g'day my name is brynden and this talk is on bpf internals i'm going to show you how it works

right down to the machine code i've not shared this content before but as it turns out it's actually quite

easy to do this i'm not going to go through the bpf reference material that lists

instructions and capabilities because that's already online what i'm going to share

is examples of bpf for tracing and i'll show you all of the components

work so you can see how it fits together to start with a quick introduction on bpf

bbf comes from bsd and it originally stood for berkeley packet filter this was an obscure technology to speed

up tcp dump and packet filtering since packet rates can be very high so with bpf

user space could define a filter that was then compiled into the most efficient instructions

and those were run close to where the packets were ingested what was interesting about this

technology although not many people knew about it was that the way it worked was like a

virtual machine in the kernel that has been extended since 2013 and

this extended bpf ebpf modernized bpf turning it from 32-bit

into 64-bit giving it more registers giving it virtually unlimited storage space

but also instead of only being attached to packets it can now be

attached to many different event sources and run many programs within the kernel

so bpf today has evolved it's now no longer an acronym it is a

technology name like lvm although bear in mind some people still call it ebpf to really make

the point that it's new however kernel engineers and myself we tend to stick to bpf

and it is a generic internal execution environment for running programs in the kernel

i've drawn this table to help you understand the differences between bpf and kernel software and user mode

software so i care a lot about performance and with resource access

running things in the kernel is fast you're closest to hardware

bpf is closer than user space so you have restricted helpers from the

kernel that can access hardware so it's faster than user space not as fast as the kernel itself

however it's safer than the kernel so it's failure modes it won't panic the kernel

and it also runs in a sandbox so it's more secure and so this makes it a

better environment for running for developing and running these add-ons into the kernel a

safer environment with high performance

another diagram i've created to help you understand how bpf works is it is event-based

programs so i create a bpf program it's enabled when an event fires i go on cpu execute

and when it ends it's waiting for the next event there are helpers these are advanced

functions that bpf can call to take various actions in the kernel

and then some bpf programs use spin locks it's not that common

and if the bpf program is flagged as sleepable it can also enter a sleep state although

that's very specialized for user space copyings

quick overview of bpf for tracing given the ability to run mini programs

in the kernel i've used it for creating many advanced performance analysis tools or tracing

tools or observability tools why did i need programmatic capabilities it's because i wanted to calculate

latency from one point to another i wanted to calculate histograms averages

and all sorts of the performance metrics that i consume i can now do those calculations in

kernel space where it is cheap and efficient and just emit the summary to user space

so this has allowed me to create all of these tools that are now much more practical this diagram i've

decorated it with various tools many of which i've created in red extra tools i created for the bpf

performance tools book these are all open source

as an overview of ppf tracing front ends if you just want to run some tools there then the recommended ones are bcc

and bpf trace if you want to hack up some new tools bpf trace it's a high level language

and if you want to develop bpf products there's a number of front ends like bcc lib bpf or doing it in go bpf

and so on if this seems like it's a little complicated to have different front ends you're already used to that with unix or

linux so an analogy with running tools is like running things from user bin

and for to hack up new tools that's like writing shell scripts and to develop products that would be

like writing c code or c plus plus code now for the internals

here's an overview of how bpf tracing works so on the left i've got user space this

is a bpf tool it has a bpf program that's turned into bytecode

and that's sent to the kernel via verifier it may also use btf bpf type format for

structure information and then the ppf program is connected to different event sources via

an event config specified in user space and then can emit output either per

event data from a perf buffer or map statistical summaries or custom summaries

via bpf maps i'm going to use various terminology

just check that you know each of these

so the first tracing example i'll go through is dynamic tracing or instrumentation of

the do nano sleep kernel function where i'm printing

out the process id that was sleeping this is a bpf trace one liner

and here's its example output it's attached to probe and i and now each time do nano sleep fires i

get to find out which process id was sleeping by itself it's not hugely useful

but explaining how this works is very useful you'll see how all the components fit together

so we have that program we want the bpf bytecode we want the kernel events mapped to the

bytecode and we want user space printing the events

here's a diagram of bpf trace mid-level internals that i've drawn and to explain bpf i'm

also going to explain bpf trace internals as it's a great example

of using bpf really well

the program actually goes through these transformations the bpf trace program which is ascii

