P6B

# Recent work in *Open*BSD relayd

**SSL Interception and other Advancements**

Reyk Floeter (reyk@openbsd.org)

March 2013

## Abstract

relayd first appeared[7] in OpenBSD 4.1, formerly called hoststated, to provide a service that helps Server Load Balancing (SLB) with OpenBSD's PF. It was written by Pierre-Yves Ritschard and Reyk Floeter. The daemon initially provided health checking capabilities of monitored backend servers and the ability to dynamically load PF tables and "rdr" L3-redirections based on the configuration and active hosts. It has been extended with support for L7-relaying of various protocols including TCP, UDP, HTTP, DNS, and SSL with optional transparent proxying capabilities and evolved into an Application Level Gateway (ALG). This paper introduces some of the latest enhancements, including integrated SSL interception or "SSL Man-in-the-middle (MITM)" support, socket splicing support for enhanced performance, the redesigned filtering subsystem for relays, and the file descriptor accounting technique.

## 1 Introduction

`relayd` is a powerful tool that combines several heavy networking duties in a typical - or not so typical - OpenBSD daemon. It is a relay, proxy, application level gateway, load balancer, link balancer or simply the complementing service to OpenBSD's Packet Filter (PF) in the kernel. It has a growing active user base and runs in many production networks around the world with millions of users a day.

This paper briefly introduces some of the recent features and advancements in relayd. Most of these features arose out of real use cases or will impact the way that `relayd` operates in its day to day usage. Not all of these features have been included in OpenBSD yet but all of them will be part of a future release; the following table displays the availability of those features based on the OpenBSD release:

| Feature | Release |
|:-:|--:|
| SSL Interception | OpenBSD 5.4 |
| TCP socket splicing | OpenBSD 5.0 |
| HTTP socket splicing | OpenBSD 5.4 |
| Filter rewrite | OpenBSD 5.4 (planned) |
| File descriptor accounting | OpenBSD 5.3 |

In addition to the features, the daemon is being constantly improved by the OpenBSD community. Many bug fixes, improvements and cleanups are actively done in `relayd` , with support from many fellow developers like most recently Sebastian Benoit (benno@openbsd.org), Alexander Bluhm (bluhm@openbsd.org) or Stuart Henderson (sthen@openbsd.org). The CVS history shows commits from 40 different developers over the years, not counting any patches that have been provided by the user community.

## 2 SSL Interception ahead

`relayd` already supports running as a transparent non-caching HyperText Transfer Protocol (HTTP) proxy that can be used for Uniform Resource Locator (URL) filtering and policy enforcements. It also supports running as an Secure Sockets Layer (SSL) server and can be used to terminate SSL connections and to forward them as plain Transmission Control Protocol (TCP). The marketing term for this is "SSL acceleration" and is typically used to provide HTTP with SSL (HTTPS) on the load balancer for HTTP-only webservers in the pool.

Some time ago, additional support for running `relayd` as an SSL client has been added which allows to terminate plain TCP connections and transparently tunnel them through SSL. In some ways, this functionality can be compared with the popular stunnel[8] program and can even be considered to be some kind of an SSL VPN.

When combining both modes, SSL server and client, it will become a man-in-the-middle that can filter between two SSL connections. This is actually already possible with the current version of relayd . However, for an effective SSL-MITM, or "SSL Interception", it also has to do some trickery with the certificates. An SSL client, especially web browsers for HTTPS, will normally check the server certificate to match at least the following conditions:

a) Is the certificate valid?

b) Is it signed by a trusted and valid Certificate Authority?

c) Does it pass additional policy checks, most importantly having a matching domain name in the certificate?

Some existing solutions and firewall vendors support SSL interception (e.g. for URL filtering on secured connections) by generating client certificates on-the-fly with a local Certificate Authority (CA) that is accepted by the clients. This way, secured HTTPS connections from web browsers behind a corporate gateway can be URL-filtered and "intercepted". The remaining problem is that the web browsers normally dont recognize the local CA, but having a corporate infrastructure allows the deployment of a custom CA certificate on internal clients. Another solution is to obtain an official CA with private key or to get an intermediate CA - a local CA signed by an official CA. Getting an official CA or intermediate CA for SSL Interception is normally only possible for governmental authorities (e.g. TURKTRUST in Turkey), or people who have access to a possibly compromised CA (e.g. DigiNotar in the Netherlands). Regarding the large number of official CAs that are accepted by modern browsers, it is supposedly a fairly common practice that keeps on questioning the CA model itself.

