Porting OpenBSD's pf syncookie Code to FreeBSD's pf

BY KRISTOF PROVOST

The internet can be a lawless place, and our on-line services can be attacked in a variety of ways. The firewalls we run can be part of the way we protect our systems, but it turns out they can also be an avenue for attack themselves.

One way of attacking services is to exhaust system resources by pretending to open connections. This is commonly called a SYN flood, and it's a fairly insidious attack. Let's start by refreshing our memories about how TCP connections work. Opening a TCP connection is a three-step process:

1) SYN: the client sends a SYN packets to indicate it wishes to open a connection. It sets an initial sequence number in the SYN packet.
2) SYN+ACK: the server responds, acknowledging the client sequence number, and settings its own.
3) ACK: the client accepts that the connection is open, and acknowledges the server's sequence number.

When the server receives the initial SYN it will set up internal data structures to support the new connection. This takes CPU time and memory.

This means that malicious clients can generate SYN packets (which are small, and easy to generate), consuming much of the server's available memory and CPU resources. Even worse, it only requires the single SYN packet. There's no need for the attacker to receive the server's SYN+ACK reply. That means that the source IP address can be faked, making these attacks difficult to filter out. They can also be targeted at the service's main TCP port (e.g., 443 for a web server), making the attack indistinguishable from real client requests.

SYN cookies

In 1996 Daniel J. Bernstein and Eric Schenk came up with a method to resist such attacks, called SYN cookies. Simply put, SYN cookies ensure the server does not run out of memory by not allocating any memory for the new connection when we receive a SYN packet.

We still generate a SYN+ACK reply, but wait to create server-side state until the client has responded to our SYN+ACK with its own ACK. This ensures that the client really exists (i.e., the source IP address is not spoofed).

The obvious issue with this is that we still need the information we'd normally save when the SYN first arrives. This information includes things like Maximum Segment Size (MSS) and Window Scale (WSCALE). While these options are … well optional, they are important for TCP performance, and we don't want to refuse them.

Furthermore, we also need some way of ensuring that the acknowledgement number in the client's ACK matches the sequence number we used in the SYN+ACK message. If we didn't check this, malicious clients could simply send a SYN, wait a little while and send an ACK blind, that is, without having to actually receive the server's SYN+ACK using a random acknowledgement number.

So, how do we accomplish this? We do so by encoding all the options into the sequence number of the SYN+ACK packet.

In the pf implementation, this is done in `pf_syncookie_generate()`:

```
uint32_t
pf_syncookie_generate(struct mbuf *m, int off, struct pf_pdesc *pd,
    uint16_t mss)
{
        uint8_t                 i, wscale;
        uint32_t                iss, hash;
        union pf_syncookie      cookie;

        PF_RULES_RASSERT();

        cookie.cookie = 0;

        /* map MSS */
        for (i = nitems(pf_syncookie_msstab) - 1;
            pf_syncookie_msstab[i] > mss && i > 0; i--)
                /* nada */;
        cookie.flags.mss_idx = i;

        /* map WSCALE */
        wscale = pf_get_wscale(m, off, pd->hdr.tcp.th_off, pd->af);
        for (i = nitems(pf_syncookie_wstab) - 1;
            pf_syncookie_wstab[i] > wscale && i > 0; i--)
                /* nada */;
        cookie.flags.wscale_idx = i;
        cookie.flags.sack_ok = 0;       /* XXX */

        cookie.flags.oddeven = V_pf_syncookie_status.oddeven;
        hash = pf_syncookie_mac(pd, cookie, ntohl(pd->hdr.tcp.th_seq));

        /*
         * Put the flags into the hash and XOR them to get better ISS number
         * variance.  This doesn't enhance the cryptographic strength and is
         * done to prevent the 8 cookie bits from showing up directly on the
         * wire.
         */
        iss = hash & ~0xff;
        iss |= cookie.cookie ^ (hash >> 24);

        return (iss);
}
```

The eagle-eyed reader will note that for both MSS and WSCALE we don't actually encode the correct value, but instead find the closest match in a lookup table. This reduces the number of bits needed to encode the information, but still gets us a good approximation of the real value. Having a slightly smaller maximum segment size or window scale will cost us a little performance, but not significantly so. The values are chosen so that the most frequently used MSS or WSCALE values are represented, so for most clients there will be no performance loss at all.

