
Zorp GPL Tutorial

Release 0.1

Szilárd, Pfeiffer

Nov 08, 2017

Contents

1	Introduction	1
1.1	What Is Zorp?	1
1.2	What Good Is Zorp?	4
2	Getting Started	7
2.1	Basic Concepts	7
2.2	Minimal Configuration	14
3	Simple Use Cases	19
3.1	Protocol Enforcement	19
3.2	Forward Proxy	20
3.3	Reverse Proxy	21
3.4	Access Control	22
4	Get Involved	25
4.1	Troubleshooting	25

1.1 What Is Zorp?

Zorp GPL is a next generation, open source proxy firewall with deep protocol analysis. It allows you to inspect, control, and modify traffic on the application layer of the ISO/OSI model. Decisions can be made based on data extracted from the application-level traffic (for example, HTTP) and applied to a certain traffic type such as users or client machines. It ensures that the traffic complies with the particular protocol standards, and allows you to perform specific actions with the traffic.

Why choose it?

- Free license and active community support
- Network traffic analysis in 7 protocols
- Encrypted channel control
- Content filtering and optional modification
- Modular, highly flexible configuration
- The only answer to many unique problems
- Established project with a 10-year history

1.1.1 Features

Access control Access control in *Zorp GPL* has a lot more possibilities than average firewalls. It is based on zones instead of hosts or IP ranges and besides “who” and “what”, it can also limit “how”. For example, clients arriving from one zone can only read a given FTP server, whereas others have write privileges.

Information leakage prevention Information leakage prevention helps to keep sensitive information inside your network. For example, HTTP data flow could include internal IP addresses, the URL of a previously visited website (referrer), or browser and operating system information (agent). *Zorp GPL* is able to remove or change this information.

Content filtering Content filtering is done by using external applications, like virus scanners, spam filters and URL checkers. Connections can be accepted, rejected or just simply logged. Suspicious content can be quarantined. *Zorp GPL* can integrate with all popular antivirus engines, such as NOD32 or AMaVIS.

Supported protocols:

- wildly used procols: *HTTP, FTP, SMTP, POP3*
- rarely used: *Finger, Whois, Telnet*
- secure: *HTTPS, FTPS, POP3S, SMTPS*

Audit Audit of all events is possible, even requests and responses of a protocol, as proxies work at the application level. This can prove not only what happened, but also what did not, for example an old version of a file was deleted, but never uploaded again.

Interoperability Interoperability helps in a world where not all protocol implementation is created equal. *Zorp GPL* is able to hide protocol features, like compression from HTTP, translate between different encryption standards, and other changes to make clients and servers interoperate more easily.

Flexibility Flexibility is a key feature of *Zorp GPL*. It is easily extendable by additional modules and customizable to solve specific security problems.

Linux support *Zorp GPL* administrators can compile and run the product on several Linux-based operating systems. Besides that, pre-compiled binaries are readily available on various Linux distributions, which greatly simplifies its installation on these platforms. Currently binary repositories are available for the following distributions:

- *Debian*: squeeze, unstable; (i386, amd64)
- *Ubuntu*: from 10.04 (i386, amd64)

1.1.2 Brief Explanation

Briefly *Zorp* is an open source proxy firewall with deep protocol analysis. It sounds very sophisticated at first, however, the explanation below will make it easy to understand.

Protocol analysis

Resulting from their functionality firewalls can analyze the network traffic to a certain extent, since without it, it would not be possible for the administrators to control the traffic. This is not different with *Zorp*. The difference between the firewall applications result from the depth of the analysis. For instance when administrators use *Netfilter* traffic can only be controlled up until layer 4 (traffic) of the *ISO/OSI* model. In contrast to that *Zorp* allows analyzation of even the topmost (application) layer, and can make decisions based on data originating from that layer. Decisions can be applied a certain traffic type, for example full access can be set to an *FTP* server for a group of users, or only a subset of commands can be granted to implement a read-only access.

Proxy firewall

Almost anything that comes to your mind can be applied on *Zorp*. First of all the fact that a *proxy* server makes independent connections with the participants of the network communication and relays messages between them separating the clients and the servers from each other. In this regard *Zorp* is better than its competitors as the analysis can take place at the application level, either firewall is used as a *forward* or a *reverse proxy*. To perform that *Zorp* implements application level protocol analyzers. These analyzers, called *proxy* in *Zorp* terminology, are written in *C*, extendable and configurable in *Python*. Nine of twenty five *proxies* of the commercial version of *Zorp* are available in the open source edition.

Modularity

One of the key features of the *Zorp* is customization. It would not be possible without the modular structure of the software. During everyday use it does not require any extra effort to get the benefits of the application level analysis of the network protocols, if we do not have any special requirements. To keep the application level traffic under control we do not have to care about neither the lower layers of the protocol, nor the details of the application level. We only have to concentrate on our goal (for example replacing the value of a specific *HTTP* header), everything else is done by the proxy. If the proxy to our favourite protocol is not given, *Zorp* can handle the connection in lower layers and we have the possibility to perform application level analysis manually.

Transport layer security is an independent subsystem in *Zorp* as far as it possible, so the *SSL/TLS* parameters can be set independently from the applied application level protocol (for example *HTTP*, *SMTP*, ...). Consequently each proxy can work within an *SSL* connection, including the case when we perform the protocol analysis. *Zorp* is a proxy firewall, neither more nor less, but can be adapted to tasks other than protocol analysis, such as virus scanning or spam filtering by integrating it with external applications.

1.1.3 Community

Please join our increasing *Zorp GPL* community by subscribing to one or more of the following forums.

- [FaceBook](#)
- [Google+](#)
- [LinkedIn](#)
- [Twitter](#)

1.1.4 Downloads

Sources

- [Zorp GPL](#), the firewall itself
- [kZorp](#), kernel module
- [libzorpll](#), low-level networking library

Binaries

- *Zorp GPL* packages from [Debian](#) and [Ubuntu](#) distribution
- *Zorp GPL* and related packages for Debian and Ubuntu distribution on [MadHouse Project](#)

1.1.5 Support

Mailing lists

- Subscribe to the lists directly in [English](#) or [Hungarian](#)
- List archives are available at the above URLs

Documentation

- [Zorp GPL Tutorial on ReadTheDocs](#)
- [Configuration examples on GitHub](#)

Evaluate

There is a set of [virtual machines](#) to test *Zorp GPL*.

