# PyWall: A Python Firewall

By the PyWall Team
Stephen Brennan, Jeffrey Copeland, Yigit Kucuk, Andrew Mason

## Background

### Firewalls

Firewalls permit or deny network traffic based on a configurable set of criteria. Typical filtering rules include source/destination IP address, protocol, and source/destination port. Modern firewalls include stateful rules that consider a sequence of interactions over time instead of individual packets, allowing firewalls to implement features like TCP connection tracking and port knocking. Because of their simplicity and subsequent speed, firewalls often serve as a first line of defense against network-based attacks.

Understanding the underlying mechanisms of a firewall is important if one is interested in computer security. Because implementing a system is the best way to understand its underpinnings. we have implemented a firewall using Python and some Netfilter tools. Before going into the details of our implementation, we should give some background on TCP, the Netfilter project, **python-netfilterqueue**, and IPTables.

### TCP (Transmission Control Protocol)

TCP, along with UDP, is responsible for nearly all traffic at the transport layer of the internet. TCP in particular is worth noting for two reasons. First, it is overwhelmingly the most popular transport layer protocol, accounting for 82% of traffic and 91% of bytes transmitted on the Web according to a 2013 study. [3] As a result, having a firewall that can filter on TCP packets is of the utmost importance. Second, TCP is connection-based, and therefore a stateful protocol. This adds a level of complexity to implementing anything more than simple port filters which will be discussed later in the paper.

### Netfilter Framework

The Netfilter framework is a set of packet filtering packages included in the Linux kernel. Of concern to our project, netfilter provides the ability to hook into kernel modules at various points in the network stack. Several libraries, notably **python-netfilterqueue** (discussed below), leverage this feature to listen for incoming packets and then filter them.

Using the Netfilter framework ordinarily would add an additional layer of complexity to the scope of the project since it would require implementing our own kernel module. However, developing a Linux kernel module is very difficult and time consuming. They must be written in C, which presents a serious amount of development overhead for a term project. Additionally, debugging kernel modules is more difficult than debugging regular C programs, which would add even more difficulty to the project. Ideally, we would implement a firewall in a high-level language so we could focus on the concepts behind the implementation, without getting bogged down in the nuts-and-bolts of C and kernel module development. Thankfully, we were able to find a way to develop a firewall without creating a kernel module.

### python-netfilterqueue

One package that assisted us was **libnetfilter_queue**, a C library from the Netfilter project that gives user processes control over packets the kernel has received. This library can then inspect, modify, accept, and/or drop the given packet -- precisely the functionality needed for a firewall. [1] Additionally, there is a library called **python-netfilterqueue** that provides Python bindings for **libnetfilter_queue**. This gave us the ability to attach a callback to an IPTables queue that gets called each time the kernel receives a packet. [2] Having low-level access in a high-level language like Python allowed us to focus more on firewall functionality than on C's nitty-gritties.

### IPTables

Part of the Netfilter suite, IPTables allows sysadmins to configure the various tables in the Linux firewall. As mentioned above, we used a Python library to receive callbacks from the **libnetfilter_queue** package. The following commands were used to set up and tear down the firewall, respectively (readers should note that these commands, and therefore our firewall, must be run as root):

- `iptables -I INPUT -j NFQUEUE --queue-num [number]`
  - Insert a rule into the iptables to hook into NetfilterQueue at the given [number] in the queue
- `iptables -D INPUT -j NFQUEUE --queue-num [number]`
  - Remove the rule that hooks into NetfilterQueue (tears down the firewall)

## Aims

The aim of the project was of course to implement a firewall. Specifically, we wanted to produce an academically interesting firewall rather than a production-ready, high-performance firewall. For this reason, and those discussed in the above section, we chose to implement this project in the Python programming language, using the bindings provided in the **python-netfilterqueue** package.

A typical firewall includes two components: an ingress and an egress filter. The ingress portion filters incoming packets, and the egress portion filters outgoing packets. Egress filtering is important in order to prevent attacks "from the inside", where a malicious program on the machine running the firewall creates an outgoing connection. Although this is an important application of a firewall, we chose to implement only an ingress filter. Implementing both ingress and egress filtering would have added extra implementation effort, without adding much additional academic interest in our implementation. Instead, we decided to focus on implementing simple stateless firewall rules, as well as two firewall rules that are significantly more complex than what was presented in class on the topic of firewalls.

