function to set up services and start listeners.

This init function receives a single argument *name* containing the name of this instance (can be set with the --as command line argument, or using zorpctl)

If you don’t provide an init function yourself the default one is used, which tries to call the function named as the instance name. So if you have an instance named intra_http and don’t provide an init() function, the function intra_http() is called and is expected to correctly initialize the instance. If this function is not found an exception is raised.

If you run several instances using the same policy file, it is suggested that you use the init function provided by Zorp.

## 3.4.3. Defining a service

A service is something Zorp provides to clients. When a connection is accepted, a service instance is started.

**Example 3-7. Creating a service**

```
def init()
        Service("intra_http",
                InbandChainer(),
                HttpProxy)
```

The service definition above creates a service whose ID is intra_http, uses the InbandChainer() and launches a HttpProxy for established connections. InbandChainer uses the protocol logic for determining destination address.

## 3.4.4. Setting up a Listener

A listener is responsible for listening on the given address, and starting a service if a connection is accepted.

**Example 3-8. Setting up a Listener**

```
Listener(SockAddrInet(’192.168.1.1’, 50080), "intra_http")
```

The example above sets up a Listener to listen on 192.168.1.1:50080, and to start our intra_http_service when a connection is accepted.

## 3.4.5. Customizing proxies

You can extend functionality of a given proxy by creating a custom proxy class derived from the original proxy class.

**Example 3-9. Customizing proxy classes**

```
class MyHttp(HttpProxy):

    def config(self):
        HttpProxy.config(self)
        self.transparent_mode = FALSE
        self.request["PUT"] = (HTTP_PASS,)
```

The example above creates a new proxy class named MyHttp, derived from HttpProxy and overriding its config() method.

# Appendix A. Example policy

```
from Zorp.Zorp import *
from Zorp import Zorp
from Zorp.Zone import InetZone
from Zorp.Service import Service
from Zorp.SockAddr import SockAddrInet
from Zorp.Chainer import TransparentChainer, DirectedChainer \
                         InbandChainer, FailoverChainer
from Zorp.Plug import PlugProxy
from Zorp import Http
from Zorp.Http import HttpProxy
from Zorp.Ftp import FtpProxyAllow
from Zorp.Listener import Listener

Zorp.firewall_name = 'fw@fiktiv'

InetZone('intranet', '192.168.1.0/24',
        outbound_services = ["BIHttp", "BIFtp", "BIPop",
                             "BDHttp", "BDFtp", "BDSsh"],
        inbound_services=[]

InetZone('DMZ', '192.168.0.0/24',
        outbound_services = ["DIHttp", "DIFtp"],
        inbound_services = ["BDHttp", "BDFtp", "BDSsh",
                            "IDHttp", "IDFtp"]),

InetZone('internet', '0.0.0.0/0',
        outbound_services = ["IDHttp", "IDFtp"],
        inbound_services  = ["BIHttp", "BIFtp", "BIPop",
                             "DIHttp", "DIFtp"])


class BIHttp(Http.HttpProxy):

    def config(self):
        HttpProxy.config(self)
        self.transparent_mode = 0
        self.request["POST"] = (Http.HTTP_DROP)
               self.request_headers["User-Agent"] = \
```

```
                                [Http.HTTP_CHANGE_VALUE, "Lynx/2.8.3rel.1"]


        class BIFtp(FtpProxyAllow):

            def config(self):
                FtpProxy.config(self)
                self.fw_server_data.ip_s = "10.9.8.7"
                self.fw_client_data.ip_s = "192.168.1.1"

        class BIPop(PlugProxy):

            def config(self):
                pass

        class BDHttp(HttpProxy):

            def config(self):
                HttpProxy.config(self)
                self.transparent_mode = 1

        class BDFtp(FtpProxyAllow):

            def config(self):
                FtpProxy.config(self)
                self.fw_server_data.ip_s = "192.168.0.1"
                self.fw_client_data.ip_s = "192.168.1.1"

        class BDSsh(PlugProxy):

            def config(self):
                pass

        class IDHttp(HttpProxy):

            def config(self):
                HttpProxy.config(self)
                self.transparent_mode = 0

        class IDFtp(FtpProxyAllow):
```

```
    def config(self):
        FtpProxy.config(self)
        self.fw_server_data.ip_s = "192.168.0.1"
        self.fw_client_data.ip_s = "10.9.8.7"


    def user(self, dir, uname):
        if uname == "ftp"
            return Z_ACCEPT
        elsif uname == "anonymous"
            return Z_ACCEPT
        return Z_REJECT

class DIHttp(PlugProxy):

    def config(self):
        pass

class DIFtp(FtpProxyAllow):

    def config(self):
        FtpProxy.config(self)
        self.fw_client_data.ip_s = "192.168.0.1"
        self.fw_server_data.ip_s = "10.9.8.7"

def init(name):

    BIHttp_service = \
        Service("BIHttp",
                InbandChainer(),
                BIHttp)

    BIFtp_service = \
        Service("BIFtp",
                TransparentChainer(),
                BIFtp)

    BIPop_service = \
                Service("BIPop",
                TransparentChainer(),
                BIPop)
```

```
    BDHttp_service = \
                Service("BDHttp",
                TransparentChainer(),
                BDHttp)

    BDFtp_service = \
                Service("BDFtp",
                TransparentChainer(),
                BDFtp)

    BDSsh_service = \
                Service("BDSsh",
                TransparentChainer(),
                BDSsh)

    IDHttp_service = \
                Service("IDHttp",
                DirectedChai-
ner(SockAddrInet("192.168.0.2", 80),
                IDHttp)

    IDFtp_service = \
                Service("IDFtp",
                DirectedChai-
ner(SockAddrInet("192.168.0.3", 21)),
                IDFtp)

    DIHttp_service = \
                Service("DIHttp",
                TransparentChainer(),
                DIHttp)

    DIFtp_service = \
                Service("DIFtp",
                TransparentChainer(),
                DIFtp)

    Liste-
ner(SockAddrInet("192.168.1.1", 3128), BIHttp_service)
    Listener(SockAddrInet("192.168.1.1", 2021), BIFtp_service)
```

```
    Listener(SockAddrInet("192.168.1.1", 2110), BIPop_service)
    Liste-
ner(SockAddrInet("192.168.1.1", 3080), BDHttp_service)
    Listener(SockAddrInet("192.168.1.1", 3021), BDFtp_service)
    Listener(SockAddrInet("192.168.1.1", 3022), BDSsh_service)
    Listener(SockAddrInet("10.9.8.7", 80), IDHttp_service)
    Listener(SockAddrInet("10.9.8.7", 21), IDFtp_service)
    Listener(SockAddrInet("192.168.0.1", 80), DIHttp_service)
    Listener(SockAddrInet("192.168.0.1", 21), DIFtp_service)
```