1.1.6 License

Zorp is not only an open source product, but also a free software as it is licensed under [GPL](#). The reason of the two licenses is the fact that *Zorp* is released in two parts and there is also a kernel module.

- *Zorp GPL* is licensed under [GPL 2.0](#)
- *libzorppl* is licensed under [GPL 2.0](#)
- *kZorp* is licensed under [GPL 2.0](#)

It must be noted that the *Zorp* is [dual-licensed](#) by the main developer [BalaSys IT Security](#), where *Zorp/Zorp GPL* is the open source version and *Zorp Professional* is the proprietary one with some extra features and proxies.

1.2 What Good Is Zorp?

A marketing specialist would claim that it is “good for everything”. Not being one of them, we would rather say that *Zorp* is not the philosopher’s stone, however, it can solve almost any issue that can be expected from a deep protocol analyzer proxy firewall. The most important cases are the following:

1.2.1 Access control

Access control is a basic functionality of proxy firewalls, but *Zorp* has an extra feature compared with other firewall suites. Access to the services can be controlled by the attributes of lower layers of the *ISO/OSI* model, like IP addresses or ports, but in case of *Zorp* there is a possibility to define sets of IP subnetworks, called *zones*. *Zones* are IP subnetwork groups that administratively belong together (for example all those who are permitted to access *FTP* servers for upload) and can be linked to a tree hierarchy. Access control rights are inherited between the levels of the *Zone* tree. A top-level access (for example a right to download from *FTP* servers) is in effect in the lower levels as long as it is not blocked. In this way an administrative hierarchy can be created that is independent from the network topology and the location of the devices, while reflecting only the network policy.

When an access control policy is being created, we first have to find answers to the “who”, “what” and “how” - questions. Resources should be accessible only for a specific group of users under the defined conditions. It may mean that each request and response must be recorded to the system log when a given server is accessed. Some features of the protocol (for example: *STARTTLS* in case of *SMTP*) causing incompatibility between the client and the server may have to be filtered out. Some items of the protocol (for example *PUT* in case of *FTP*) may be rejected. Some protocol items (for example *user-agent* in case of *HTTP*) may be changed to avoid information leak. Secure connection may be decrypted on one side and encrypted again on the other side. The following sections will describe this in detail.

1.2.2 Information Leak Prevention

Several protocols leak information about the running softwares, the networking options of the clients, which is usually not filtered or not blocked by the firewalls, because they are absolutely compliant with the related standards. An example of this is the `user-agent` header in the *HTTP* protocol, which contains the name and the version of the web browser connected to the server. In this case an information about the software being run on the client machine is received by the visited web server without the knowledge or the permission of the user.

The proxy settings of the web browser, the *IP* address of the machine, the *URL* of the previously visited web page (referrer of the currently visited one) are leaked in the same way. Similar methods exist in case of several protocols, besides *HTTP*. System administrators have to be aware of these type of information leaks and have the means to forbid them. *Zorp* is an easy-to-use and flexible tool for that.

1.2.3 Interoperability

Continuing the example above, not only forbidding of complete protocol items is possible, but also the modification of their values. It can solve the problem of the interoperability for example when a web server constraints the type or the version of the connecting browser despite of the fact that it has no good or valuable reason. Such a situation can be solved easily by changing the value of the `user-agent` header in the request sent by the browser to a value which is acceptable to the server.

The lack of encryption support may cause interoperability mainly in case of old-fashioned software especially when the traffic should pass through an untrusted network. There are several solutions to this problem, but if we want to proxy the traffic and use different methods of encryption (*STARTTLS*, *SSL*) to the client and the server, *Zorp* is still one of the best solutions. It is possible to establish an encrypted connection through the untrusted network and a plain connection through the trusted one. It is also possible the use different versions of encryption (*TLS 1.0*, *TLS 2.0*) to the client and server.

To do that, capability of establishing encrypted connections separately to the client and the server is necessary, but not sufficient. The reason is the way to upgrade a plain text connection to an encrypted (*TLS* or *SSL*) one instead of using a separate port for encrypted communication (*STARTTLS*), where understanding the protocol is a must. If we want to hide this functionality from the client and the server even if both of them support it, to solve an incompatibility problem, *Zorp* can help us. We can conceal features of the clients or the servers (for example *STARTTLS* in *SMTP*, or compression in *HTTP*) from each other.

To continue the encrypting example, *Zorp* can hide the *STARTTLS* feature of the *SMTP* server from the client, which prevents to initiate encrypted communication in this way. Certain combinations of client and server side *SSL* settings (for example when *SSL* is forced in server side) *Zorp* does it automatically.

1.2.4 Content Filtering

Content filtering is a key feature of firewalls. *Zorp* is not an exception to this rule, even if without extensions there are only limited opportunities to do that work. However, each of spam filtering, virus scanning, *URL* filtering is possible by means of external software components. Let the cobbler stick to his last. *Zorp* does nothing else, but analyzes the protocol to find the particularly interesting parts of the traffic (*URL*, downloaded data, e-mail attachment, ...) and passes it to the necessary application. As the result of the content filtering and possibly other conditions, *Zorp* may accept, reject or only log the request, or even quarantine the response. We have nothing to do, but establish connection between the *Zorp* and the chosen content filtering software (for example: ClamAV, SpamAssassin, ...) with a simple adapter application, which makes the location of the data known to the content filtering tool and forwards the result to *Zorp*.

1.2.5 Audit

Establishing an access control system is only the first step on the way to achieve a well-controlled and secure network. Operating and administrating this network is more difficult. Above all, we need to know what is happening in our network, because only this information can create the possibility to improve the access control system. On the one hand we have to answer what kind of events have violated the current network policy. On the other hand we are in need of the information whether a permitted action has happened or not and if so, than how. *Zorp* is able to log the necessary information in both cases.