The first complex rule we decided to implement was a stateful TCP connection tracking rule. This type of rule is important in order to filter based on whether packets are part of existing connections. Part of the netfilter project includes a library, **libnetfilter_conntrack**, which gives access to the kernel's connection state table. However, we decided that using the kernel's connection table would take away a key aspect of the concepts behind the implementation of a firewall. Therefore, we chose to implement our own TCP connection tracking facility.

Our second complex rule was a port knocking rule. Port knocking is a technique where an external host can gain access to a port on the host running PyWall, by executing a correct sequence of "knocks." These knocks are connection requests on specific TCP or UDP ports. This layer of protection is a useful way to allow remote connections to your computer, while preventing unauthorized hosts from accessing those same ports.

## Design

### Design Constraints

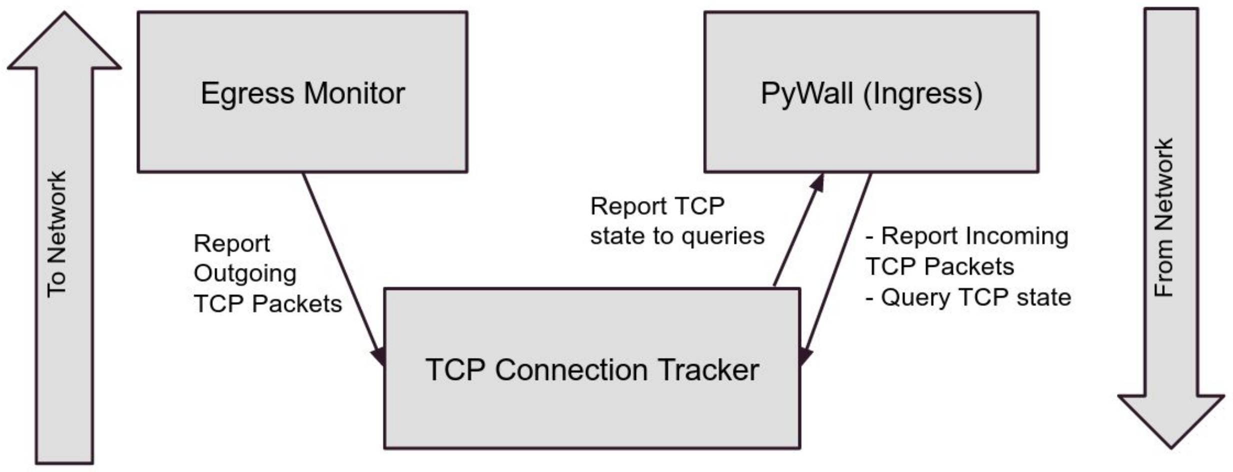
One constraint we faced during the development of PyWall was the design of the Python binding to **libnetfilter_queue**. In order to implement something as complex as a TCP connection tracker, a program must be able to examine both ingress and egress traffic -- one side of the connection is not enough to track the TCP connection state. So, PyWall had to be capable of receiving ingress and egress traffic, even though our aim was only to create an ingress firewall. The design of **libnetfilter_queue** is such that we had a few options to direct incoming and outgoing traffic to a userspace program:

- Create rules redirecting input and output to the same queue. This would require that the userspace program separate ingress from egress packets via packet headers, which are easily forged.
- Alternatively, we could create two queues, and redirect input into one, and output into another. This requires that the userspace firewall listen bind and read from two netfilter queues.

The C library provides an interface where a single queue binding provides a file descriptor [4]. In theory, more than one queue handle and file descriptor could be created, and a program could listen to more than one stream of packets (for instance, by making use of the POSIX `select` system call). This way, a single program could handle an ingress and an egress packet stream in the same process. However, the Python binding to **libnetfilter_queue** creates a different API. It requires the user to create an instance of the `NetfilterQueue` class, bind to a single queue, and provide a callback. Then, the API user invokes the `run()` method of the queue instance, and the program hangs until a packet is received, at which point the callback is called with the packet data. The consequence of this API structure is that, while a C program could listen to multiple queues, a Python program is limited to a single queue for a single process.

## Process Structure

Due to the limitation above, we had to create a multiprocess system in order to track connection state. We defined three different processes: the connection tracking process, the egress monitor process, and the ingress filter process.



The connection tracker process is the master process. On startup, it creates three Python multiprocessing communication channels: two queues (single directional communication) and one pipe (bidirectional communication). It then spawns the egress and ingress processes. It provides the egress process with a queue, and the ingress process with a queue and a pipe. The queues are used by the ingress and egress monitor to report the SYN, ACK, and FIN flags for each TCP packet that goes through the monitor. The pipe is used by the PyWall ingress filtering rule to request the status of a TCP connection. The TCP connection tracker sets up the processes and IPC, and then it creates an instance of the `PyWallCracker`, which then handles TCP connection tracking.

