

## Review of *flop*- how it's signatures work to fingerprint Applications

by Robert Floodeen AKA Hol

*flop* was designed by Michael Zalewski (<http://lcamtuf.coredump.cx/soft/flop-devel.tgz>) on the successful product *p0f*, which is a Passive Asset Identification tool. The intent of *flop* is to passively identify applications as they traverse the network. A quick comparison of *flop.c* to *p0f.c* shows 179 differences over the 1173 lines of code. A visual review of the two *.c* files shows that *flop.c* is indeed a pared down version of *p0f* with some additions.

*flop* looks at the exchange of packets with a payload within two end points. The packets between these two end points are classified as a session. *flop* does packet inspection, however it does not do protocol analysis in layers 5,6, or 7; it simply counts the bytes. While this may seem very similar to what is produced with flow data (like in *argus*) it is not. Flow data also counts the bytes in the headers (layer 2 – 4) and aggregates all the bytes together at the end of a session.

While *flop* has not had any project growth since 2006, the concepts behind the algorithm it uses to identify applications and the algorithm it uses for signatures is still current. For example, *flop* can identify different activities in the establishment of an *ssh* connection or determine if an established 443 connection is browser activity or an encrypted tunnel doing something not *https* related.

To start we will look at the structure and intent for *flop*'s signatures. Signatures are loaded from the configuration file into the variable *sig*, which is an array of the *struct signature* type defined in *flop.h* as seen in **Figure 1**.

```
/* Single signature entry: */
struct signature {
    _u8* name;           /* Traffic description (NULL for silent) */
    _u8 proto;          /* Protocol ID (IPPROTO_*) */
    _u16 port;          /* Destination port (0 - any) */
    _u8 siglen;         /* Signature element count */
    _u8 f_check_mode:4, /* Flag checking mode (SIG_FLAG_*) */
        f_set_mode:4;  /* Flag setting mode (SIG_FLAG_*) */
    _u32 f_check,       /* Which flags to check */
        f_set;         /* Which flags to set */
    struct sig_item
        el[MAXSIGLEN+1]; /* Signature entry */
    _u8 need_open,      /* Session init required (OPEN) */
        need_close,    /* Session termination required (CLOSE) */
        no_fuzzy;      /* Fuzzy matching off (EXACT) */
};
```

**Figure 1 – Signature Structure in *flop.h***

*flop*'s *config.h* is used to define the traditional “configuration file” items, and acts as the configuration file, while pointing to the fingerprint database files (*flop.fp* in this case). This is also inherited from *p0f*. While reviewing this configuration file we can see that there is a suggested limit to a maximum number of signatures. This is defaulted to 200. Currently (as of 2006, latest release) there are 29 signatures.

A signature consists of the following items observed in a network connection:

- The Protocol used
- The Direction - the system originating the packet, was it from the client, or from the server
- The Layer 5,6, and 7 payload size in each packet
- Delay – between packets, generally larger delays are indicative of asynchronous events, user input, or extensive input/output operations
- Session Boundaries – when one “use” of a connection ends and another begins, however the session itself might still be open

While it is possible to build a signature from hand, it is suggested that the traffic be captured and ran through *flOp* with the “-u” option. This will create a fingerprint with the following characteristics:

- “c” means it came from the Client
- “s” means it came from the Server
- “+” means a delay of more than the specific time unit was exceeded. This is 500ms by default and set in the *config.h* file