The benefit of *Zorp* is the fact that we can retrieve information from the proxies in application level so events of the network can be handled in the application level also. Even requests and responses of a protocol can be recorded to the system log, which can be very useful in case of an audit. After the necessary configuration of the proxy from the log messages it can be proved whether an event has happened or not in a specific time interval and also statistics can be created based on them.

1.2.6 Flexibility

Zorp is able to solve the general uses mentioned above as it is, but the strength of the *Zorp* lies in the fact that it is easily extendable and customizable to solve specific problems. We do not need to reimplement any kind of functionality, especially the protocol analyzers, we can reuse and extend them to meet our requirements. Nevertheless the proxies are mainly written in *C*, they can also be scripted in *Python* with all of the benefits of the language. Existing ones (*HTTP*, *FTP*, ...) can be specialized, or a new one can be implemented if we want to analyze the protocol at application level only. It is possible with a special kind of proxy (*AnyPy*) which does anything, but the application level analysis, so we can focus on that job.

2.1 Basic Concepts

2.1.1 Zone

What Zone is good for?

Usually access to the services controlled by the attributes of lower layers of the *ISO/OSI* model, like IP addresses or ports. *Zorp* has an extra feature compared with other firewall suites. There is a possibility to define sets of IP subnetworks, called *Zone*.

Administrative Hierarchy

Zones group IP subnetworks that administratively belong together. What is it good for? In this way an administrative hierarchy can be created that is independent from the network topology, reflecting only the network policy. Imagine the situation when all those who are permitted to access an *FTP* servers for upload not belongs to the same IP subnet. In this case we would have to add at least two IP based rules to our network policy. If we use *Zorp* we only have to add necessary IP subnetworks to the necessary *zone*.

Inheritable Rights

Other notable feature of zones, that they can be linked to a tree hierarchy. Access control rights are inherited between the levels of the *zone* tree. A top-level access is in effect in the lower levels as long as it is not blocked. For instance a top-level access can be the right to download from *FTP* servers. When a group of users should have special rights. These special rights can be granted in the lower levels of the *zone* tree.

How Zone can be configured?

The simplest way to define a *Zone* to write the followings to the configuration file `policy.py`. It defines an empty *Zone*, which has not contain any subnetwork, but can be referred from the firewall rules by its name `zone`. Obviously it is not so useful, but it is simple as we promised.

```
Zone('zone')
```

As it has already mentioned a *Zone* groups the administratively belonging IP subnetworks together, so we have to define these subnetworks somehow to give meaning to the *Zone*. It can be done by creating the *Zone* class with additional `addrs` parameter, which value must be an iterable object, which contains IP subnetworks in CIDR notation.

```
Zone(name='intra.devel', addrs=['10.1.0.0/16', 'fec0:1::/24'])
Zone(name='intra.it',      addrs=['10.2.0.0/16', 'fec0:2::/24'])
```

How Zone works?

Inheritance

As it has also mentioned a *Zone* can refer another *Zone* as its parent, which makes possible to create a tree from the *Zone* s. This tree represents the administrative hierarchy of our network. When a *Rule* refers to a parent *Zone* in the hierarchy it implicitly refers to the whole subtree. It practically means that we accept a special kind of traffic in a parent *Zone* it will be accepted all of its child *Zone* s also.

```
Zone(name='intra',
      addrs=['10.0.0.0/8', 'fec0::/16'])

Zone(name='intra.devel', admin_parent='intra',
      addrs=['10.1.0.0/16', 'fec0:1::/24'])
Zone(name='intra.it',    admin_parent='intra',
      addrs=['10.2.0.0/16', 'fec0:2::/24'])
```

If the *Zone* hierarchy above is defined and we create a *Rule* which accepts for example the *HTTP* traffic from the *Zone* `intra` it also accepts the *HTTP* traffic from `intra.devel` and `intra.it` and any other *Zone* will be crated in the future which defined as the child of `intra` independently from the fact that subnetworks of parent and child *Zone* s contains each other or not.

Conflicts

Identical IP subnetworks – same IP and mask pair – cannot be added to different *Zone* explicitly (by `addrs` property of *Zone* class). It considered invalid configuration and rejected by *Zorp*.

2.1.2 Rule

What Rule is good for?

There is no firewall without access control and *Zorp* is no exception to this rule. When an access control policy is being created, we first have to find answers to the “who”, “what” and “how” - questions. Resources should be accessible only for a specific group of users under the defined conditions.

How Rule works?

The *Rule* answers to the “who”, “what” and indirectly the “how” questions.

Who and What?

The “who” and the “what” questions can be answered by a set of traffic properties. A specific *Rule* matches to a certain traffic when the parameters what were given to the *Rule* match to the traffic.

```
Rule(service='service_dns',
      dst_port=53)
```

In the example above the *Rule* matches to any kind of traffic which target the port destination 53. In other words it grants access to any name server on the internet. It works only when protocol is *TCP* or *UDP*, because port is not defined in case of other protocols (for example *IGRP*), but we can add another conditions to the *Rule* to make the rule definite.

```
Rule(service='service_dns',
      proto=(socket.IPPROTO_TCP, socket.IPPROTO_UDP),
      dst_subnet='8.8.8.8/32',
      dst_port=53)
```

As it can be seen multiple conditions can be defined, so the “who” and the “what” can be answered at the same time. The questions are what kind of conditions can be set, what is the relation between the different type of conditions, what is the relation between the items of a certain condition.

Conditions

First of all list the possible conditions parameters of a *Rule*. As you can see there are 8 different type of conditions, which can be set independently from each other. If more than one condition is given the rule matches only if the logical conjunction of the conditions matches. If there is more than one value in a specific condition there is logical disjunction between them.