This information is encoded into an authenticated hash. That is, to recreate the hash you need both the input information (MSS, WSCALE, ...) and a secret key. In other words: attackers cannot predict the result of the hash, and consequently cannot predict the sequence number the server will choose. That's handled by the `pf_syncookie_mac()` function:

```
uint32_t
pf_syncookie_mac(struct pf_pdesc *pd, union pf_syncookie cookie, uint32_t seq)
{
        SIPHASH_CTX     ctx;
        uint32_t        siphash[2];

        PF_RULES_RASSERT();
        MPASS(pd->proto == IPPROTO_TCP);

        SipHash24_Init(&ctx);
        SipHash_SetKey(&ctx, V_pf_syncookie_status.key[cookie.flags.oddeven]);

switch (pd->af) {
case AF_INET:
        SipHash_Update(&ctx, pd->src, sizeof(pd->src->v4));
        SipHash_Update(&ctx, pd->dst, sizeof(pd->dst->v4));
        break;
case AF_INET6:
        SipHash_Update(&ctx, pd->src, sizeof(pd->src->v6));
        SipHash_Update(&ctx, pd->dst, sizeof(pd->dst->v6));
        break;
default:
        panic("unknown address family");
}

SipHash_Update(&ctx, pd->sport, sizeof(*pd->sport));
        SipHash_Update(&ctx, pd->dport, sizeof(*pd->dport));
        SipHash_Update(&ctx, &seq, sizeof(seq));
        SipHash_Update(&ctx, &cookie, sizeof(cookie));
        SipHash_Final((uint8_t *)&siphash, &ctx);

        return (siphash[0] ^ siphash[1]);
}
```

With the resulting hash post-processed we have enough information to send the server's SYN+ACK response.

At this point pf processing stops. We do not create state, we do not perform any further examination of the packet. This also means that if the firewall protects a different host (i.e., it's running on a router between the client and server) the server will not even be aware that the client has attempted to initiate a new connection. We want that because it means the server is protected from SYN floods, without needing any code or configuration changes.

If the client never responds, nothing happens. The server has remembered nothing about this specific SYN message and has no memory allocated to it. If on the other hand the client does respond (i.e., is a legitimate client, at least for the purpose of this discussion), we must reconstruct the information we've not retained when we received the original SYN message.

Upon receiving a SYN+ACK message we first validate it in `pf_syncookie_validate()`:

```
uint8_t
pf_syncookie_validate(struct pf_pdesc *pd)
{
        uint32_t                hash, ack, seq;
        union pf_syncookie          cookie;

        MPASS(pd->proto == IPPROTO_TCP);
        PF_RULES_RASSERT();

        seq = ntohl(pd->hdr.tcp.th_seq) - 1;
        ack = ntohl(pd->hdr.tcp.th_ack) - 1;
        cookie.cookie = (ack & 0xff) ^ (ack >> 24);

        /* we don't know oddeven before setting the cookie (union) */
        if (atomic_load_64(&V_pf_status.syncookies_inflight[cookie.flags.oddeven])
            == 0)
                return (0);

        hash = pf_syncookie_mac(pd, cookie, seq);
        if ((ack & ~0xff) != (hash & ~0xff))
                return (0);

        counter_u64_add(V_pf_status.lcounters[KLCNT_SYNCOOKIES_VALID], 1);
        atomic_add_64(&V_pf_status.syncookies_inflight[cookie.flags.oddeven], -1);

        return (1);
}
```

We check that the cookie contains the correct authentication string. If it does, we continue into `pf_syncookie_recreate()`, where we reconstruct the original SYN packet. This isn't strictly required for the syncookie system itself, but we need to tell pf about the SYN packet we'd originally discarded so it can create the relevant state entries.

This also allows pf to continue processing, and potentially forwards the reconstituted SYN packet to the remote server. The remote server would then reply with its own SYN+ACK packet, with a different sequence number from ours. pf will have to modify the sequence and ac-

knowledgement numbers on all traffic between client and server. Happily, this is standard functionality for pf.

At this point the connection is fully established on both sides, and it does not meaningfully differ from a connection set up without syncookies. No special action is taken on connection shutdown because this does not present new opportunities for a malicious client to generate memory pressure.

Downsides

So far, we've discussed how syncookies help us, but we've not spent much time any drawbacks. Does that mean that there are none? Sadly, no.

We've already talked about MSS and WSCALE. With syncookies we are unable to reflect the proposed value from the client with full fidelity. This may mean that for some clients we leave some TCP performance on the table. In most cases this is not something to worry about.

With syncookies we are unable to reflect the proposed value from the client with full fidelity.

Another downside is implicit in how syncookies work: we unconditionally reply SYN+ACK to the SYN packet. Even if the port is actually closed. That means that the client may think opening the connection is working, until it's fully established, only to receive an RST afterwards. That's not ideal and may provoke unexpected client behavior. That is, this may look different from a "normal" failure to connect to users.

This can be mostly mitigated by ensuring that the firewall immediately rejects packets to closed ports. That's generally a good idea anyway, and it goes doubly so if syncookies are enabled.

