The surprising complexity of TCP/IP checksums in the network stack

Henning Brauer
BS Web Services

Abstract

TCP and IP have well known and well understood checksum mechanisms. The actual checksum math is easy and, from a performance standpoint, so cheap that it can be considered free. In the process of improving the use of hardware checksum offloading engines, recalculating the IP checksum has been found to be essentialy free. However, that is not the case for the TCP and UDP checksums, which was partially expected. On further inspection a surprising complexity in dealing with the protocol checksums has been found.

We'll look at how these checksums are calculated, where the complexity comes from, how an ancient BSD performance hack made it into hardware offloading engines, the stack interaction and issues with hardware offloading engines.

1 Introduction

For a long time I had been annoyed by the checksum handling in pf. pf used to fix up checksums on the fly, as in, whenever it modified a packet - which it would do for all forms of NAT, for example - it adjusted the original checksum for the applied word delta. This is not only annoying and error prone in many places, it also lead to deeply nested calls to pf_cksum_fixup. When Theo de Raadt somewhen in 2009 or 2010 pointed me to one of these super ugly and not exactly efficient nested pf_cksum_fixup calls, i knew it was time to look deeper into the issue.

On top of this, there is a long standing bug with said checksum fixup and packets pf redirected to localhost and checksum offloading, a bug that Christian Weisgerber (naddy@) has explained to us pf people repeatedly over the years, but it was neither easy to follow nor to fix.

Eventually, at the k2k10 hackathon in Iceland, I started looking into our checksum handling in general, and the findings were quite interesting.

2 Checksum Calculation

The actual calculation of the checksum is quite simple. The checksum is the lowest word of the one-complement sum of all the words the checksum covers, basically.

3 General Performance Considerations

After years of profiling our network stack it is clear that the actual math here is so cheap that it can be considered free on every halfway modern system. The actual integer units are never our bottleneck, the limiting factors are latency and bandwidth to caches, memory and devices.

Thus, a checksum covering a few fields that have been touched very recently and thus are in cache is almost free. A checksum covering a whole bunch of data that hasn't been accessed yet at all is expensive, since the data has to be read from RAM, which is relatively slow.

The actual checksum algorithm has optimized assembler implementations on many platforms we support. Wether these hand-optimizied versions are actually faster than the generic C version is an interesting question that has not been evaluated here.

4 The IP Checksum

The checksum in the IP header (referred to as the IP checksum) covers the IP header. It has to be updated by each router forwarding the packet as it updates the ttl field. IPv6 does not have this checksum.

Since this checksum only covers the relatively small IP header and several fields of that header have just been accessed before, recalculating it is pretty much free. The performance advantage of offloading it to a capable NIC is so small that it gets lost in the noise when trying to measure it.

5 IP Checksum Implementation in OpenBSD

In our network stack - and similiar in the other BSD-derived stacks - an incoming IP packet is handled in ip_input(). Besides a lot of validity checks, ip_input() decides wether that packet is to be delivered locally, in which case it is handed off to upper layers, to be forwarded or to be dropped as undeliverable, e. g. when forwarding is off. The inbound pf_test() call, which makes pf examine the packet, is also here.

In the forwarding case, the packet is handed off to ip_forward, which deals with routing and ttl decrementation. The actual route lookup and some other related tasks are already done in ip_input(), as an implementation side-effect. If the packet is to be forwarded it gets handed off to ip_output().

ip_output() checks and, for locally generated packets, fills in a lot of the IP header fields. The outbound pf_test() call is here as well. Right after the pf_test call the ip checksum is recalculated unconditionally, last not least to cover the ttl decrement for forwarded packets, possible changes done by pf. Locally generated packets do not even have a checksum at this point and get it filled in.

At this point it seems obvious that the ip checksum fixup done all over the place in pf is useless work, since the ip checksum is recalculated just after pf in ip_output anyway, and inbound the check happens before pf. However, pf is not only called from ip_input() and ip_ouput(). There also is the bridge case - the bridge calls pf_test() too, and the bridge does of course not decrement the ttl, nor does it make other changes to the IP header, thus it does not recalculate the ip checksum after pf. This is also the reason why the bridge is special-cased all over the stack.

The solution to this problem is to make the bridge behave like a regular output path. To complicate matters, checksum offloading enters the picture.