1. VPN id (*reqid*)
2. source interface (*iface* or *src_iface*)
3. protocol (*proto*)
4. protocol type (*proto_type*)
5. protocol subtype (*proto_subtype* or *icmp_type*)
6. source port (*src_port* or *icmp_code*)
7. destination port (*dst_port*)
8. source subnetwork (*src_subnet* and *src_subnet6*)
9. source *zone* (*src_zone*)
10. destination subnetwork (*dst_subnet* and *dst_subnet6*)
11. destination interface (*dst_iface*)
12. destination *zone* (*dst_zone*)

In the *complex rule example* above the *Rule* matches when the protocol of the traffic is *TCP* or *UDP* and the destination address is *8.8.8.8* and the destination port is *53*. In general we can say if we want a more restrictive *Rule* we have to add a new condition, if want a more permissive rule we have to add a new value to an existing condition.

Best match

In contrast to the *Netfilter* where the first matching rule takes effect, in case of *Zorp* the best matching rule takes effect. It entails that the order of the rules is irrelevant. When a new connection is occurred the evaluation will check each rule against the parameters of the traffic to find the best one.

The word best in the expression *best match* means that the more accurate rule will take affect. The accuracy of a *Rule* depends on two thing, the evaluation order of the conditions and the accuracy of the specific condition in the *Rule*.

Evaluation order There is a precedence between the different condition types, which determines the order of the evaluation. It means if a rule has a condition with higher precedence it considers better that the other one. The *condition list* enumerates over the conditions in top to bottom in descending precedence. It practically means that a rule with a destination subnetwork condition is always better than a rule with destination *zone* condition and both of them are worse than a rule with a source *zone* condition and so on ...

Condition scope If two rule are considered to be identical – in other words they have conditions with the same precedence – the value of the conditions determines which one considered to be better. In general a narrower is always better than a wide scope, which means an IP subnetwork with greater prefix value, a port number instead of a port range, a child *zone* instead of a parent is more specific, so the rule with it is considered better.

How Rule can be configured?

Lets imagine the situation when we want to grant access to any kind of *FTP* server on the internet in read-only mode for everyone in our local network (10.0.0.0/8), but we have to grant read-write access to a specific server (1.2.3.4) and for a certain department (10.10.0.0/16) of our organization. How can we use the *best match* to fulfill the requirements?

First of all solve the general requirement, which is the read-only access to any *FTP* server for everyone from our subnet. It can be done by a *rule* which contains two explicit and an implicit condition and an action. The explicit conditions are about the destination port, namely 21, the standard *FTP* port, and the source subnetwork, namely 10.0.0.0/8 which is our private network in the example. The implicit condition is about the destination subnetwork that does not appear in the rule, which means it matches independently from the destination of the traffic. The action can be set by the *service* parameter of the rule which is *service_ftp_read_only* in this case.

```
Rule(service='service_ftp_read_only',
      dst_port=21)

Rule(service='service_ftp_read_write',
      dst_subnet='1.2.3.4/32',
      dst_port=21)

Rule(service='service_ftp_read_write',
      src_subnet='10.10.0.0/16',
      dst_port=21)
```

The second requirement was to grant read-write access to a specific server (1.2.3.4). It can be done by a *rule* matches “better” to the traffic than the previous one. As the second rule has a condition to the destination subnetwork (*dst_subnet*), while the first one has not, it considered to more specific, so it is a “better” match.

The third requirement was to grant read-write access for a department (10.10.0.0/16) of our organization to any *FTP* server. It is also possible by adding a new *rule* with a condition to the source subnetwork (*src_subnet*) with the necessary value (10.10.0.0/16).

The question arises, what is the *best match* to a traffic which comes from the subnetwork 10.10.0.0/16 and its destination is the address 1.2.3.4, as in this case each *rule* matches. As we have already mentioned the second and the third one more specific than the first, so the first one cannot be the bast match. Inasmuch source subnetwork condition has higher precedence than the destination subnetwork the second *rule* will be the *best match*.

2.1.3 Service

What service is good for?

The *service* answers the earlier mentioned “how” question, as it determines what exactly happens with the traffic, whether it is analyzed in the application layer of the *ISO/OSI* model or not, rejected or accepted. After the best matching *rule* has found, an instance of a *service* set in the *rule* starts to handle the new connection.

How service works?

There are three different service types in *Zorp* with completely different functionality and configuration.

PFService Transfers packet-filter level services, so if you want to transfer connections on the packet-filter level only, and you do not want analyze application-level traffic making decisions based on it, use *PFService*. It provides better performance, as the decision about the traffic can be made in kernel space by *KZorp*, without the assistance of the user space firewall (*Zorp*) itself.

Service Transfers application-level (*proxy*) services, so if you want to transfer connections on the application-level to make possible audit, analysis, restriction or modification, use *Service*. It does not provide as good performance as *PFService*, since the decision about the traffic cannot be made in kernel space (*KZorp*), it also requires the assistance of the *Zorp*, that runs in the user space, which makes deeper and also more resource-consuming operations.

DenyService New in version 3.9.8: The *DenyService* class.

Rejects the connections in a predefined way. In general, it can be used to handle the exceptions in your policy. If you have a general rule that grants access to any *FTP* servers from any subnetwork, but you want to make an exception (for example there is a prohibited server), you can create a more specific *rule* (with the server address in *dst_subnet* condition) that rejects the traffic as it is set in the *DenyService*.

How service can be configured?

Minimal configuration of a *service* depends on its type, but at least it must contain a name. The *name* parameter is used to refer to the *service* from another object (for example from a *rule*).

PFService With the defaults of the additional parameters, *PFService* transfers the traffic through the firewall in the packet-filter level without passing it to the user space (just like in *Netfilter*).

```
PFService(name='PFService')
```

Service In case of *Service*, the *proxy_class* parameter is also mandatory. This is the most important parameter in the point of view of a proxy firewall, while its value determines what will happen with the traffic in the application layer.

```
Service(name='Service', proxy_class=HttpProxy)
```

DenyService New in version 3.9.8: The *DenyService* class.

With the defaults of the additional parameters, *DenyService* drops the traffic silently (just like *DROP* target in *Netfilter*).

```
DenyService(name='DenyService')
```

2.1.4 Proxy

As it has already been mentioned earlier the network traffic analysis can take place at the application level. To perform that, *Zorp* implements application level protocol analyzers. These analyzers are called proxies in the terminology of *Zorp*. Proxies are written in *C*, and they are extendable and configurable in *Python*.