The major problem is that the CA model obviously depends on third parties that can only marginally be trusted. Server certificates are only signed by a single CA - it is a hierarchical model and not a network of trust - and browsers do not offer a user-visible way to double-check the certificates over an alternative channel.

There is currently no active solution to improve the situation yet and this practical paper neither proposes an effective solution nor discusses any other proposals. However, there a few countermeasures that are used to mitigate the problem:

**Client-side SSL storage:** The OS and application implementations simply try to make it harder to install custom CAs and to display even more warnings if anything suspicious has been detected. The Safari web browser on Apple's OSX uses the system key chain that requires superuser privileges to import custom CA Root certificates - simply accepting and trusting the certificate in the browser will not stop Safari from warning the user about the unofficial certificate.

**Server-side monitoring:** It has been reported that Google is actively monitoring the certificates when accessing its own servers with the Chrome browser which allows the company to detect possible interception attempts. This shouldn't be a problem for interception behind a corporate gateway with a local CA - but it allowed Google to detect the misused CA certificate that was issued by TURKTRUST[3] to a governmental authority in Turkey.

**Manual validation:** And, of course, users can try to compare the returned server certificates over different channels, for example by accessing a public SSL side from different locations and Internet connections like the corporate network and a mobile network.

## 2.1 A controversial Implementation

This situation inspired the implementation of SSL Interception for relayd . If "SSL Interception" is a fairly common feature in commercial firewall products, for example from Juniper[5] or Check Point[4], why shouldn't it be freely available in OpenBSD as open source software? This might even have the effect that the increased availability of the feature will raise the awareness of the problem and lead to practical solutions in the future.

The code is not part of OpenBSD's official source tree yet, but a patch has been posted to its "tech@openbsd.org"[6] mailing list, will be committed soon and will be included in the next official OpenBSD 5.4 release.

When relayd is configured for SSL Interception, it will listen for incoming connections that have been diverted to the local socket by PF. Before accepting and negotiating the incoming SSL connection as a server, it will look up the original destination address on the diverted socket, and pre-connect to the target server as an SSL client to obtain the remote SSL certificate. It will update or "patch" the obtained SSL certificate by replacing the included public key with its local server key because it doesnt have the private key of the remote server certificate. It also updates the X.509 issuer name to the local CA subject name and signs the certificate with its local CA key. This way it keeps all the other X.509 attributes that are already present in the server certificate, including the "green bar" extended validation attributes. Now it finally accepts the SSL connection from the diverted client using the updated certificate and continues to handle the connection and to connect to the remote server.

## 2.2 Configure the Man in the Middle

This section describes a simple configuration to test the SSL Interception with HTTPS connections by using a local CA. It depends on the latest version of relayd in OpenBSD 5.3 and the patch that can be found in the mailing list archives, if it is not already part of the official source tree. As an obligatory disclaimer: this should only be used with a CA certificate and private key that were legally created or obtained and it should carefully respect if the privacy policy of the individual environment allows to intercept encrypted HTTPS/SSL connections.

The first step is to generate a local CA key and certificate that will be used for intercepted SSL connections. The resulting ca.crt file includes the public CA certificate in Privacy Enhanced Mail (PEM) format that should be installed on the clients - the web browsers or SSL keychains. Most systems support importing the CA certificate by simply opening it in the browser from a URL or file.

```
# openssl req -x509 -days 365 -newkey
rsa:2048 -keyout /etc/ssl/private/ca.key
-out /etc/ssl/ca.crt
```

The relay SSL server also needs a private key and certificate for the 127.0.0.1 address. relayd will attempt to look up a private key in /etc/ssl/private/127.0.0.1:8443.key and a public certificate in /etc/ssl/127.0.0.1:8443.crt. If these files are not present, it will continue to look for /etc/ssl/private/127.0.0.1.key and /etc/ssl/127.0.0.1.crt without the port specification. The certificate will not be used in interception mode, as it will be replaced on-the-fly, but is required to start the SSL service.

The next step is to configure an appropriate line for PF in /etc/pf.conf that will divert any HTTPS traffic that is coming from an internal interface to the local relay service. Please note "divert-to" method is the preferred way for redirecting traffic to a local service and should be used instead of the older "rdr-to" whenever applicable.