The Egress monitoring process simply waits on a netfilter_queue. If the packet received from the queue is TCP, it places the connection tuple (source IP, source port, destination IP, destination port) along with the three important flags (SYN, ACK, FIN) into the queue it shares with the connection tracker. Finally, it always accepts packets it receives from Netfilter queue. The PyWall (Ingress) process has more duties than just TCP monitoring, but it performs the TCP monitoring in the same way, except that it only reports TCP packets that are accepted by the firewall.

The `PyWallCracker` implements the task of tracking TCP connection state. It receives the three communication channels outlined above, and it waits (using the POSIX `select` system call) for input from one of them. If it receives input from the query pipe, it looks up the queried connection in its table. If it exists, it returns the state of the connection, otherwise it reports the connection state as "CLOSED". When the connection tracker receives input from one of the packet reporting queues, it looks up the connection, and performs the transition specified by the TCP connection state machine.

In order to encode the TCP connection state machine, we first looked up the specification for TCP, RFC 793 [5]. This document, while originally published in 1981, is still accurate in the majority of its specification of TCP, including its state machine (according to RFC 7414, published in February 2015) [6]. The TCP state machine presented in RFC 793 specifies the lifetime of a TCP connection, from open to close. We implemented this entire state machine within `PyWallCracker` except for a few key differences:

- RFC 793 has CLOSED and LISTEN as separate states, but from the perspective of TCP segments sent on the wire, there is no difference between these states, so we grouped them together.
- RFC 793's state machine specifies the states from the perspective of the TCP "network stack." That is, it contains transitions in which the TCP stack receives a segment and sends a segment in the same transition. Since our connection tracker only observes a single segment at a time, such transitions were broken into two transitions, with an intermediate state. For example, due to the two-step transition between LISTEN and SYN_RCVD, we were forced to break SYN_RCVD into two states, SYN_RCVD1 and SYN_RCVD2.

Using this modified state machine, we created two long "case" statements, one for received segments, and one for sent segments. We included a case for each state where a transition could occur (either for receiving a segment or sending a segment, depending on the case). In the event that a segment's flags did not match a legal transition according to the state machine, we output an error message, and kept the connection in the same state.

The only exception to this rule was for segments whose connections were in the CLOSED state, but whose flags did not match any legal transition from CLOSED to another state. We observed that this would only occur when the firewall started up, and existing TCP connections were running. When the connection tracker observes an illegal transition out of CLOSED state, it assumes that this is due to an existing TCP connection that had not been recorded, and it transitions that connection into the ESTABLISHED state. This does not compromise the security of the firewall and connection tracker, due to two assumptions. The first is that, by virtue of writing an ingress firewall, outgoing traffic is trusted implicitly. The second is that, although an incoming TCP segment could theoretically cause the transition from CLOSED to ESTABLISHED, it would have to be accepted by the firewall before it would be sent to the connection tracker to do so. Therefore, any packet incoming segment which caused the CLOSED to ESTABLISHED transition would already be a trusted segment.

There is one other process not mentioned in the above architecture. It is a logging process, whose purpose is simply to receive log messages from the other processes and collate them into a single log for the entire process. A single log process avoids redundancy in the code, and prevents issues from simultaneous file writing by multiple processes.

### Class Structure

Our packet filter's design mirrors that of IPTables. In principle, the packet filter is made up of two types of objects: chains and rules. Chains are named sequences of rules. Rules may either match a packet and cause a jump to a new chain, or fail to match, in which case the packet is tried against the next rule in the chain. Packets start in the INPUT chain, and they end in one of two chains: ACCEPT, or DROP. In the case that a packet reaches the end of a chain, and no rule has matched, there is a configurable default chain specifying either ACCEPT or DROP. This way, the firewall rule set can be created to drop only certain packets, or to allow certain packets.

To allow this interface, we define several classes. The `Rule` class is abstract, and it defines the base for all other classes. It defines a single function, `__call__()`. This is a Python "magic method" which allows instances of the class to be called as if they were functions. The method must return either the name of the next chain, if the rule matches, or `False` otherwise. In addition to the `Rule` class, we also define a subclass named `SimpleRule`. This class accepts the target chain in its constructor, and defines a single method named `filter_condition`. When the filter condition is true, it will return the target chain. When the filter condition is false, it will return `False`. This `SimpleRule` serves as the foundation for most of our rules. However, some rules (such as the `PortKnocking` rule) require more fine-grained control of the the packet's return chain, and so they simply subclass `Rule`.

