Flowreplay Design Notes

Aaron Turner
http://synfin.net/

Last Edited:
October 23, 2003

1 Overview

Tcpreplay[1] was designed to replay traffic previously captured in the pcap format back onto the wire for testing NIDS and other passive devices. Over time, it was enhanced to be able to test in-line network devices. However, a re-occurring feature request for tcpreplay is to connect to a server in order to test applications and host TCP/IP stacks. It was determined early on, that adding this feature to tcpreplay was far too complex, so I decided to create a new tool specifically designed for this.

Flowreplay is designed to replay traffic at Layer 4 or 7 depending on the protocol rather then at Layer 2 like tcpreplay does. This allows flowreplay to connect to one or more servers using a pcap savefile as the basis of the connections. Hence, flowreplay allows the testing of applications running on real servers rather then passive devices.

2 Features

2.1 Requirements

1. Full TCP/IP support, including IP fragments and TCP stream reassembly.
2. Support replaying TCP and UDP flows.
3. Code should handle each flow/service independently.
4. Should be able to connect to the server(s) in the pcap file or to a user specified IP address.
5. Support a plug-in architecture to allow adding application layer intelligence.
6. Plug-ins must be able to support multi-flow protocols like FTP.
7. Ship with a default plug-in which will work "well enough" for simple single-flow protocols like HTTP and telnet.
8. Flows being replayed "correctly" is more important then performance (Mbps).
9. Portable to run on common flavors of Unix and Unix-like systems.

2.2 Wishes

1. Support clients connecting to flowreplay on a limited basis. Flowreplay would replay the server side of the connection.
2. Support other IP based traffic (ICMP, VRRP, OSPF, etc) via plug-ins.
3. Support non-IP traffic (ARP, STP, CDP, etc) via plug-ins.
4. Limit which flows are replayed using user defined filters. (bpf filter syntax?)
5. Process pcap files directly with no intermediary file conversions.
6. Should be able to scale to pcap files in the 100's of MB in size and 100+ simultaneous flows on a P3 500MHz w/ 256MB of RAM.

3 Design Thoughts

3.1 Sending and Receiving traffic

Flowreplay must be able to process multiple connections to one or more devices. There are two options:

1. Use sockets[2] to send and receive data
2. Use libpcap[3] to receive packets and libnet[4] to send packets

Although using libpcap/libnet would allow more simultaneous connections and greater flexibility, there would be a very high complexity cost associated with it. With that in mind, I've decided to use sockets to send and receive data.

[1] http://tcpreplay.sourceforge.net/
[2] socket(2)
[3] http://www.tcpdump.org/
[4] http://www.packetfactory.net/projects/libnet/

3.2 Handling Multiple Connections

Because a pcap file can contain multiple simultaneous flows, we need to be able to support that too. The biggest problem with this is reading packet data in a different order then stored in the pcap file.

Reading and writing to multiple sockets is easy with select() or poll(), however a pcap file has it's data stored serially, but we need to access it randomly. There are a number of possible solutions for this such as caching packets in RAM where they can be accessed more randomly, creating an index of the packets in the pcap file, or converting the pcap file to another format altogether. Alternatively, I've started looking at libpcapnav[5] as an alternate means to navigate a pcap file and process packets out of order.

3.3 Data Synchronization

Knowing when to start sending client traffic in response to the server will be "tricky". Without understanding the actual protocol involved, probably the best general solution is waiting for a given period of time after no more data from the server has been received. Not sure what to do if the client traffic doesn't elicit a response from the server (implement some kind of timeout?). This will be the basis for the default plug-in.

3.4 TCP/IP

Dealing with IP fragmentation and TCP stream reassembly will be another really complex problem. We're basically talking about implementing a significant portion of a TCP/IP stack. One thought is to use libnids[6] which basically implements a Linux 2.0.37 TCP/IP stack in user-space. Other solutions include porting a TCP/IP stack from Open/Net/FreeBSD or writing our own custom stack from scratch.

4 Multiple Independent Flows

The biggest asynchronous problem, that pcap files are serial, has to be solved in a scaleable manner. Not much can be assumed about the network traffic contained in a pcap savefile other then Murphy's Law will be in effect. This means we'll have to deal with:

- Thousands of small simultaneous flows (captured on a busy network)
- Flows which "hang" mid-stream (an exploit against a server causes it to crash)
- Flows which contain large quantities of data (FTP transfers of ISO's for example)

How we implement parallel processing of the pcap savefile will dramatically effect how well we can scale. A few considerations:

- Most Unix systems limit the maximum number of open file descriptors a single process can have. Generally speaking this shouldn't be a problem except for highly parallel pcap's.
- While RAM isn't limitless, we can use mmap() to get around this.
- Many Unix systems have enhanced solutions to poll() which will improve flow management.