text gets first converted into an abstract syntax tree

which is commonly used for processing languages it's simplified it simplifies things

that then gets converted into llvm intermediate representation and that

then gets compiled into bpf bytecode and then that gets compiled into machine code

so to start with step one our bpf trace program it's that it's text it's ascii

step 2 we want to parse it and turn that into an abstract syntax tree which helps us

then convert it into the instructions

here's an example of what we want it to look like once that conversion's finished so probe k

probe do nano sleep calls printf and those are the two arguments

and we're going to use lex and yak this is very easy you may have done lex and jack at university

likes to apply regular expressions to identify content

and then yak has the grammar expressions to understand the program structure

and the end result of that is the ast as an example from that program so

here's process id and you can see this is really a table of regular expressions

so his pid gets matched and that then identifies it as a

built-in and you can see various other regular expressions here as well for matching

things the grammar side of things if it's a built-in

it will then call ast built in and that creates the ast node for the built-in so that

has completed the process from identifying the pid string and now we have a ast built-in node for

pid here's how it looks for printf very easy

printf that's a call and then in the grammar

call is an expression and call can have variable arguments and they get

passed into the ast c plus function

and that creates the node i found this detail a little interesting

this is showing how this string is passed and so here's a double quoted string and

a little more complicated in alexa but here you can see how the escaped new

line gets turned into a new line

and then in the grammar we then create the ast node for the string

now for the probe itself k probe do nano sleep so words are identified as idents

and so that will match k probe and an ident colon and a wildcard expression

that is our attach point or probe point and so that matches our k-probe do nano

sleep and it gets called as the ast attach point function and i've

enumerated the steps there straightforward

for the program structure itself here's some of the grammar rules for parsing the

curly brackets and semicolons the statements that are in a program block

so the end result of lex and yak is you have the ast nodes in fact here i've got bpf trace to print

them out for us bpf trace minus d and it's printed out the ast structure

next is processing structs

so especially in the kernel sometimes you might want to walk strucks task struct and fish out various members

of interest and there is a clang parser for helping with that

there is also btf which is the new kernel feature which has a lightweight representation

of the type format embedded in the kernel that bpf trace can refer to and there's also

argument information for trace points so this step then converts those

dereferencing into the correct memory offsets for this particular example i don't have

i'm not dereferencing anything so i don't have those steps here but if i did that's work that's

where it would happen in the sequence the next is the semantic analyzer this

does syntax checks on the program map creation it adds probes

and it catches many program errors so in this example i've mistyped pid pd and in bpf trace semantic analyzer

it then will print unknown identifier so here we go

if it couldn't find it and there's over 2 000 lines of code in the semantic

analyzer and that's really a feature we want to add as many of these tests to help newcomers

learn the bpf trace language by giving them meaningful error messages

now if you if you survive the semantic analyzer you have a valid program now it's time

to convert it into bpf byte code but we're not going to convert it directly we're going to

leverage llvm the llvm compiler and the reason is llvm

has all of these optimizations and it already has a bpf target so if we can prepare that program into

llvm's intermediate representation it will then do the rest for us

so converting into ir now we're walking through all of the ast nodes so

here we're visiting built-ins if the ident is pid then call

createpidtjid and do a creative logical shift right instruction

how these work is create get pidgid it's in ir builder dot c plus

it ends up doing a create call and it's using as an argument

bp a funk get current paid tged this is the helper call number that's defined in pp in bpf and it's provided through lib bpf

so we're doing a create call for the helper call number

the helper call number comes from the include file bpf.h and it's number 14 in the list

after walking through all the ast nodes you end up with l vmir and it looks like this you can get bpf

tracer printed out as well vpf trace minus d so pid

and it gets converted into this so here's call and there's number 14. so we're going to

call helper number 14 and there's our logical shift right

and that shift right is because it's fetching the bits necessary to get the pid because that

return value has both the pit and the tg

this is all generated from the ir calls and if you read through this you'll see

there's lots of create alec a bpf create get element pointer and so on

it turns out it gets verbose i can't imagine convincing someone to

write i uh it would go like this it's imagine assembly

but much more verbose you have to do these calls you don't normally have to do

but not only that you're not going to write this assembly directly you're going to call functions that then

write the assembly so to start with you'll have to debug

errors on your function calls and then when you've got them working you then debug errors on the lvm ir

itself and have to backtrack them to see how you change your functions to call the