What proxy is good for?

Any kind of application level protocol analysis, restriction, modification can be done by *proxy*.

How proxy works?

Predefined proxies

Zorp contains several proxies which can be used without any improvement or modification to work on the application level traffic.

HTTP, FTP, SMTP Proxies to analyze widely used protocols

Finger, Telnet, Whois Proxies to analyze rarely used protocols.

Plug As its name shows it does nothing else, but to plug the client and server connection. It has all the benefits that other proxies have, except the protocol analysis.

AnyPy It is a simple proxy like the *Plug* proxy with a *Python* interface. It makes it possible to do anything with the application level network traffic which can be done by the help of the *Python* language, while the lower layers of the connection is handled by *Zorp*. For instance if the proxy to our favorite protocol is not implemented yet in *Zorp* we have the possibility to perform application level analysis manually.

Proxy Inheritance

As it is mentioned each proxy is configurable and extendable in *Python*. It means each proxy represented as a class in *Python* and the system administrator can inherit his own *Python* class from that to override the behavior of the parent class. A derived class inherits everything from the base class, which is necessary for the protocol analysis, so the system administrator has to care about his specific problem. For instance to change a value of a header in the *HTTP* protocol needs only an extra line of code over the lines related to the *Python* inheritance mechanism.

General SSL Handling

General *SSL* handling follows from the fact, that transport layer security is an independent subsystem in *Zorp*. It means, that *SSL/TLS* parameters can be set independently from the fact, that we perform protocol analysis or not. Consequently not only *HTTP*, *FTP*, *SMTP* and *POP3* proxies are *SSL* capable, but also the *Plug* and the *AnyPy* proxies. Server and client side *SSL* parameters can also be set independently. So it is possible to encrypt on the client side, but not on the server side and vice versa. Of course both of the sides can be encrypted.

Program stacking

Zorp is a proxy firewall, neither more nor less, but can be used to do tasks other than protocol analysis, such as virus scanning or spam filtering by integrating it with external applications. For instance in case of the *HTTP* protocol *Zorp* can forward responses to a virus scanner software. After that depending on the result of the scan *Zorp* can accept or reject the original request.

How proxy can be configured?

Zorp proxy classes can be implemented or customized in *Python* language. As the following example show the only thing we have to do is deriving a new class from the necessary base class (`HttpProxy`) and customizing its behaviour.

```
from Zorp.Http import *

class HttpProxyHeaderReplace(HttpProxy):
    def config(self):
        HttpProxy.config(self)
        self.request_header["User-Agent"] = (HTTP_HDR_CHANGE_VALUE,
                                             "Forged Browser 1.0")
```

The example above only a demonstration of a customization, it is uncommented now, we will back to later.

2.1.5 Instance

An *instance* groups the *rules* and *services* belonging in some way together and separates the resulting groups from each other.

What instance is good for?

It makes possible the independent

- modification of the elements of (for example: *rules*, *services*)
- parameters passing to (for example: log level, thread limit)
- control (for example start, stop, reload) of
- monitor (for example memory usage, thread number) of
- get statistics (for example connection information, thread rate) from

each *instances*.

How instance works?

An *instance* is represented as a process at the level of the operating system. In order of the list above an *instance*

- represented as a function in configuration (see also)
- can have separate configuration file
- runs as a separate process, so
 - can be controlled separately (see also)
 - can be monitored separately (see also)
 - can have separate statistical informations (see also)

New in version 3.9.2: The `num-of-processes` parameter.

An instance can be run as several processes which can provide better performance on multi-core processors.

How instance can be configured?

Arguments of the process can be set in the `instances.conf` where each line represents an *instance*. The line begins with the name of the *instance* followed by the command line arguments of the *Zorp* process and after two dashes (`--`) the argument of the *Zorp* controller application named `zorpcctl`.

```
default_instance --verbose 3 --policy /etc/zorp/policy.py -- --num-of-processes 1
```

Zorp instance represented as a function in the configuration file `policy.py`. Any object related to the instance declared inside this function.

```
from Zorp.Http import *

def instance():
    Service(name='service', proxy_class=HttpProxy)
    Rule(dst_port=80, service='service')
```

2.2 Minimal Configuration

2.2.1 Zorp Kernel Module

KZorp is the kernel module of the *Zorp* application level firewall. The module makes possible to make kernel space decisions about the traffic according to the configured *Zorp* policy. It also provides some extensions to *IPTables* so that you can build your own packet filter ruleset that uses *Zorp* concepts and policy objects.

Rule evaluation

Zorp communicates the policy to *KZorp* when starting up, so initial policy decisions can be applied to certain traffic in kernel space. As the result of the decision, packets are either dropped or put back to the chain of *IPTables* where the *KZORP* target has been called.

IPTables relation

The *KZorp* kernel and *IPTables* modules allow using certain *Zorp* concepts in packet filter rulesets.

It adds support for the following *IPTables* modules:

- `zone` match: you can match on *Zorp* zones (defined in the *Zorp* policy) in your *IPTables* ruleset.
- `service` match: matches on either the name or the type of the service that has been selected for the packet based on your *Zorp* policy.
- `KZORP` target: handles DAC checks, transparent proxy redirections and generic processing of packets for PF-Service services (that is, *Zorp* services that process packets on the packet filter level, not in a user-space proxy).

2.2.2 Configuration

The main problem of transparent proxy firewalls is the fact that the traffic does not target the firewall itself, but a host behind the network security device. In a usual case the traffic is forwarded to the originally targeted server, but in case of a firewall the traffic must be delivered locally to the proxy, which will connect to the originally targeted server, or another according to the policy. The divertable packets should be identified somehow in the packet filter rulesets. It can be performed by the means of transparent proxy (*TProxy*) kernel module of the kernel.

The idea is to identify packets with the destination address matching a local socket on your box, set the packet mark to a certain value, and then match on that value using policy routing to have those packets delivered locally.