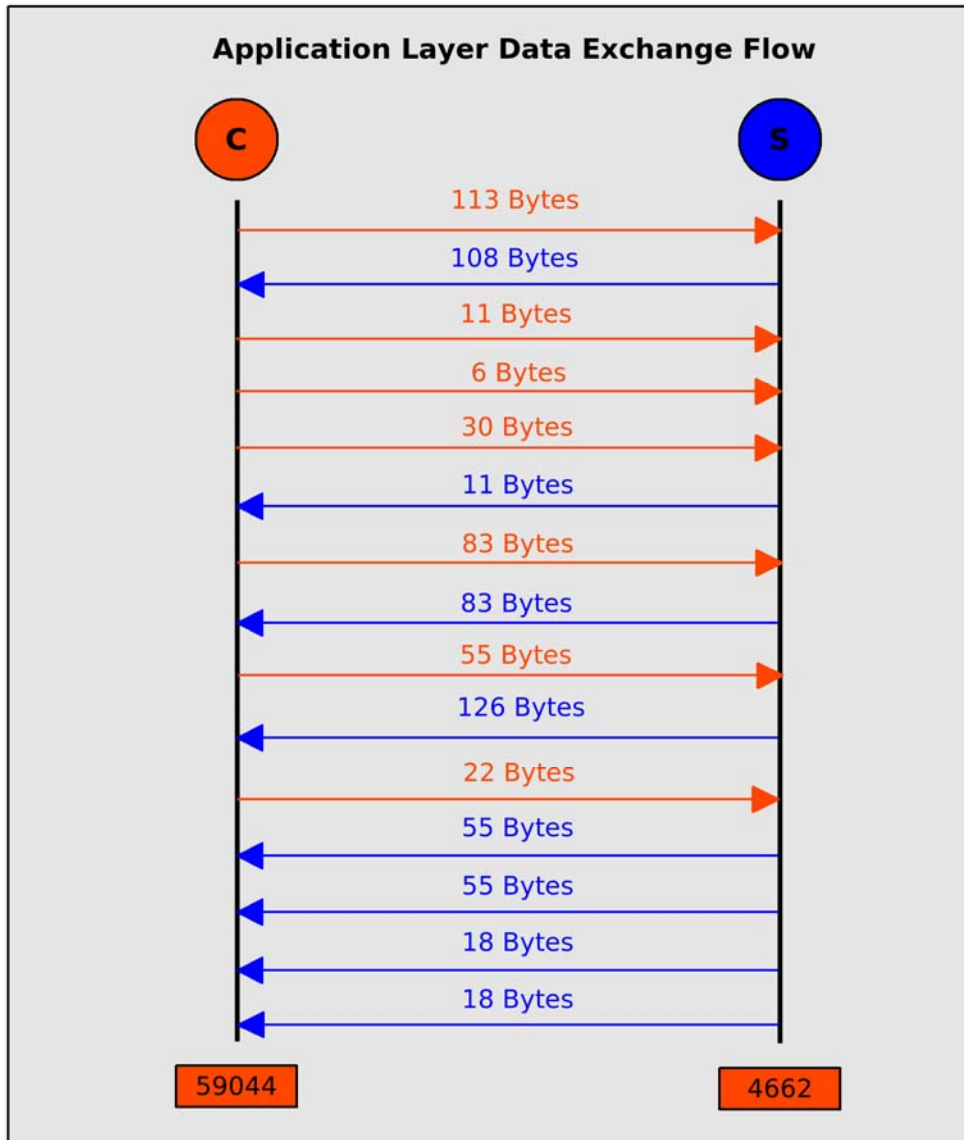
From this, a user can then create a fingerprint with elements from the *flOp.fp* file. However, there are a few options remaining that are useful to the final signature creation. The first is the ability to declare a range of packet sizes. “s27/15” would indicate that a packet from the Server is sent with a size from 12 to 43 bytes. The second is the ability to declare the start or end of a session by using “<” for start and “>” for end. These do not have to be used together. Third is the ability to declare an unknown location but an exact pattern of packet exchanges between the client and server. This is referred to as “back and forward references in packet sizes”. This is similar to the ability in a Regular Expression to identify a match on something that is not yet known. This works well in conjunction with the ability to declare a range of packet sizes. For example, we don’t know exactly how big the first packet will be, but once it has occurred, we know it’s size will not change. Therefore we can use this ability to match on the result of the first packet and ensure that the following packets are of the same size, not just in the same range. To accomplish this, the “@” is used. “Back and forward references” works as an offset from the first packet identified in the signature, it does not count other @’s, and it does not look at the original source of the packet being matched.

The default signature format as seen in **Figure 2**:

<code>proto ( port   * ) [ = ] [ &lt; ] signature [ &gt; ] : description</code>
---

**Figure 2 – Default Signature Format**

An example of a signature (taken from geek001's "Basic f10p Signature Writing Guide <http://www.rawpacket.org/anonymous/papers/F10p-Sigs-Writing.pdf> ) as seen in **Figure 3 – Signature Example**



```
tcp * = < c113/10 s108/10 c11/1 c6 c30/4 s@3 c83/10 s@7 c55/8 s126/10 c22/3 s@9 s@9 s18/2 s@14
```

**Figure 3 – Signature Example**

The main function used to compare the captured packets to a signature, is contained in *f10p.c* in the function *static void do\_matching(struct, \_u8)*. We reprint the function below and break it into 3 pieces with comments, the initial checks, the checking of elements in the signature, and the cleanup.

```
static void do_matching(struct f10p_stream *f, _u8 final) {
```

```

_u32 i, a, gotone = 0;

/* Try normal */

for (i=0;i<sigcnt;i++) {

    if (f->proto != sig[i].proto) continue;
    if (sig[i].port && sig[i].port != f->sport) continue;
    if (sig[i].need_open && f->pcount != sig[i].siglen) continue;
    if (sig[i].need_close && !final) continue;

    if (sig[i].f_check_mode == SIG_FLAG_SET) {
        if ((f->flag & sig[i].f_check) != sig[i].f_check) continue;
    } else if (sig[i].f_check_mode == SIG_FLAG_ZERO)
        if ((f->flag & sig[i].f_check)) continue;
}