right ir and also it's not ultimately documented as well as assembly

so if you like programming challenges what i'm saying is lvmir does get challenging

to generate this it just i mean it is just very verbose but the ultimate advantage of going

through all of this is lvmir

has a back end for bpf and so once we have it lvm my lfm can then do the rest and we

have not just byte code bpf trace bytecode but optimize bpf trace bytecode

on this diagram i've also drawn something else so i've drawn bc for a bpf compiler

and i'll get back to that in a bit so let's finish talking about lvm

so we want to emit bpf instructions their ebpf instructions are

64-bit wide for here i've got an example for the alu and jump classes

so we've got four bits for the upcode now for our called get current pid tjid

we know it's helper number 14 that goes in the signed immediate constant the opcode is bpf call and the

instruction class is bpf jump they're in the linux include files so

bpf.h jump becomes 5 and call becomes

0x 80 in hex these are all together so our final op code is 0x85

and then we've got e0 here that's our 14.

so there's our final bpf bytecode straightforward

llvm clang will do that for us so it will go through those steps and

omit the bpf bytecode now i do want to mention

lvm can be a little bit heavyweight to install in some systems especially if

you're thinking about embedded systems and this is why i've written here we

are working on a bpf trace internal compiler as an option

that has no dependencies and will be lightweight and so this would be

suited for embedded the downside of the bpf trace internal compiler

is it won't emit as optimal code as llvm has many optimizations so that's a work in progress

so with llvm and clang lvm has the bpf target it reads those

header files it's part of the target and so it knows all of the codes to use to generate the bpf bytecode

to show an example of how it works i actually find this really interesting because this is one part of bpf i didn't work on

myself this was alexian chandler and jung hong who did the bpf back end so here is llvm

target bpf and if the class is call we've got alu jump and

bpf call so these get turned into the 85 and here we're setting the bits for

the immediate value which is going to be 14 the argument and that's what emits the bpf byte code

it ends up just being a lot of llvm boilerplate and so you can go through this file and

you can see how all of the llvmir maps to the bpf instructions

now we have bpf bytecode and here's our bpf funk get current pid

tg there's our number 14. this is the full program from earlier jump call

we now need to send it to the kernel and there's a bpf syscall for that the

bbf cisco has different actions so we can load programs it can also create maps and read from maps

i'll mention that later so we're going to load a program it's a type k probe and it's got the

instruction account and the address success passed the verify it didn't return an error

so the verifier when we send this bpf by code to the kernel

this is how it works and i've begun with a different diagram showing bpf internals this is from my bpf book

and we've got the bpf instructions we first go to the verifier and it may reject it if you're doing

something you shouldn't and then it gets passed through the bpf

jit there's actually multiple passes or phases there used to be a way

to have the instructions interpreted so these were not g compiled however

that has been compiled out of recent kernels and so that's permanently disabled we're no longer going to do

that we want people to use bpf jit compilation for two reasons one it's a

little bit faster it generates a little bit faster instructions and two it allows us to

patch things patch security holes on bpf pi code dynamically

and so if there were issues like the meltdown inspector issues with bpf by code programs we can just change the jit

to patch over them so it gives us extra security benefits as well

at the the result of ppf jetties you now have native instructions it's not just those instructions that

make up a bpf program you can also call helpers like bpf get current paid and tged bpf

getcurrentprocessname.com and there's lots and lots of other helpers also as part of bpf we have the

events k probes and so on and the context they bring and then the bpf syscalls

like to load programs which was over here and here i've got for maps look at

lookup element and so on i'll talk about them more later so that's a different view of bpf internals

so to verify bpf instructions imagine we call a bogus function so here

i've got one two three four five six seven eight here is the beep kernel bpfverifier.c

and check helper call if the function id is out of the valid range return ian val

so very simple the verifier has over 9000 lines of code

so it will check lots of things and here's some examples of the verifier functions

check sub progs register arguments stack write stack read stack access and so on

there's over 260 error returns in the verifier so it's going to check every instruction

every code path it'll also rewrite some bytecode as a phase just for efficiency later on to make

some things easier to execute for memory access direct access to

memory is extremely restricted you can only read initialized memory and you can only

read from certain areas like the stack context like event context map

values and sockets many of my tracing programs i need to read arbitrary memory i need to read