**PF configuration** in /etc/pf.conf:

```
# Divert incoming HTTPS traffic to relayd
pass in on vlan1 inet proto tcp to port 443\
    divert-to localhost port 8443
```

The final step is to configure and run relayd . The relayd.conf configuration file must include two new "ssl ca" directives that specify the CA key with a mandatory password and the related CA certificate.

**relayd configuration** in /etc/relayd.conf:

```
http protocol httpfilter {
        return error

        label "Get back to work!"
        request url filter "facebook.com/"

# New directives for SSL Interception
        ssl ca key "/etc/ssl/private/ca.key"
password "humppa"
        ssl ca cert "/etc/ssl/ca.crt"
}

relay sslmitm {
        listen on 127.0.0.1 port 8443 ssl
        protocol httpfilter
        forward with ssl to destination
}
```

When testing the SSL Interception, clients can recognize that all server certificates are signed by the same local CA.

## 3 High-speed with Socket Splicing

In a traditional configuration, an application layer relay accepts and terminates a connection from a client, opens a second connection to a selected server and passes - or "relays" - all data between client and server with the additional ability to filter content and headers of the OSI Layer 7 (L7) protocol. All the data of the received packets have to be send from the kernel to the relay process in user space, copied to the other socket and copied back to the kernel that sends out data to the other side. In contrast to normal forwarding or routing that is completely done on the IP-level in the kernel, this introduces a significant performance penalty.

Socket splicing is a kernel-supported feature that can significantly improve the performance of TCP or User Datagram Protocol (UDP) relaying. It was implemented by Alexander Bluhm (bluhm@openbsd.org) in early 2011 and first available in the OpenBSD 5.0 release. The idea behind socket splicing is that the kernel can be instructed to directly forward any data between two opened sockets without copying it to a user space process. Just passing the stream of socket buffer data from one socket to another in the kernel can gain a performance that is almost comparable to pure OSI Layer 3 (L3) routing, especially when using an effective "zero-copy" in the kernel.

The socket splicing implementation for OpenBSD allows to connect two socket file descriptors. The SO_SPLICE socket option can instruct the kernel to pass any subsequent data directly between the specified socket without sending anything to the user space process:

```
if (setsockopt(fd1, SOL_SOCKET,
    SO_SPLICE, &fd2, sizeof(int)) == -1)
        return (-1);
```

The initial implementation only allowed to enable socket splicing for a given connection once and for an unspecified amount of data. This only worked for plain TCP connections without any content inspection by the user space process or with simple application protocols with single initial headers. In the HTTP case, this would allow to inspect a single request or response header in user space and switch to splicing for the remaining content body. This obviously fails with connections that include multiple requests and responses, like "HTTP Keep-Alive" or "Chunked Encoding" in HTTP 1.1.

As an extension, an optional alternative form of the socket option call allows to pass a maximum content length and timeout to the kernel, before it returns to normal operation and continues to pass any subsequent data to the user space process again:

```
bzero(&sp, sizeof(sp));
sp.sp_fd = fd2;
sp.sp_max = content_length;
sp.sp_idle = timeout;
if (setsockopt(fd1, SOL_SOCKET,
    SO_SPLICE, &sp, sizeof(sp)) == -1)
        return (-1);
```

This allows to handle mixed control/data or header/body streams. It was only partially adopted for `relayd` until recently in 5.3-current and the daemon now fully supports splicing of persistent HTTP connections with normal or chunked encoding transfers.

## 3.1 Zero configuration zero copy

The configuration of splicing in `relayd.conf` is very simple: it is enabled by default and does not need any additional configuration options. `relayd` will use splicing for all supported scenarios unless it is explicitly disabled with the `tcp no splice` configuration directive in the relay's protocol section.

# 4 Announcing the Filter Rewrite

The current implementation for relaying HTTP(S) connections provides a yet powerful filtering API that allows to operate on HTTP headers, cookies and URL elements. It can be used for various actions supporting the HTTP relay, including modification of headers or URL filtering. For example, a load balancer might inject an `X-Forwarded-For` header that includes the original client address or a transparent proxy might use URL filter lists to restrict access to public web pages.

