

Effective Network Livelock Mitigation

David Gwynne <dlg@openbsd.org>

introduction

- who am i?
- what's my problem?
- what's the solution

who am i?

- i'm a core developer in OpenBSD
 - lot's of experience with storage
 - relatively new to hacking network stuff
- infrastructure architect in EAIT at UQ
 - i do lots, including running firewalls

what's the problem?

- what is livelock?
 - it's a special form of resource starvation where a system gets so busy doing useless work that there's no time for real work
- processing excessive network traffic can floor a computer, therefore causing a denial of service

how does this happen?

- traffic is sent to your machine based on very coarse criteria
 - lladdr (mac), or ip address
- your machine has to process some of the incoming traffic to figure out if it's relevant
 - we care about open ports/connections
 - almost everything else is a waste of time

how does this happen?

- so if you get sent a lot of traffic to closed ports, you still have to receive it to figure out that you have to throw it away
- too much of this and you'll hit livelock
 - lots of work to make no real progress

how does this happen?

- processing traffic you want also has a cost
- its possible the cost of processing wanted traffic leaves no time for you to do anything else
 - like sshing to the box to see whats up
 - this is also bad

how do we fix it?

- this isn't a new problem, it has been fixed before (but not very well)
- there's some rules of thumb
- i run network services on openbsd, so that's what i'm focusing on
 - by "focus on" i mean "write code on"

what is openbsd?

- it is a general purpose unix-like operating system
- descended from the original UNIX by way of the Berkeley System Distribution and NetBSD
- aims for “portability, standardization, correctness, proactive security and integrated cryptography.”

what is openbsd?

- it has evolved into a secure platform a variety of network services
- that's what the contributors were/are interested in
- a lot of users running routers and firewalls on openbsd

axioms

- we cannot predict the future
 - even on extremely small timescales
- networks are unreliable, you can lose packets
- if you are getting too many packets, you can drop some to cope

so what's the context?

- you need to understand how packets get into a system
- you have a network card, it has a wire on one side, and a computer on the other
- i am a genius at explaining things

nic operation

- a nic wants a ring of descriptors that point it at memory it can copy packets into
- producer/consumer style thing
- the kernel allocates the ring, allocates the packets, writes the descriptors, then posts the index of the last descriptor it filled to the chip (it's the producer)

nic operation

- the chip rxes a packet, copies it to the first available packet on the ring, posts an interrupt to the host
- the host enters the interrupt handler, reads the index of the last descriptor used (this is the consumer)
- the packet goes up the stack

nic operation

- the kernel allocates another packet, fills the descriptor, posts the index to the chip
- ie, it all starts over again straight away
 - what the stack does depends on the stack, let's look at openbsd

the stack in openbsd

- the hardware interrupt handler calls ether_input
- ether_input does some rough classification
 - ip packets get put on the ip_input queue and the software network interrupt is hit
- when the hard handler ends, the soft handler is unmasked

the stack in openbsd

- all the ip processing happens at softnet
 - hardware interrupts are higher priority, we can dequeue more packets while processing ip
- we do packet filtering at the start of the ip input (pf)
- different paths for local and forwarded pkts

the stack in openbsd

- forwarded traffic
 - we do a route lookup, pick an outgoing nic, put the pkt on it's output queue, eventually call the nics transmit code
- local traffic
 - this gets queued on a socket for userland

back to the problem

- each of these steps takes cpu time
- there are a lot of places in the stack where a packet is determined to be useless
 - so the time spent processing it is wasted
 - its trivial to get all the way into ip_input before we discover the pkt is useless
 - syn floods, ping floods, full sockets...