Another downside is that there's no retransmit mechanism for lost SYN+ACK packets. There couldn't be, because as soon as we send the SYN+ACK we forget everything about it. This isn't too much of a concern because the client will just assume its SYN packet got lost and retransmit that. That will lead the server to generate a new SYN+ACK, which will hopefully not get lost this time.

A final thing to bear in mind is that syncookies are not magic. They work well against SYN-flood attacks, but they cannot protect against other attacks. For example, if a specially crafted HTTP request consumes excessive system resources in the web server, this will not be stopped by syncookies.

There's also nothing to stop a motivated attacker from initiating connections from many different client IP addresses without spoofing the source address. In that case, the attacker can still potentially open enough connections to exhaust the server's resources. However, syncookies make this much more expensive for the attacker. A SYN flood can be performed from a single attacking host, with moderate bandwidth requirements. An attack that has the same effect using non-spoofed, TCP connections will require many more attacking hosts.

History

The FreeBSD pf syncookie code was adapted from the pf syncookie code in OpenBSD's pf. This code was originally written by Henning Brauer in 2018 with help from Alexandr Nedvedicky.

The OpenBSD pf syncookie code was based on syncookie code in FreeBSD's TCP stack, originally developed by Jonathan Lemon in 2001 (a9c96841638186f2e8d10962b80e8e9f683d0cbc).

It looks like the OpenBSD commit message is incorrect in its attribution to Andre Oppermann. Andre did make significant improvements to the syncookie code in 2013 (81d392a09de0f2eeabaf68787896863eb9c370a8), which is probably where the misunderstanding came from.

Implementation Notes

While OpenBSD and FreeBSD's pf versions have diverged a bit over the years the similarities still greatly outweigh the differences. As such, porting OpenBSD pf features to FreeBSD is often relatively straightforward. The main stumbling block is the different approaches in locking strategy. OpenBSD's pf, like OpenBSD's network stack, is protected by a single lock (NET_LOCK). This has the advantage of great simplicity but does come with some performance drawbacks.

FreeBSD's pf takes a much more complex approach to locking, but does get better performance in return.

This turned out to be relevant for the adaptive mode. Other aspects of the syncookie code fit neatly into the existing locking approach. However, OpenBSD's approach of incrementing and decrementing a single counter value to track the number of half-open states and in-flight syncookie packets. This required the use of atomic operations in FreeBSD because there's no equivalent NET_LOCK and multiple cores can be processing TCP SYN or other packets at the same time.

While OpenBSD and FreeBSD's pf versions have diverged a bit over the years the similarities still greatly outweigh the differences.

While this ensures we do not under or over count the number of half-open states or in-flight syncookie packets, it is still imperfect. The retrieval of the values is atomic, but as we retrieve multiple values, they do not always reflect a perfect snapshot. Happily, there is no requirement for strict correctness here. The worst case is that we enable or disable syncookies slightly early or late. As both syncookie-mediated and normal connections can be established at the same time, this is not a noticeable concern for users.

Configuration

After all of that, readers could be forgiven for assuming that the configuration of syncookies is a complicated affair, but this is not the case. There's only one required line, in the options section of pf.conf:

```
set syncookies always
```

or

```
set syncookies adaptive
```

The first will always respond to SYN packets with a syncookie SYN+ACK. In adaptive mode pf will only do so when a lot of connections are in half-open state. That is, we've replied SYN-+ACK to an ACK message and are waiting for the ACK in response. This ideally combines the best of both worlds: we get all the advantages of normal TCP connection processing (i.e., full-option negotiation, immediate feedback when the connection cannot be opened) but with some protection against SYN floods.

Adaptive mode, low and high water marks (i.e. where we disable and enable syncookies respectively) can be configured as well:

```
set syncookies adaptive (start 25%, end 12%)
```

The values are expressed as percentages of the state table size. If no syncookie configuration line is present the feature will default to being disabled. This means there is no change in behavior unless users explicitly enable syncookies.

Conclusion

Are syncookies right for you? They may be if your systems are attacked by SYN floods. If they are not you may want to leave them disabled, but even so, it's good to know they exist. SYN floods are a very old type of attack, but as long as they work, even occasionally, attackers may decide to use them. Defenders must be ready with appropriate tools.

The new pf syncookie feature is already present in the recent 12.3 release, and will also be present in the upcoming 13.1 release.

The effort to port the OpenBSD pf syncookie code to FreeBSD's pf was sponsored by Modirum MDPay.

KRISTOF PROVOST is a freelance embedded software engineer specializing in network and video applications. He's a FreeBSD committer, maintainer of the pf firewall in FreeBSD and a board member of the EuroBSDCon foundation. Kristof has an unfortunate tendency to stumble into uClibc bugs, and a burning hatred for FTP. Do not talk to him about IPv6 fragmentation.