The current implementation has been proven and works well with "huge" filter lists; `relayd` supports millions of URLs in the filter. Nevertheless, the filtering grammar and API lack some desired features that would be hard to implement in the current way. It is built around an HTTP-centric view that does not allow to filter on other parameters of the connection and it does not allow more advanced actions, like target host selection based on an URL path.

The new implementation will replace the current one by introducing a completely new grammar and by replacing the internal code of the filtering implementation. It follows the following design decisions:

- Use a new grammar that is loosely based on PF's style.
- Provide a more flexible interface that can also filter on TCP/IP options and other protocol headers like DNS.
- Improve and simplify the internal implementation and "filter hooks".
- Turn `relayd` into an ALG.

## 4.1 Filter Hooks

The original implementation tried to filter and modify the headers inline directly when the data stream is red. It was also based on a red-black (RB) tree to look up headers and values quickly. This worked well with the initial implementation but added problems after adding additional features that could not always allow inline modification of the header stream or would not work with the RB tree. The code was modified to handle all kinds of special cases by adding many different "HOOKs" and by extending the RB tree with attached linked lists.

### Old filtering HOOKs

Each HOOK calls the filtering subsystem to look up certain attributes in the tree and its connected lists. It ended up with four different places that independently use their HOOKs based on fairly complicated states in `relayd` 's internal HTTP session handling. The efficiency of the inline modifications might improve the performance of the header handling and lower the initial connection latency but the complexity of the code has grown in a way that introduced a potentially critical and hardly maintainable complexity.

1. per-line header filter
   a) gather information
      - path → HOOK
      - url → HOOK
      - http header → HOOK

   b) modify/delete headers
   c) action
2. after headers: resolve information
   a) append/check headers → HOOK
   b) action

### New filtering HOOKs

The new filtering implementation replaces the RB tree and attached linked lists of the filter rules with a linked list of last matching filtering rules. Additional tree-based lookups will only be used for large lists of similar items, like URL filter lists. The implementation is heavily inspired by PF's internals and is using its "skip step" algorithm to speed up the otherwise linear scan through the list. When loading the rules, `relayd` calculates skip steps to determine the next applicable rule for a possible input. For example, if a connection is TCP and the next 100 hundred rules only match for UDP, the "skip steps" would pre-calculate a pointer to the next rule that could possibly match and the filter would not have to evaluate these 100 rules for the current connection. Support for inline modifications of the header stream has been removed and replaced with a simple approach that is much more flexible. The relay code will always gather information about the connection and collect all headers and related information in a meta header (or "descriptor") first. It only uses one HOOK into the filter engine that handles all supported lookup types in the single list of filter rules. Any modifications to the headers are applied to the meta header before the final forwarding decision is done (block, pass or alternative target selection) and a newly constructed header is written to the output stream.

1. gather information
   a) read input header
   b) create meta header ("descriptor")
2. resolve information
   a) scan/modify meta header ⇒ HOOK
   b) action
   c) create output header

## 4.2 Grammar

### Old HTTP-centric view

```
+--------+ request     relayd      +--------+
|        |------------------------>|        |
| client | response                | server |
|        |<------------------------|        |
+--------+                         +--------+
```

As the filter language was implemented to handle HTTP filtering, it was built around HTTP features. The request/response-based protocol with an almost consequent use of key/value pairs and a strict separation between client and server is hard to adopt to some other more bidirectional and stream-oriented protocols. Additionally, the original language does not have any obvious matching order because the implementation uses a tree-based "best match" of the rules. And last, this approach made it almost impossible to add any advanced actions to the session handling, for example the possibility to select an alternative forwarding target based on various filtering criteria.

```
header append "$REMOTE_ADDR" \
    to "X-Forwarded-For"
header append "$SERVER_ADDR:$SERVER_PORT" \
    to "X-Forwarded-By"
header change "Connection" to "close"

# Block disallowed sites
label "URL filtered!"
request url filter "www.example.com/"

# Block disallowed browsers
label "Please try a <em>different
Browser</em>"
header filter \
    "Mozilla/4.0 (compatible; MSIE *" \
    from "User-Agent"

# Block some well-known Instant Messengers
label "Instant messenger disallowed!"
response header \
    filter "application/x-msn-messenger" \
    from "Content-Type"
response header filter "application/x-icq" \
    from "Content-Type"
response header filter "AIM/HTTP" \
    from "Content-Type"
```