6 IP checksum offloading

Pretty much every network interface chip/card made in the last decade is capable of performing the ip checksum calculation in hardware. To make use of that, our stack has been modified a long time ago to delay the actual checksum calculation up until we definately know on which interface the packet is going to be sent out. We can then check wether the interface in question has matching offload capabilities, indicated via interface flags. If so we don't need to do much more but to mark the packet for hardware checksumming. If not, we calculate the ip checksum in software.

To know wether a packet needs checksumming at all we use a flag in the mbuf packet header, a structure attached to the packet data for bookkeeping in throughout the stack. Everywhere we know the packet needs checksumming we plain set this flag.

This works fine for all the regular output pathes. It doesn't for the bridge, due to its lack of checksum handling alltogether.

The bridge code is quite old and not exactly an example for good programming. It is hard to follow. Adding the missing checksum handling in its output pathes - there is unfortunately even more than one - turned out to be not so easy. Once this was done and the bridge special casing all over the stack removed, things mostly worked. Some weird behaviour was eventually tracked down to problems with broadcase packets, and upon closer inspection the bridge uses a gross hack to shortcut broadcast processing, so that a packet that supposedly goes out to a checksum offloading capable interface can get copied and sent out on another interface, potentially without matching offloading capabilities. This resulted in packets being sent out unchecksummed in that case.

Fixing the broadcast hacked was not straightfoward, this needs to be adressed at a later time. The special casing in the stack had to stay. However, with that basic checksum handling and the special casing in place, we were able to stop doing any ip checksum handling in pf, since now all output pathes recalculate the checksum if necessary.

Since recalculating the ip checksum is so cheap even in software on any halfway modern system performance improvements from this change were to small to be really measurable.

7 The TCP and UDP checksums

The tcp and udp, often referred to as protocol checksums, are quite a different beast from the ip checksum. They only cover a few ip header fields, that part is called pseudo header checksum, the tcp/udp header and the entire payload. Due to the full payload coverage recalculating the protocol checksum is not as cheap. While chances are good that the payload is still in cache for locally generated packets, the forwarding case almost certainly means fetching the payload from RAM, since we don't touch it for pure forwarding otherwise.

As with the ip checksum, pf used to update the protocol checksum on the fly, with the same problems as with the ip checksum, just in more places.

8 protocol checksums in the OpenBSD network stack

The procotol checksum handling is much more complex than the ip checksum. As all BSD-derived network stacks OpenBSD used protocol control blocks, in short pcbs, to track connections. Even for udp, which is a connectionless protocol - connectionless on the wire doesn't mean that the stacks don't have some kind of state. The pcbs are looked up using hash tables, or, in OpenBSD, by following the link to them from the pf state.

When a socket is opened, a template pcb for connections from or to this socket is created. The known parts are already filled in and checksummed. Once a connection is made using that socket, the template pcb is copied, the other side's information is added, and the checksum updated for that. This only covers the ip header parts, not the protocol header, and forms the pseudo header checksum. The packet is marked for needing checksumming at this point and then passes on to get protocol header and payload.

Eventually, late in the outbound path, the flag indicating a checksumming need is evaluated. If the interface that this packet should go out on has matching offloading capabilities, we don't need to do anything, otherwise we do the checksumming in software. That's the theory, at least.

The early calculated pseudo header checksum is a hack that might have made sense on a hp300 or a vax, but is counterproductive on any halfway modern system and foremost complicated things considerably. This is where the pf redirect to localhost problem comes from. Some network interface cards - last not least intel and broadcom - implemented their offloading engines so that they rely on this hack, by relying on the pseudo header checksum being there.

However, when such a packet passes through pf, we don't know that it just has a partial checksum, and happily update it for fields it doesn't cover. On a packet that originated from localhost and gets rewritten by pf - prime example being replies to packets that have been redirected to localhost - we have exactly that and end up with a broken pseudo header checksum. For cards that rely on and just update it we end up with broken checksums on the wire. Both the software engine as many other interface hardware recalculate the entire checksum and don't care about the existing pseudo-header checksum.

As with the ip checksum the bridge code had to be updated for this output path to behave.

9 protocol checksum offloading

As with the ip checksum, almost all halfway recent network interface chips and cards support tcp and udp checksum offloading, at least for IPv4. Things are considerably more complicated here tho, since we have to deal with 3 offloading cases: no offloading, pseudo header checksum required, and full offloading. Since the pseudo header case was broken due to the pf handling, protocol checksum offloading is disabled in the drivers in this case.