back to the problem

- even if we want all the traffic we receive, there is still a cost associated it
- receiving the traffic is a higher priority than processing it
 - hardnet is higher than softnet
 - kernel is always higher priority than userland

back to the problem

- this is why flooding certain types of traffic cause a DoS
- in addition to exhausting cpu time, you can also run out of memory
 - you have to provide memory to rx into
 - if you rx but never process, you can run out of memory to rx into

back to the problem

- memory exhaustion a huge problem too
- you need cpu time to give to userland to process and therefore free memory
- you need userland to run for things like ipsec rekey, bgp keepalives, and so on
- anyway, let's look at some solutions

interrupt mitigation

- entering and leaving the isr has a cost
- the chip can delay raising the interrupt until a timeout is reached or a certain number of packets are rxed
- the isr cost is mitigated, it's spread over several packets rather than added per packet

polling

Device polling is a technique that lets the operating system periodically poll devices, instead of relying on the devices to generate interrupts when they need attention. This increase server network responsiveness and performance. In particular, polling reduces the overhead for context switches which is incurred when servicing interrupts, and gives more control on the scheduling of the CPU between various tasks (user processes, software interrupts, device handling) which ultimately reduces the chances of livelock in the system.

polling

- a lot of that quote is a lie (aka marketing)
 - but it is true it helps avoid livelock
- it's basically an extremely deterministic form of interrupt mitigation that the host controls
- hard limits on the pps a box will handle
 - polling only handles X packets every tick

polling

- the pps needed for high throughput (eg 10Gb) divided by the polling frequency (eg 1000Hz) means you need very big rings (megabytes of buffers)
- you get very spiky load (100% usage after every tick, then 0% till the next one)
- think of packet latency (ping times) too

polling

- if you increase Hz to compensate, you'll end up doing the same rates as you would with interrupt mitigation
- if you're not doing the pps, then you're wasting time by polling frequently
- just do int mit properly and avoid the extra code complexity polling requires

receive side scaling

- should be called rx side spreading
- instead of a single rx ring, you have many and you tie isr's for each to different cpus
- the nic spreads traffic over the rings
 - usually the ring is decided by a hash on header characteristics (ips, ports, etc)

receive side scaling

- DoS from a single host will still livelock the cpu the ring is on cos the hashing sends the traffic there
 - but you have other cpus with idle time
- but small systems (think embedded) still only have one cpu

socket buffer limits

- kernel limits the amount of data queued on a socket for handling by userland
- data over this limit is dropped
- mitigates against memory exhaustion
- also mitigates how much memory a slow process can make the kernel hold for it
 - its good, but not the complete solution

ip input queue limits

- hardnet takes packets off the rx ring and puts them on a queue for softnet to handle
 - but hardnet is higher priority than softnet
- you're wasting memory on packets you dont have time to process
 - ip input queue will grow but never shrink
 - so we limit (or you run out of memory)

pf congestion counter

- running pf adds a cost to every packet
 - the ruleset evaluation is really slow
- if you fill the ip input queue, then we drop packets hitting the ruleset temporarily
 - makes softnet do less work so it can catch up with hardnet
 - prioritizes existing connections

semi-polling

- if ip input queue is full, turn off nic interrupts till we catch up
 - gives the machine room to breathe
- but the rx ring fills without isrs clearing it, so you end up filling the input queue again and going back into the masked state

softnet limiter

- detect when a lower priority task has NOT happened
 - we do it with softclock
- make softnet processing drop packets until we're sure timeouts have run, then we start the stack processing again
 - its like a generic pf congestion handler

the solutions suck

- there are more, and i only have so much time. however, why do they suck?
- the packet gets into the box
 - try to do the smart dropping of packets around softnet
 - by that time you've done a lot of work to get them there, all so you can drop them

the best solution

- invent prescience, then give it to your isp so they will drop anything you wont want
- next best thing is to have the bgp thing were you tell your isp which hosts to filter
- but not everyones big enough to have this
- and not all load comes from outside

the smart solution

- you want to handle as much traffic as possible and as smoothly as possible
- but you want to back off when you livelock
- in livelock you need to drop packets
 - ideally before they end up in memory, ie, you want the nic to drop the traffic for you when you're in trouble

here's how

- dont give the nic (as much) memory to rx packets into when you're in livelock
 - ie, dont fill the rx ring
 - a small number of descriptors on the ring plus interrupt mitigation means extremely fine grained control over how many pps you will do and how evenly theyre spread apart