```

**Figure 4 – Initial Set Up, *fl0p.c***

A “for loop” is created which will attempt to match every signature in the signature db to a single item in the *fl0p\_stream struct*. The *fl0p\_stream struct* is preloaded prior to this function being called. Up to this point we are inside the initial “for loop”. The items which are tested first are ones with clear and definitive answers. These are the protocol (tcp/udp/icmp), the server port (1 – 65535), if we are testing from the beginning of a connection “<” and the packets processed do not match the current signature element count, or if we need a connection closed “>” yet there are more packets to process.

```

for (a=0;a<sig[i].siglen;a++) {
    _s32 st = 0, en = 0;
    _u32 tpos = MAXSIGLEN * 2 - sig[i].siglen;

    if (f->tail[tpos + a].type != sig[i].el[a].type) break;
    if (sig[i].el[a].d_prev != ITEM_D_NONE)
        if (f->tail[tpos + a].d_prev != sig[i].el[a].d_prev) break;

    if (sig[i].el[a].valtype == ITEM_VAL_ANY) continue;

    if (sig[i].el[a].valtype == ITEM_VAL_NORM)
        st = en = sig[i].el[a].value;
    else /* ITEM_VAL_REF */
        st = en = f->tail[tpos + sig[i].el[a].value].value;

    st -= sig[i].el[a].range;
    en += sig[i].el[a].range;

    if (f->tail[tpos + a].value < st || f->tail[tpos + a].value > en)
break;

}

```

**Figure 5 – testing elements of a signature ( c10 s12 + s@1 ... ), *fl0p.c***

If the packet makes it past the first set of tests, it then goes into a “nested for loop”, which tests against the standard elements in the signature (items after the “=” and before the “:” in **Figure 3 – Signature Example**)

```

if (a != sig[i].siglen) continue;

if (sig[i].name) {
    f->matched++;
    report_match(f, sig+i, 1);
    gotone = 1;
} else {
    if (sig[i].f_set_mode == SIG_FLAG_SET)
        f->flag |= sig[i].f_set;
    else if (sig[i].f_set_mode == SIG_FLAG_ZERO)
        f->flag &= ~sig[i].f_set;
}

}
if (no_fuzzy || gotone) return;

--- Edited remainder out which attempts fuzzy Matching---

}

```

**Figure 6 – Clean up, *flop.c***

The Clean up starts by the first line after the “nested for loop”. It checks to ensure that leaving the “nested for loop” has gone through all of the elements in the signature. It then tests to see if the signature was constructed properly and has a name associated with it. If this is the case, then a signature has been matched. It should be noted that at every step, if a signature is not matched to this session, then the current match is exited and the next one began.

Overall, it should also be pointed out that this algorithm would initially appear to be Big-Oh of  $O(n^3)$  based on  $i(jk)$  where

*i* == Number of data packets being tested  
*j* == Number of signatures in database  
*k* == Number of elements in a signature

We determined however that we will end up with just  $O(n)$  because the value of *k* will never grow large and *j* is contained to 200. So, as the number of packets move to infinity the time to process them will not significantly increase for any other reason than the fact that the number of packets are increasing.

In conclusion, we can see that *flop* has some powerful capabilities, but at the same time has not taken into account any for protection and can still be easily confused. It is also limited in the number of items it can watch for. All in all this is a great idea that is exactly where the author put it, beta. So my suggestion is to use it, but do so with a very specific purpose and as an added layer.

## Appendix A: Sample Output

A sample of output from *flop*: **Figure 7**.

```
(tcp) 213.195.140.12:4667 -> 213.134.128.25:25
  Observed for: 188B, 6 packets, spans 17 seconds
  Matches: Possible manual line-by-line interaction (hit: 1)
(tcp) 83.31.193.40:3403 -> 213.134.128.25:22
  Observed for: 584B, 9 packets, spans 5 seconds
  Matches: SSH1 - client manually accepted key (hit: 1)
(tcp) 83.31.193.40:3406 -> 213.134.128.25:22
  Observed for: 820B, 18 packets, spans 9 seconds
  Matches: SSH1 - invalid password attempt (hit: 2)
(tcp) 83.31.193.40:3436 -> 213.134.128.25:22
  Observed for: 2.9kB, 19 packets, spans 2 seconds
  Matches: SSH2 - correct password (hit: 2)
```

**Figure 7 – Sample Output from Signature Matches**