disk information or tcp information or task struct information and so to to access them they have to

pass through a helper bpf pro breed that does all sorts of safety checks and checks that that reading that memory is

going to be okay and then it puts it into an area that i can then load and store directly

so it checks all the instructions arguments of the correct type register usage is allowed no right overflows and

so on it also verifies code paths that all instructions must lead to an exit

that there's no unreachable instructions there's no backwards branches or loops except for the allowed bpf

bounded loops so we can't just code our own loop by mistake i'm visualizing these are actual

bpf programs i'm visualizing using graph phase

so here's a pre-verified bpf bytecode it's got a phase where it rewrites some things for efficiency

so here's the post verifier bpf byte code

and just to explain it a little bit so call get current tg instead of still having the helper

number 14 just to save a step that's become the instruction offset address from ppf call

base just to make it quicker to jump to that those instructions when we're actually

running on cpu you can print out this bpf bytecode

using bpf tool so instead of just reading these lists of bytecode with human words

bpf prog tool prog dump excellated up codes and i can see here's the bytecode but i can also see

up the up codes and then a disassembly so what the instructions

are actually doing here's our the op code 85 and that's the it's going to be a call and it's doing

the bpf get current tgid

if i don't put the argument up codes then it doesn't include the beep the full bpf bytecode so

you get a more condensed listing here's the entire program so i can just see the column of op codes only and then the disassembly for the

bpf program

that was the first phase of jet the second phase will turn it into machine code

and so we've got the bpf byte code there's different jits for different architectures x86

arm spark and so on here's an example of the actual source code that does the

jit compilation in the kernel so this is from arch x86 net bpfgcomp.c

and as it turns out it's actually quite easy it is a big loop that iterates over each

instruction for i equals one to the instruction count and then we switch on the instruction

code and it's a case statement for each of the instructions so if i'm the bpf jump or

call emit call and make call itself will end up emitting e8

which is the x86 call instruction and so this is a big file it just maps

those bpf bytecode into the x86 machine code

and now you have x86 machine code there's our call getcurrentpitgid

e8 and there's the offset for that helper function or you might have arm machine code and

boy that looks different because it's risk and not those cisc instructions it's

also longer 48 instructions you can also use bpf tool to show the

x86 instruction disassembly so instead of just looking at the

machine code this shows you the instructions in a human readable way

and there's our call for the helper

i do want to mention that what we're looking at isn't all of the bpf program because we've got

these calls where you go and call a helper and it does things as well so to fully finish telling the story of what's running

here's the helper code bbf get current tgid and this is it this is the this is all the code for it

it returns the target group id and the process id as a uint64

so packs them together and user space can unpack them great now let's see how we attach

that machine code or bike youtube machine code to events

again i'll use fstrace to show you how user space is doing this so we have a bpf call

to load the program that returns file descriptor 14 so that now represents our bpf program in the kernel

we do perf event open that returns 13 for the k probe so perfect open is what

initializes the k probe and now we use ioctl to map the two together

so k-probe 13 runs bpf program footing

how do the k-probes work so we were actually looking at do now

sleep and this is the source code this is from time hr timer.c how do we instrument when that fires so

it turns out it's actually quite easy if you look at the instruction disassembly using gdb of the kernel

functions you'll have noticed that they start with f entry this is the f trace function

f trace is a different tracer profiler that's built into the kernel and we call that at the start of every

function we actually don't call it at the start of every function because at runtime there is a a boot time pass that knobs

them all out so they become fast knob instructions and f trace puts them back into these

call f entries as needed bpf or k probes can use a

f trace optimization so that it adds its program onto the f entry so that's one way k

probes can instrument the start of do nano sleep but k probes have different ways they can work as well they can also live

patch instruction kernel instruction memory and put a break point like an in three

or a jump instruction in there as well to go and do the instrumentation it will call stop machine to ensure that

other cores don't execute instruction text as it's changing for safety

that's okay rob's worked now we want to print the information out how does that work s trace

it opens perf event open and we open output buffers for each cpu

and bpf trace then reads them using e-pull weight and this is also why output from bpf

trace can sometimes be out of order because we're reading the cpu buffers in turn

and it's also why i will post sort output to make sure it's in the right order

and then bpf trace then prints out what we send out of the perf buffer just some internals about bpf trace