rx descriptor limits

- the trick is to know what the limits are
 - you cant hardcode values because of the huge variety in systems and configs
 - you cant add buttons cos people are dumb, lazy, and inclined to break things
 - so you need an adaptive system with livelock detection

livelock detection

- this bit is easy, we steal from softnet limiter
 - if a task at a lower priority than the network stack hasn't run when we expect it to, we're in livelock
- we compare the wall clock against its value last time we ran a soft clock tick
 - if they diverge then softclock isn't running

figuring out the limits

- you need to know how much network traffic you can do just around the point you hit livelock
- remember, we're going to limit the rings at livelock, but we still want to do a reasonable amount of work still

figuring out the limits

- we start a nic up with (generally) only two rx descriptors on its rx ring
- a nic has to prove its using its allocation before we give it more
 - if it uses its entire allocation in a single isr, we grow the limit by one descriptor
 - it has a bigger ring to use for the next int

figuring out the limits

- this means that a busy system moving a lot of pps will grow the number of descriptors on its rings, but it can only grow to its natural limit
- we now know how many packets per isr the system can handle because it has proved it by using them

livelock mitigation

- but it is possible to grow the number of descriptors a little bit too much
 - as soon as that happens, we spend too much time in hardnet/softnet, we miss a softclock tick, and livelock is detected
 - next time any nic asks for more descriptors, we halve its allocation and peg it for a small interval

livelock mitigation

- the nic is still able to rx traffic, but there's a reasonable limit on how much
 - extra traffic gets dropped by the nic
 - no rx processing by the kernel for traffic it will just end up dropping
 - saves mem/io bandwidth by avoiding useless dma

livelock mitigation

- a short time after we detected livelock, we allow the rings to grow again
- this does mean some hysteresis
- but between functional extremes, not all and nothing, so it is relatively smooth

the implementation

- packets in the kernel are represented by structures called mbufs
- kernel mgmt stuff, pointers, etc, and a small bit of space for data (real small)
- mbufs can point at any external buffer for big data, but there is a generic allocator for 2k byte chunks called MCLGET()

rx rings

- drivers generally use mbufs and MCLGET() to allocate their rx buffers
- traditionally drivers filled their rx rings, eg, em had 256 entries, so it had 256 descriptors filled (512KB)
- makes producer/consumer easy since they become the same thing

rx rings

- drivers actually prioritized keeping their rings full over anything else
- if they rxed a packet, but couldnt replace it on the ring, it would put the rxed packet back on the ring rather than process it

enter MCLGETI()

- MCLGET is a macro, it's really m_clget() that adds a 2KB cluster to an mbuf
- the problem is that clusters are a fixed size
 - this whole thing started because i wanted a generic jumbo allocator
 - every driver that does jumbos writes its own version of m_clget, but they suck

MCLGETI()

- jumbos are bad because they're so big
 - kernel memory is a precious resource
- i wanted the restricted ring semantic
 - we start low, and grow on demand
 - if you dont use jumbos then you dont waste memory, but if you do, they appear

m_clget()

- so m_clget now takes two extra args
 - size of the buffer you want (for jumbos)
 - and which interface it is for
- this is where the rx ring accounting is done
- NULL for the interface argument if the cluster is not for an rx ring

MCLGETI()

- MCLGET macro was fixed to fill the new args to m_clget with 2k size and NULL
- MCLGETI exposes the new args
- drivers have to be modified to use it
 - this is surprisingly hard, but it's getting easier with practice

MCLGETI()

- clusters allocated via MCLGETI() get counted against the nic
- when they're freed or sent up the stack they get "uncounted"
- if a nic uses all its allocation in an interrupt, we raise the limit inside MCLGETI()
- if we livelock, we lower the limit

results

- we have a lot of nic drivers (over 100)
- we've converted 15, most importantly:
 - em, bge, bnx - common on big boxes
 - sis, vr - common on small boxes
- you can't kill the cpus on these boxes with network traffic now

fin