—TProxy Kernel Module Documentation

The following sections will describe the *IPTables* and policy routing rules that are essential to make *Zorp* operable.

IPTables

At least the following *IPTables* ruleset is required for *Zorp*. Note that this ruleset is fair enough for *Zorp*, but it is inadequate for even the simplest firewall. The ruleset submits a working example of *Zorp*, so it must be extended with some other rules that are ordinary in case of a proxy firewall (for example: grant *SSH* access, handle *ICMP* messages).

```
*mangle

:PREROUTING ACCEPT
# mark and accept connection already handled by Zorp <1>
-A PREROUTING -m socket --transparent -j MARK --set-mark 0x80000000/0x80000000
-A PREROUTING --match mark --mark 0x80000000/0x80000000 --jump ACCEPT
-A PREROUTING -j DIVERT

:INPUT ACCEPT
-A INPUT -j DIVERT

:FORWARD ACCEPT
-A FORWARD -j DIVERT

:OUTPUT ACCEPT

:POSTROUTING ACCEPT
-A POSTROUTING -j DIVERT

# chain to collect KZorp related rules <2>
:DIVERT
# insert rules here to bypass KZorp <3>
# jump to KZorp and mark the packet <4>
-A DIVERT -j KZORP --tproxy-mark 0x80000000/0x80000000

COMMIT

*filter

:INPUT DROP
# accept earlier marked packet <5>
-A INPUT -m mark --mark 0x80000000/0x80000000 -j ACCEPT

:FORWARD DROP
# accept connection relates to a packet filter service <6>
-A FORWARD -m conntrack ! --ctstate INVALID -m service --service-type forward -j ACCEPT
↪ACCEPT

:OUTPUT ACCEPT

COMMIT
```

[*KZorp* related *IPTables* rules]

1. The `socket` matcher inspects the traffic by performing a socket lookup on the packet (non-transparent sockets are not counted) and checks if an open socket can be found. It practically means that *Zorp* (or any other application) has a socket for the traffic, it is already handled by *Zorp* in the userspace, no kernel-level intervention is required. In this case it is marked with the *TProxy mark* value (0x80000000), meaning that it should be handled by *Zorp*.
2. There are some chains of table `mangle` where *KZorp* must be hooked for certain purposes (rule evaluation, NAT handling, ...). In these cases we are jumping to a user-defined chain (`DIVERT`) where the corresponding rules can be placed to pass the traffic to *KZorp* or even bypass it.
3. This is the place where we can put rules which match to certain traffic should be hidden from *KZorp* and accept it.
4. If no rule has been matched in this chain earlier, this rule jumps to *KZorp* and also marks the packet. This mark can be used in policy routing rules to divert traffic locally to *Zorp* instead of forwarding it to its original address. Note that this mark is the same that we use in case of the first rule of the `PREROUTING` chain.
5. If the traffic has already been marked in table `mangle` with the corresponding value (0x80000000), we should accept it. For example the data channel connection of active mode FTP matches the first rule of `mangle` table `PREROUTING` chain, so it has been marked, but should be accepted in the `INPUT` chain of `filter` table as it is an incoming connection.
6. The `service` matcher looks up services specified within *KZorp*. Services can be identified by name or by type. Type `forward` means a forwarded session (or *PFSERVICE*). These kind of sessions should be forwarded in the `FORWARD` chain of the `filter` table.

Caution: The ruleset above contains those and only those rules which are essential to make *KZorp* and *Zorp* operable. The ruleset must be extended with other rules that make the firewall operable (for example: accepting incoming *SSH* connection or particular typed *ICMP* packets).

Note: The ruleset above is IP version-independent, so it can be used both in case of `iptables` and `ip6tables`.

Advanced Routing

Packets have been marked to a certain value in *IPTables*. Now match on that value using policy routing to have those packets delivered locally to *Zorp* instead of forwarding it to the original address, and *Zorp* will connect to a server depending on the policy.

```
ip -4 rule add fwmark 0x80000000/0x80000000 lookup tproxy #<1>
ip -4 route add local default dev lo table tproxy #<2>
```

[*Zorp* related policy routing rules]

1. Rule instructs the system to lookup route for the traffic from table `tproxy` if the traffic has been marked with the required value (0x80000000) in the `DIVERT` chain of the `mangle` table in *IPTables*.
2. Table `tproxy` has only one route that diverts the traffic locally to *Zorp*, so it is not forwarded as it would have been done by default.

Table name `tproxy` can be used only if the following line is added to `rt_tables` file.

```
100    tproxy
```

[*Zorp* related policy routing table names]

Note: The policy routing rules above must be repeated with options `-6` instead of `-4` to make IPv6 operable.

3.1 Protocol Enforcement

3.1.1 Use case

The most common use case of a proxy firewall – including *Zorp* – nowadays is to rule the Internet, means take control over the *HTTP* traffic. This is a simple, but good example to show the advantage of a proxy firewall technology. When the system administrator has to grant access to the World Wide Web, usually only one rule is created, which opens port 80 to the Internet. It solves the original problem, but generates another one. With the help of this rule anybody can access any kind of service of any server on the port 80 independently from the fact, that it is a *web* service or not.

3.1.2 Solution

The application level solution of the problem is enforcing the *HTTP* protocol on the traffic on the destination port 80. It is easy with *Zorp*, because there is a predefined proxy (`HttpProxy`) to enforce the *HTTP* protocol. We only have to start a *service* which sets this *proxy* as `proxy_class` parameter, when the traffic meets the mentioned requirements.

```
from Zorp.Http import * #1


def default_instance():
    Service(name='service_http_transparent', #2
           proxy_class=HttpProxy
           )
    Rule(service='service_http_transparent', dst_port=80, #3
         src_zone=('clients', ),
         dst_zone=('servers', )
         )
```

1. Imports anything from the `Zorp.Http` module, which makes it possible to use *HttpProxy*-related names without any prefix.
2. Creates a simple *service* with the name `service_http_transparent`, which uses the predefined `HttpProxy` of *Zorp*.