4.1 IP Fragments and TCP Streams

There are five major complications with flowreplay:

1. The IP datagrams may be fragmented- we won't be able to use the standard 5-tuple (src/dst IP, src/dst port, protocol) to lookup which flow a packet belongs to.
2. IP fragments may arrive out of order which will complicate ordering of data to be sent.
3. The TCP segments may arrive out of order which will complicate ordering of data to be sent.
4. Packets may be missing in the pcap file because they were dropped during capture.

[5] http://netdude.sourceforge.net/
[6] http://www.avet.com.pl/~nergal/libnids/

5. There are tools like fragrouter which intentionally create non-deterministic situations.

First off, I've decided, that I'm not going to worry about fragrouter or it's cousins. I'll handle non-deterministic situations one and only one way, so that the way flowreplay handles the traffic will be deterministic. Perhaps, I'll make it easy for others to write a plug-in which will change it, but that's not something I'm going to concern myself with now.

Missing packets in the pcap file will probably make that flow unplayable. There are proabably certain situation where we can make an educated guess, but this is far too complex to worry about for the first stable release.

That still leaves creating a basic TCP/IP stack in user space. The good news it that there is already a library which does this called libnids. As of version 1.17, libnids can process packets from a pcap savefile (it's not documented in the man page, but the code is there).

A potential problem with libnids though is that it has to maintain it's own state/cache system. This not only means additional overhead, but jumping around in the pcap file as I'm planning on doing to handle multiple simultaneous flows is likely to really confuse libnids' state engine. Also, libnids is licensed under the GPL, but I want flowreplay released under a BSD-like license; I need to research if the two are compatible in this way.

Possible solutions:

- Developing a custom wedge between the capture file and libnids which will cause each packet to only be processed a single time.
- Use libnids to process the pcap file into a new flow-based format, effectively putting the TCP/IP stack into a dedicated utility.
- Develop a custom user-space TCP/IP stack, perhaps based on a BSD TCP/IP stack, much like libnids is based on Linux 2.0.37.
- Screw it and say that IP fragmentation and out of order IP packets/TCP segments are not supported. Not sure if this will meet the needs of potential users.

4.2 Blocking

As earlier stated, one of the main goals of this project is to keep things single threaded to make coding plugins easier. One caveat of that is that any function which blocks will cause serious problems.

There are three major cases where blocking is likely to occur:

1. Opening a socket
2. Reading from a socket
3. Writing to a socket

Reading from sockets in a non-blocking manner is easy to solve for using poll() or select(). Writing to a socket, or merely opening a TCP socket via connect() however requires a different method:

> It is possible to do non-blocking IO on sockets by setting the O_NONBLOCK flag on a socket file descriptor using fcntl(2). Then all operations that would block will (usually) return with EAGAIN (operation should be retried later); connect(2) will return EINPROGRESS error. The user can then wait for various events via poll(2) or select(2).[7]

If connect() returns EINPROGRESS, then we'll just have to do something like this:

```
int e, len=sizeof(e);
if (getsockopt(conn->s, SOL_SOCKET, SO_ERROR, &e, &len) < 0) {
    /* not yet */
    if(errno != EINPROGRESS){  /* yuck. kill it. */
       log_fn(LOG_DEBUG,"in-progress connect failed. Removing.");
       return -1;
    } else {
       return 0; /* no change, see if next time is better */
    }
}
/* the connect has finished. */
```

> Note: It may not be totally right, but it works ok. (that chunk of code gets called after poll returns the socket as writable. if poll returns it as readable, then it's probably because of eof, connect fails. You must poll for both.

[7] socket(7)

5 pcap vs flow File Format

As stated before, the pcap file format really isn't well suited for flowreplay because it uses the raw packet as a container for data. Flowreplay however isn't interested in packets, it's interested in data streams[8] which may span one or more TCP/UDP segments, each comprised of an IP datagram which may be comprised of multiple IP fragments. Handling all this additional complexity requires a full TCP/IP stack in user space which would have additional feature requirements specific to flowreplay.