## Usage

### Main Program

To run PyWall, enter the directory containing `main.py` and call `sudo python main.py` <u>`path-to-config-file`</u>. (PyWall requires root to bind to a network layer socket, modify IPTables rules, and to use **libnetfilter_queue**.)

PyWall config files are specified in JSON. The config parser loads the top level JSON dictionary, and looks for the "default_chain" attribute, which specifies whether unmatched packets are put into "ACCEPT" or "DROP". Every other entry in the config is the name of a PyWall chain, mapping to a list of rules. Every rule is also a JSON dictionary, with the rule class name specified in the field "name" and with other entries being keyword arguments to that rule's constructor. (See `example/example.json` for an example configuration file.)

### Rule Classes

- `PortRule` - Filters on a single source/destination port
- `PortRangeRule` - Filters if source/destination port falls within a given range

- `SourceIPRule` - Filters on a source IP address
- `DestinationIPRule` - Filters on a destination IP address
- `IPPortRule` - Filters on a combination of PortRule and Source/DestinationIPRule
- `PortKnocking` - Allows connection on specified port after a sequence of doors (i.e. pair of protocol and port) are knocked on (i.e. have connection requests on).
- `PrintRule` - Prints the packet on standard output.
- `TrueRule` - Always matches. Most useful to prevent a chain from falling through to the default chain.
- `TCPRule` - Matches if a connection is TCP.
- `TCPStateRule` - Matches if a packet is part of a connection with a given state.

For the list of arguments that each of these rules take, please refer to `RULES.md`.

## Testing

PyWall has three types of tests: unit tests (`./unit_test`), integration tests (`./run-integration-tests.py`), and acceptance tests (`./run-acceptance-tests.py`).

Unit tests live in the `test/unit` subdirectory. They create a `Packet` object and and confirm that `Rules` accept, drop, or pass these forged packets as specified. Because Rules operate on internal Packet objects, testing `Rules` can happen independently of the firewall's remaining machinery.

Integration tests live in the `test/integration` subdirectory and check for the firewall's correctness with most parts moving. For each test, the firewall is started with a given configuration, and two sockets, a client and a server (listener), are opened on the loopback interface. The server will listen for a connection from the client. If the configuration is supposed to block some operation, the socket read should time out; if the configuration is supposed to allow some operation, the server should accept a connection and/or read some data. The listener reports whether or not it received a connection back to the test case, which in turn determines whether the test passes or fails.

Acceptance tests live in the `test/acceptance` subdirectory and they simulate a typical use case by having a remote machine perform some network operation on a target machine running PyWall. There are two components to these acceptance tests -- local scripts, which run on the target and live in `test/acceptance/local` (i.e. the machine running PyWall), and remote scripts, which run on the remote host and live in `test/acceptance/remote` (i.e. the machine that will send packets to the target.) The `PyWallAcceptanceTestCase` class sets up a listener on the target, then starts the specified remote module on the remote host via SSH. (The configuration file for this can be found in `test/acceptance/local/conf.py`) The listener then reports whether or not it receives a connection or data to the test case, which then decides the outcome of the test.

## Limitations & Future Work

### TCP Connection Tracking

Unfortunately, creating a TCP connection tracking system is very complex. TCP implementations differ from the RFC 793 specification, and so a connection tracker written to the specification sometimes fails to follow the state of some connections. For instance, RFC 1122 describes a half-duplex close sequence, which is implemented by the Linux TCP stack [7]. Additionally, we noticed that some TCP implementations perform a compressed close sequence: "FIN - FINACK - ACK," as opposed to the close sequence specified by RFC 793: "FIN - ACK - FIN - ACK." We were able to modify our state machine to accept the compressed close sequence, but we did not make any attempt to support half-duplex close sequences. When an unrecognized set of flags appear in a TCP connection, we simply keep the connection in the same state, and log an error. A better approach might be to mark the connection as "INVALID" and begin rejecting its packets. However, this would adversely affect valid TCP connections which use non-standard extensions from the kernel's TCP implementations, so we decided against that approach.

One way to address all of these issues would be to simply make use of the kernel's connection state table. This would allow the firewall to be completely aligned to the TCP stack on the host machine. As we mentioned in the Aims section, we decided to focus on TCP connection tracking as a main component, and using the kernel's state table would have dodged the complexities presented by this task.