### New PF-style filter language

```
+--------+    in +--------+ out    +--------+
|        |------>|        |------->|        |
| client |req out| relayd | resp in| server |
|        |<------|        |<-------|        |
+--------+       +--------+        +--------+
```

The new grammar uses some well-known keywords from PF including `block`, `pass` and `match`. The language introduces last matching rules that will be evaluated in linear order. The `match` keyword allows to add sticky actions to matching input but does not alter the decision of subsequent `block` or `pass` rules if the connection will be rejected and closed. The new implementation also adds support for filtering based on TCP/IP information (addresses, protocols, ports etc.) and is extensible for any other application protocol. A much desired feature is finally available: the `relay-to` directive and the ability to select a forwarding target based on filter matches. Any supported filtering option of a `match` or `pass` rule can be used to alter the default forwarding target. For example, this could be an URL path to send all requests for the "/images" directory to a different server or a source IP address that will get to a different backend.

```
match request header \
    append "X-Forwarded-For" \
    value "$REMOTE_ADDR"
match request header \
    append "X-Forwarded-By" \
    value "$REMOTE_ADDR:$SERVER_PORT"
match request header \
    set "Connect" value "close"

block client in url "www.example.com/" tag
"URL filtered!"
pass client in from 10.0.0.1 url
"www.example.com/"

match request header "User-Agent" \
    tag "Please try a <em>different
Browser</em>"
block request header "User-Agent" \
    value "Mozilla/4.0 (compatible; MSIE *"

block response header "Content-Type" value {
"application/x-msn-messenger"
"application/x-icq"
"AIM/HTTP"
}

match request path "/images" \
    relay-to 10.1.1.1
```

The language design and the implementation of the new filters have not been finished yet but can be followed in a separated GIT repository[1] outside of the OpenBSD tree. It will be merged into OpenBSD at a later point and finished in the tree.

# 5 File descriptor accounting

Sebastian Benoit (benno@openbsd.org) implemented a carefully designed file descriptor accounting mechanism that helps `relayd` to prevent file descriptor exhaustion problems. It is built as a wrapper around the `accept()` call and will additionally check if a) the number of opened file descriptors plus b) the number of statically reserved file descriptors and c) the number of "in-flight" sessions that aim to open a new file descriptor do not exceed the number of available file descriptors. The implementation will defer accepting new connections until enough file descriptors are available again. The number of statically reserved file descriptors will respect any potential file descriptors that might be required by library routines, including `libc` calls.

# 6 Appendix

## 6.1 About the Author

Reyk Floeter[2] works as a freelance consultant and software developer with a focus on OpenBSD, networking, and security. He lives in Hannover, Germany, but works with international customers like Internet Initiative Japan Inc. (IIJ) in Tokyo. As a member of the OpenBSD project, he contributed various features, fixes, networking drivers and daemons since 2004, like OpenBSD's ath, trunk, vic, hostapd, relayd, snmpd, and iked. For more than nine years and until mid-2011, he was the CTO & Co-Founder of .vantronix where he gained experience in building, selling and deploying enterprise-class network security appliances based on OpenBSD.

# References

[1] Reyk Floeter, *relayd filter development branch*, `https://github.com/reyk/relayd/tree/filter`.

[2] ———, *Reyk Floeter Consulting*, `http://www.reykfloeter.com/`.

[3] The H, *Fatal error leads TURKTRUST to issue dangerous SSL certificates*, `http://www.h-online.com/security/news/item/Fatal-error-leads-TURKTRUST-to-issue-dangerous-SSL-certificates-1777291.html`.

[4] Check Point Software Technologies Ltd., *HTTPS Inspection FAQ*, `https://supportcenter.checkpoint.com/supportcenter/portal?solutionid=sk65123`.

[5] Juniper Networks, *Inspection of SSL Traffic Overview*, `http://www.juniper.net/techpubs/en_US/idp5.0/topics/concept/intrusion-detection-prevention-ssl-decryption-overview.html`.

[6] OpenBSD, *Mailing Lists*, `http://www.openbsd.org/mail.html`.

[7] ———, *OpenBSD relayd*, `http://www.openbsd.org/cgi-bin/cvsweb/src/usr.sbin/relayd/`.

[8] Michal Trojnara, *The stunnel Program*, `http://www.stunnel.org`.