Rather then trying to do that, I've decided to create a pcap preprocessor for flowreplay called: flowprep. Flowprep will handle all the TCP/IP defragmentation/reassembly and write out a file containing the data streams for each flow.

A flow file will contain three sections:

1. A header which identifies this as a flowprep file and the file version
2. An index of all the flows contained in the file
3. The data streams themselves

8 Bits			
32 Bit Word			

Flowprep File Header

Magic Number	
Version	Reserved

Flow Index Entry

Client (Source) IP		
Server (Destination) IP		
IP Protocol	Flags	Instance
Client Port/ICMP Type	Server Port/ICMP Code	
Offset to First Data Stream		

Flag 1: Last Index
Flag 2: Ignore
Flag 3: Server Socket

Data Stream Header

Data Length of This Stream		
Flags	Urg Data	Reserved
Timestamp		
Offset to Next Data Segment In This Flow		
Data Stream		

Flag 1: Direction
Flag 2: Ignore
Flag 3: More Data Streams
Flag 4: Urgent Data Exists

At startup, the file header is validated and the data stream indexes are loaded into memory. Then the first data stream header from each flow is read. Then each flow and subsequent data stream is processed based upon the timestamps and plug-ins.

[8] A "data stream" as I call it is a simplex communication from the client or server which is a complete query, response or message.

6 Plug-ins

Plug-ins will provide the "intelligence" in flowreplay. Flowreplay is designed to be a mere framework for connecting captured flows in a flow file with socket file handles. How data is processed and what should be done with it will be done via plug-ins.

Plug-ins will allow proper handling of a variety of protocols while hopefully keeping things simple. Another part of the consideration will be making it easy for others to contribute to flowreplay. I don't want to have to write all the protocol logic myself.

6.1 Plug-in Basics

Each plug-in provides the logic for handling one or more services. The main purpose of a plug-in is to decide when flowreplay should send data via one or more sockets. The plug-in can use any *non-blocking* method of determining if it appropriate to send data or wait for data to received. If necessary, a plug-in can also modify the data sent.

Each time poll() returns, flowreplay calls the plug-ins for the flows which either have data waiting or in the case of a timeout, those flows which timed out. Afterwords, all the flows are processed and poll() is called on those flows which have their state set to POLL. And the process repeats until there are no more nodes in the tree.

6.2 The Default Plug-in

Initially, flowreplay will ship with one basic plug-in called "default". Any flow which doesn't have a specific plug-in defined, will use default. The goal of the default plug-in is to work "good enough" for a majority of single-flow protocols such as SMTP, HTTP, and Telnet. Protocols which use encryption (SSL, SSH, etc) or multiple flows (FTP, RPC, etc) will never work with the default plug-in. Furthermore, the default plug-in will only support connections *to* a server, it will not support accepting connections from clients.

The default plug-in will provide no data level manipulation and only a simple method for detecting when it is time to send data to the server. Detecting when to send data will be done by a "no more data" timeout value. Basically, by using the pcap file as a means to determine the order of the exchange, anytime it is the servers turn to send data, flowreplay will wait for the first byte of data and then start the "no more data" timer. Every time more data is received, the timer is reset. If the timer reaches zero, then flowreplay sends the next portion of the client side of the connection. This is repeated until the the flow has been completely replayed or a "server hung" timeout is reached. The server hung timeout is used to detect a server which crashed and never starts sending any data which would start the "no more data" timer.

Both the "no more data" and "server hung" timers will be user defined values and global to all flows using the default plug-in.

6.3 Plug-in Details

Each plug-in will be comprised of the following:

1. An optional global data structure, for intra-flow communication
2. Per-flow data structure, for tracking flow state information
3. A list of functions which flow replay will call when certain well-defined conditions are met.
   - Required functions:
     * initialize_node() - called when a node in the tree created using this plug-in
     * post_poll_timeout() - called when the poll() returned due to a timeout for this node
     * post_poll_read() - called when the poll() returned due to the socket being ready
     * buffer_full() - called when a the packet buffer for this flow is full
     * delete_node() - called just prior to the node being free()'d
   - Optional functions:
     * pre_send_data() - called before data is sent
     * post_send_data() - called after data is sent
     * pre_poll() - called prior to poll()
     * post_poll_default() - called when poll() returns and neither the socket was ready or the node timed out
     * open_socket() - called after the socket is opened
     * close_socket() - called after the socket is closed