### Netfilter Queue Bindings

Another limitation we faced was the design of **python-netfilterqueue**. As discussed in the Design section, the design of **python-netfilterqueue** limited our Python program to listen on one queue per process. However, this is not a fundamental limitation of the **libnetfilter_queue** C library. We could substantially reduce the complexity of PyWall if we had a binding which allowed us to bind to and discriminate between multiple queues. Our resulting firewall would have a single process, and therefore a much simpler design. Given more time, we could have created such a binding. However, we did not have enough time to create such bindings, and such a task was out of scope for our project.

### Egress Filtering

As mentioned briefly in the Aims section, although PyWall has queues for both ingress and egress packets, it currently only performs ingress packet filtering. A firewall that lacks egress filtering is susceptible to attacks from the inside; any program that finds its way a machine running such a firewall can send out packets of any type to anywhere on the network. This poses two threats:

- An attacker can use the compromised system as the launching point of another attack. The malicious program can send packets to other machines on the network which had trusted the compromised machine, and the ingress-only firewall can do nothing to stop this.

- An attacker can trick the firewall into allowing malicious data in from the network. If the malicious program opens a TCP connection to the attacker's IP address, the TCP connection tracker will mark that connection as established. At this point, the attacker can send whatever data he or she wants through this TCP connection.

With such threats in mind, future work for PyWall would be to add support for egress packet filtering. Having egress packet filtering would thwart attacks from the inside, and make our firewall considerable for practical use.

## Performance Limitations

A second concern regarding our firewall is related to its speed, as noted above. Because we have implemented our firewall in Python, a high-level language, it is unavoidably slower than a C implementation. We are aware of this limitation and reiterate that the purpose of this project was to explore the implementation details of firewalls and packet filtering, not to get slowed down with the implementation details of C and kernel hooking.

We measured two performance benchmarks: latency and throughput. Latency was measured by pinging a host on the same wireless subnet with the ping tool and recording the RTT over 64 trials with PyWall running with the example config and without PyWall running at all. We found that the average ping was 1.236±0.86 ms without Pywall running and 1.611±1.317 ms with PyWall running. Overall, it appears the latency increase from running PyWall does not severely hamper performance on the host machine. The results are summarized in Table 1, below.

| | Min (ms) | Mean (ms) | Max (ms) | StDev (ms) |
|---|---|---|---|---|
| With Pywall | 0.686 | 1.611 | 8.818 | 1.317 |
| Without | 0.752 | 1.236 | 5.579 | 0.860 |

**Table 1**: *Latency comparison*

We measured the throughput by downloading the Ubuntu 15.04 image (1.1 GB) from a web server on Case's network. The results are summarized in Table 2, below.

| | |
|---|---|
| Time with PyWall | 10:22.75 |
| Time without PyWall | 11:04.98 |

**Table 2:** *Throughput comparison.*

The throughput results above suggest a speed-up of ~7% by using PyWall. Obviously, this must be within the margin of error for our download time. However, this result does suggest that PyWall does not significantly slow down a normal internet connection.

(One group member consistently saw PyWall being removed from IPTables' input queue when transferring large files and was unable to rebind. However, 1) this behavior was not reproduced on other members' machines, and 2) because no exception was thrown and no exceptional value was returned, this issue could not be fixed without changing libraries, which goes beyond the scope of this project.)

## Conclusion

In our project, we aimed to create an academically interesting firewall. Our resulting implementation, PyWall, succeeds in this endeavor. PyWall implements a stateful ingress firewall in a high-level language which is easy to understand. In addition to simple stateless rules, we implemented TCP connection tracking and port knocking, as extensions to the basics of packet filtering. As such, our firewall can be used as a learning tool for people who are interested in learning the concepts of packet filtering, connection tracking, and port knocking, without getting lost in the complexities of C.

## References

[1] H. Welte, (n.d.). “The netfilter.org ‘libnetfilter_queue’ project” [Online]. Available: http://www.netfilter.org/projects/libnetfilter_queue/

[2] M. Fox, (2011, Oct. 14). “python-netfilterqueue” [Online]. Available: https://github.com/kti/python-netfilterqueue

[3] S. Kulkarni and P. Agrawal, in *Analysis of TCP Performance in Data Center Networks.* New York: Springer, 2013, p. 31.

[4] H. Welte, (n.d.). “libnetfilter_queue Documentation” [Online]. Available: https://netfilter.org/projects/libnetfilter_queue/doxygen/

[5] *Transmission Control Protocol,* RFC 793, 1981.

[6] *A Roadmap for Transmission Control Protocol (TCP) Specification Documents,* RFC 7414, 2015.

[7] *Requirements for Internet Hosts -- Communication Layers,* RFC 1122, 1989.