Unfortunately we have seen many silicone bugs with offloading. While that mostly affects early implementations and is long history, we have recently seen a case with an Intel 10GE chip that corrupted ospf packets - ospf is neither udp nor tcp! - when the tcp/udp offloading engines were turned on.

10 changing the stack to make better use of offloading engines

The basic principle is easy: work under the assumption that we have always have offloading engines. If we hit a path that doesn't, provide a software fallback.

To accomodate for this, the actual checksumming has been moved far down in the stack and is done pretty much as late as possible now, when we know for sure wether we have an outgoing path with offloading capabilities. That allows for removal of pretty much all checksum handling everywhere else in the stack, we just have to set the flag now to ask for checksumming.

Subsequently, all ip and protocol checksum adjustments have been removed from pf, with just the flag modifications remaining. This has interesting consequences. Since we are not updating but recalculating the checksum for forwarded packets that are modified by pf now, that case suffers if there is no checksum offloading available, since we have to access the entire payload. With pretty much any system made in the last 10 years supporting offloading, only the case where pf modifies forwarded packets (that means NAT mostly) being affected, and preventing this - by calculating the pseudo header checksum before and after pf making changes and applying the delta to the existing checksum - hurts machines with offloading capabilities, this seems acceptable.

Since we are not updating the checksum any more but recalculating, we have to consider the case of broken checksums. Adjusting a broken checksum leads to a broken checksum, so all's good - but if we're recalculating, we have ti verify the old checksum first. Again, in many cases we already have that check performed by the hardware offloading engines already, if not, we have to fall back to software. This again punishes hardware without offloading capabilities, but is not fixable without punishing the offloading case substantially. On a halfway modern i386 or amd64 system with disabled offloading capabilities the total hit is in the 5 to 10traffic mix - exact numbers vary a lot with specific hardware implementations. And these systems have offloading capable interfaces.

Last not least this finally allows us to enable the protocol checksum offloading on the interfaces requiring the pseudo header checksum, since pf won't munge the checksum any more.

11 ICMP

ICMP has a checksum too. The icmp checksum handling has been remodeled after the tcp and udp case, instead of calculating the checksum early on and every time changes are being made, we just set the flag. Late and way down in the stack we check the flag and calculate the checksum. Should there ever be hardware that can offload the icmp checksum calculation it is trivial to hook in now.

ICMP shows another interesting case. ICMP errors quote the packet they refer to, usually that is tcp or udp. And this quoted packet of course has a checksum as well. In most cases the quoted packet is truncated and the checksum can't be verified anyway, so there is no real need to adjust it. pf used to do it nontheless. In the cases where the quoted packet is not truncated, we can recalculate the inner checksum as usual, just without being able to use any offloading. In return the quoted, not truncated packets are tiny, so the cost is minor. In my current implementation this is not done, since nothing in the real world cares about the inner checksum. The only case I am aware of is scapy, a program generate and verify packets.

12 performance considerations

On a modern amd64 system we could not see any performance benefit from these changes, not even from offloading in general. On older systems (Pentium M) I have seen around 7indicates that the modern Xeon has so much raw computing power and large caches that other factors, like latency to the devices, hides the saved work completely when just doing packet forwarding.

However, these changes simplify the checksum handling a lot, shortening the code considerably and allow for even more simplifications by followup cleanups.

13 future work

The gross bridge broadcast hack mentioned earlier needs to be fixed so that the special casing all over the stack can go away.

There is no point in the partial early checksum calculations in the pcb templates and upon establishment of a new connection any more, we can simplify things by just calculating the pseudo header checksum in the output routines where we figure out wether we have offloading or use the software engine.

14 Acknowledgments

A lot of help and support, last not least with testing, came from Christian Weisgerber (naddy@) and Mike Belopuhov (mikeb@).

The IPv6 protocol checksum software engine was written by Kenneth R Westerback (krw@).

My trip to AsiaBSDcon has been funded by the conference.

15 Availability

The ip checksum handling changes are part of OpenBSD since 5.1. The protocol checksum parts should be committed soon.

This paper and the slides from my presentation will be availabe from the papers section on

```
http://www.bulabula.org
```

and be linked from OpenBSD's paper section on

```
http://www.openbsd.org/papers
```