we actually use the printf mechanism to do more than just printf

so to start with there's an id which identifies the format string we're

going to print and then it has the arguments for the format string but if that format string is really high

that the printf id is really high then it's a different action altogether so instead of printf we might be doing

a exit or a clear or something else so we we just use the same perf output

mechanism to do all of the bpf trace async actions and there's the code in bpf trace that

actually prints out the message so we go and pick the arguments out of the perf

output buffer add values and then we just print it out and that's the output

now to show some different internals let's have a quick look at static tracing and map summaries we've

already covered most of the internals i'll just show those differences this time the one liner is using a trace

point that's static tracing of disk issues and we're going to do

a map frequency count of the process name here's the example output this shows

that dmcrip write did 1993 disk issues

block devices issues while tracing we start off with the parser this time

it's maps so it's got the at symbol it gets identified as a map it could have variable arguments it can have

one or more keys to the map and then we create the ast node for the map

maps themselves this is custom data storage so it's key value storage

and you can put lots of things in there so i use them for statistical summaries for frequency counts

this diagram i've drawn shows the process names and the values

they're created from user space so the bpf syscall can do a map create and then can look up the elements and

there's some equivalent functions in the kernel for looking up and updating elements so you can do that from the bpf trace

program as well here's how they're created bpf bpf map

create type per cpu hash and it's setting the key size value size and max entries

how trace points work well they're different to k probes because this is baked into the kernel source code

so there's header files this is trace events block.h here's the block rq class and we define

the arguments that get shipped with it and here's defining the event

and then here's the actual source code so this is now block block mq.h and trace block

rq issue trace points are a best effort stable

interface they are better to use than k probes since k-probes are instrumenting

the raw kernel functions they can change from one minor version of the kernel to the next and that ultimately breaks your tracing

programs trace points try to be stable so that for a period of years your tools

won't break so definitely try to use these trace points if they're available

we do tend to use k probes because trace points aren't everywhere so there's trace points in a lot of important

locations but when they aren't available i'll use k-probes

now i see there are trace points in a lot of important locations how do we add them to the kernel without

incurring overhead and it's actually quite easy what happens is

when they're compiled it leaves a knob instruction here's a five byte not placeholder that does nothing very

quickly when the trace point is enabled that gets rewritten to a jump instruction to the trace point

trampoline and then when you stop using the trace point it gets written back

how we print them out bpf trace has a while loop bpf get next key look

up the element and print it out those are functions from lib bcc and lead ppf and they do the bpf syscalls

with the arguments i listed earlier

and i can s-trace it and you can see it working through a map to print out all the elements

wouldn't this have high overhead if i'm having to do cyscals to read every bpf map element

well bear in mind that in the tracing programs i write i often only read a map once a program

exit to print out the summary or maybe it's once per 10 seconds to do an interval summary

so yes it's a lot of syscalls but it's an infrequent activity to go and dump a map

and there's the final output there's the process names and accounts

if you're interested in bpf internals there many there are many other topics briefly there's stack walking so bpf

uses the perf infrastructure in linux to walk stacks and so that can primarily do a frame

pointer base walk but in the kernel it will support awk based walks as well

btf is the new kernel capability to store all the type information

so that we can de-reference kernel structs there's core that's the bpf compile once

run anywhere so that's how we can now create these elf binaries that have the bpf bytecode embedded in them

and they can even work on different kernel versions as they have relocation data

there's the kernel self-tests which are very important for bpf it's why it's so reliable it has many many self-tests there's the

new event types there's raw trace points that are a bit faster than trace points and f entry there's other use cases of

bpf as well and that has internals so networking security and so on

so there's much more if you're interested so as a high level recap we went through

the user space ppf tool how it would send bpf biker to the

verifier and attach events to the bpf program and it could be per-event data or map

summaries internally with bpf jig compilation

emitted the native instructions the machine code that we saw and they can call helpers to do more

advanced things if you haven't turned this on yet please

turn it on it helps with the future of bpf programs as it gives us that extra struct information

it's already been turned on a lot of places for more information please see these references they include the kernel

headers in particular and the psyllium bpf guide is also great

also please see my book bpf performance tools that i wrote with help from the bpf

community and that's my tour of bpf internals by

example i hope you found that useful and you've learned a few things about how bpf works internally and as you saw

it's actually quite easy thank you

you

Inglese (generati automaticamente)

TuttiVideo correlati