- Creates a *rule* with the necessary conditions, traffic from *zone* clients to *Zone* servers targets the port 80 and starts a *service* named `service_http_transparent`.

3.1.3 Result

The result is as simple as possible. The traffic goes through a transparent service without the client or the server being aware of that, while the *HTTP* protocol is enforced by the `HttpProxy` of *Zorp*.



```

It works!
This is the default web page for this server.
The web server software is running but no
content has been added, yet.

server:# grc tail -n 0 -f /var/log/apache2/access.log
172.0.20.1 -- [19/Sep/2011:00:42:06 +0200] "GET / HTTP/1.0" 200 446 "-"
"u3m/0.5.2+cv5-1.1027"

<< ↑ ↓ Viewing <>

firewall:# grc tail -n 0 -f /var/log/messages
Sep 19 00:42:05 firewall zorp/zorp_instance[1570]: core.session(3): (svc/service_http_transparent:31): Starting proxy instance; client_fd='20', client_address='AF_INET(172.0.10.1:53590)', client_zone='Zone(clients, 172.0.10.0/23)', client_local='AF_INET(172.0.20.254:80)', client_protocol='TCP'
Sep 19 00:42:05 firewall zorp/zorp_instance[1570]: core.session(3): (svc/service_http_transparent:31/http): Server connection established; server_fd='23', server_address='AF_INET(172.0.20.254:80)', server_zone='Zone(servers, 172.0.20.0/23)', server_local='AF_INET(172.0.20.1:60225)', server_protocol='TCP'

```

Fig. 3.1: Transparent HTTP proxy

3.2 Forward Proxy

3.2.1 Use case

We intend to use the firewall as a proxy server, like a *Squid* web cache.

3.2.2 Solution

The solution is very simple, since there is a proxy class that we can use to control the traffic on the proxy level. In this case, the clients connect to *Zorp* that acts as a proxy server, and allows traffic flow according to the rules, but communicates with the clients “in the proxy language”.

```

from Zorp.Http import *

def default_instance():
    Service(name="service_http_nontransparent_inband", # <2>

```



```

        proxy_class=HttpProxyNonTransparent, # <3>
        router=InbandRouter(forge_port=TRUE, forge_addr=TRUE) # <4>
    )
    Rule(service='service_http_nontransparent_inband', # <1>
        dst_port=3128,
        dst_subnet=('172.16.10.254', ),
        src_zone=('clients', )
    )

```

1. Creates a *rule* which matches only, when the traffic comes from the `clients` zone and targets the IP address `172.16.10.254` and the port `3128`, which address is the address of the client side interface of the firewall.
2. Creates a service that works like a proxy server.
3. It uses the predefined `HttpProxyNonTransparent`, because this *proxy* class – against the `HttpProxy` in the code snippet transparent proxy use case – handles the traffic as a proxy server.
4. In this case the address of the *HTTP* server, that the client wants to connect to, comes from the application layer traffic and not from the network layer, so the default `DirectedRouter`. It routes the traffic where it was originally considered to be routed, but in this case (non-transparent service) the client targets the proxy server (here, the firewall) itself. Setting `InbandRouter` as `router` can handle the situation for both the *HTTP* and *FTP* protocol.

3.2.3 Result

Now the IP address `172.16.10.254` and port `3128` can be set as *HTTP* proxy in the internet browsers.

3.3 Reverse Proxy

3.3.1 Use case

A common requirement is the following case: Client connects to a proxy server, that appears to the client as an ordinary server, but it forwards the request to the origin server, which handles it. Thus, we communicate with the origin server through a proxy server. For example we reach a mail server in *DMZ*, we connect to a firewall, but in reality, we communicate with the *SMTP* server in the *DMZ*.

3.3.2 Solution

The communication can be inspected on the protocol level, since the *SMTP* proxy is available in *Zorp*. Based on the abovementioned example, we connect to the firewall, and indirectly communicate with another server through the firewall.

```

from Zorp.Smtplib import *

def default_instance():
    Service(name="service_smtp_transparent_directed", # <3>
        proxy_class=SmtplibProxy,
        router=DirectedRouter(dest_addr=SockAddrInet('172.16.20.254', 25)) # <2>
    )
    Rule(service='service_smtp_transparent_directed', # <1>
        dst_port=25,
        src_zone=('dmz', ),
        dst_subnet=('172.16.40.1', )
    )

```

```
)
```

1. This case is similar than it was at the *HTTP* forward proxy code snippet, the clients connect to the firewall directly, so the destination IP address is the firewall's address (172.16.40.1) and the port is the standard port of *SMTP* (25).
2. The service uses the predefined `SmtPProxy` proxyclass to enforce the *SMTP* protocol.
3. As the clients target the firewall, the traffic must be routed to the origin server (172.16.20.254) directly. As its name shows, this function can be solved by the `DirectedRouter` class, where the `dest_addr` parameter contains the address and the port value of the origin server.

There is another relevant question in case of a *forward proxy*. As the firewall connects to the origin server, in the log of the *SMTP* server on the origin server it would always show the IP address of the firewall, if we did not extend the router with following parameter:

```
router=DirectedRouter(dest_addr=SockAddrInet('172.16.20.254', 25),
                      forge_addr=TRUE
                      )
```

The `forge_addr` and `forg_port` options of the *router* can be used to forge the client address and port to the traffic instead of the firewall's ones.

3.3.3 Result

The client connection is forwarded to the origin server by the firewall to handle it. Replies are forwarded back to the client in a transparent way (at the application level). The client is not aware of the forwarding, so additional settings are not required – like proxy in the *forward proxy* use case – on the client side.

3.4 Access Control

3.4.1 Use case

It is a general use case that we want to grant access for a user to an *FTP* server on the Internet to allow downloading anything, but at the same time we want to prevent them from uploadin anything.

3.4.2 Solution

The application level solution of the problem is to accept the read-only commands of the *FTP* protocol, but drop the commands used to write to the server (for example: `PUT`). As it is a general issue, *Zorp* provides a predefined proxy to perform that, so the system administrator does not have to do anything to implement a read-only *FTP* access, only use that proxy.

```
from Zorp.Ftp import *

Service(name="service_ftp_transparent", # <1>
        proxy_class=FtpProxyRO
        )

Rule(service='service_ftp_transparent', # <2>
      dst_port=21,
```

```

src_zone=('clients', ),
dst_zone=('servers', )
)

```

1. Creates a *rule* that matches the *FTP* traffic, as the destination port in the *rule* is the standard port of the *FTP* servers (21).
2. The service uses the predefined `FtpProxyRO`, which analyzes the traffic and when we issue a read-only command, it will be sent successfully to the server. When we issue a write command an error message will be sent to the client, but nothing will be sent to the server, as *Zorp* rejects them.

3.4.3 Result

Any kind of read-only operation works successfully, but error message is displayed on the client side when it tries to perform a write operation on the server.

```

client:# lftp server.zorp
lftp server.zorp:> dir
-rw-r--r--  10      0
18:00 test
lftp server.zorp:> get test
lftp server.zorp:> put test
put: Fatal error: 500 Error parsing command
lftp server.zorp:>

server:# grc tail -n 0 -f /var/log/vsftpd.log
0 Sep 19 Mon Sep 19 18:08:06 2011 [pid 2] CONNECT: Client "172.0.20.1"
Mon Sep 19 18:08:06 2011 [pid 1] [ftp] OK LOGIN: Client "172.0.20.1", and
n password "lftp@"
Mon Sep 19 18:08:14 2011 [pid 3] [ftp] OK DOWNLOAD: Client "172.0.20.1",
"/test", 0.00Kbyte/sec

firewall:# grc tail -n 0 -f /var/log/messages
Sep 19 18:08:03 firewall zorp/zorp_instance[1942]: core.session(3): (svc/service_ftp_transparent:10): Sta
rting proxy instance; client_fd='20', client_address='AF_INET(172.0.10.1:59922)', client_zone='Zone(clien
ts, 172.0.10.0/23)', client_local='AF_INET(172.0.20.254:21)', client_protocol='TCP'
Sep 19 18:08:03 firewall zorp/zorp_instance[1942]: core.session(3): (svc/service_ftp_transparent:10/ftp):
Server connection established; server_fd='23', server_address='AF_INET(172.0.20.254:21)', server_zone='Z
one(servers, 172.0.20.0/23)', server_local='AF_INET(172.0.20.1:57608)', server_protocol='TCP'
Sep 19 18:08:03 firewall zorp/zorp_instance[1942]: ftp.policy(3): (svc/service_ftp_transparent:10/ftp): R
ejected answer; from='211 End', to='500 Error parsing answer'
Sep 19 18:08:23 firewall zorp/zorp_instance[1942]: ftp.policy(3): (svc/service_ftp_transparent:10/ftp): R
equest rejected; req='ALLO'
Sep 19 18:08:23 firewall zorp/zorp_instance[1942]: ftp.policy(3): (svc/service_ftp_transparent:10/ftp): R
equest rejected; req='STOR'

```

Fig. 3.2: Read-only FTP proxy

The configuration file snippet above grants read-only access to the servers in the zone `servers` for the users who come from the zone `clients`. Without the `src_zone` and `dst_zone` filters the *rule* would grant access to any server for any user.

4.1 Troubleshooting

4.1.1 Kernel debugging

Dynamic debugging

If the dynamic debugging is enabled in your kernel configuration, *KZorp* debug messages can be enabled and disabled dynamically. Before enabling any debug messages or leaving them enabled, consider the fact that logging these messages may cause serious performance issues especially in case of heavy traffic.

```
echo 'file kzorp_netlink.c +p' > /sys/kernel/debug/dynamic_debug/control # <1>
echo 'module kzorp +p' > /sys/kernel/debug/dynamic_debug/control # <2>
```

1. Enables debug messages in source file `kzorp_netlink.c`
2. Enables debug messages `kzorp` module

Debug messages can be disabled with the same command, using the `-p` switch after the name of the source file or module instead of `+p`. For details, read the [dynamic debug howto](#).

Function tracer

If the dynamic debugging support is not enabled in our kernel, there is another possibility to trace *KZorp*, the kernel part of *Zorp*. Functions are traced whether there are debug messages in them or not. *KZorp* related functions have a `kz_` prefix in their names, so tracing them can be enabled with the following commands:

```
sysctl kernel.ftrace_enabled=1 # <1>

cd /sys/kernel/debug/tracing
echo function_graph > current_tracer # <2>
echo 'kz_*' > set_ftrace_filter # <3>
```

```
echo 0 > tracing_on # <4>
sleep 1
echo 1 > tracing_on # <5>
```

1. Checks that `ftrace_enabled` is set in the kernel configuration, otherwise this tracer is a nop.
2. Sets the tracer that provides the ability to draw a graph of function calls similar to *C* code.
3. Limits tracing to functions with names starting with the `kz_` prefix.
4. Starts the tracing.
5. Stops the tracing.

The result of the trace can be read in the `trace` file. For details, read the [function traces documentation](#).

A

access control, 7, 8
audit, 6

B

best match, 10

C

content filtering, 5
 spam filtering, 5
 URL filtering, 5
 virus scanning, 5

D

DMZ, 21

E

encryption
 SSL, 3, 5
 TLS, 3, 5

F

forward proxy, 22

I

integration, 3
IP, 4, 7

L

licence
 dual-licensin, 4
 GPL, 4
logging, 6

N

Netfilter, 2, 10

O

OSI model, 2, 4, 7

P

program stacking, 12
programming language
 C, 2, 6
 Python, 2, 6
protocol
 FTP, 2, 7, 21, 22
 HTTP, 5, 19, 21
 user-agent, 5
 SMTP, 5, 21
proxy
 AnyPy, 6, 12
 forward proxy, 2, 20
 Plug, 12
 reverse proxy, 2

R

reverse proxy, 21

S

spam filtering, 3
Squid, 20
subnetwork, 4, 7

V

virus scanning, 3

Z

Zone, 4, 7
Zorp
 Zorp GPL, 4
 Zorp Professional, 4