Zero Trust, Software Defined Perimeter, and P4

Open Networking Summit, Europe 2019

Omer Anson
Huawei
Introduction
Who Am I?

- Omer Anson
- Software Physicist
  - About 11 years under the keyboard
  - Almost four years at Huawei
  - Working on (mostly):
    - Linux
    - Networking
    - Cloud
Introduction

Motivation
Zero-Trust Networking

- Everyone and everything has an identity
- Doesn’t matter who they are
  - Lambda
  - Microservices
  - Users
  - External entities
  - PaaS
  - IaaS
- Pure Whitelist Security
  - Only permitted entities can communicate.
- Helps prevent data breaches
  - Read: Saves money!
Software Defined Perimeter (SDP)

- VPN mesh with whitelist policy
- All endpoints are authenticated and identified
- Endpoints only responded to permitted endpoints
Introduction

Technologies
Packet flow in Netfilter and General Networking

Image from Wikipedia: https://en.wikipedia.org/wiki/Iptables
OVS / OpenFlow

- OpenVSwitch - Virtual OpenFlow switch
- Allows to define whole pipelines
- Match-action rules (flows)
- Example flows:

```
table=55, priority=100, metadata=0x1f, dl_dst=fa:16:3e:d3:3d:8b actions=load:0x66->NXM_NX_REG7[], resubmit(,75)

table=55, priority=100, metadata=0x1f, dl_dst=fa:16:3e:c6:23:3f actions=load:0x67->NXM_NX_REG7[], resubmit(,75)

table=55, priority=200, metadata=0x1f, dl_dst=fa:16:3e:2e:a1:eb actions=load:0x10->NXM_NX_REG5[], resubmit(,60)

table=55, priority=200, metadata=0x1f, dl_dst=fa:16:3e:3a:39:bf actions=load:0x10->NXM_NX_REG5[], resubmit(,60)

```

```
table=55, priority=100, metadata=0x1f, dl_dst=01:00:00:00:00:00/01:00:00:00:00:00 actions=load:0x67->NXM_NX_REG7[], resubmit(,75)
```

```
>>>>>>>> actions=load:0x67->NXM_NX_REG7[], resubmit(,75), load:0x66->NXM_NX_REG7[], resubmit(,75), load:0->NXM_NX_REG7[], resubmit(,75)
```
extended Berkeley Packet Filter (eBPF)

- “Super powers have finally come to Linux”
  
  - Brendan Gregg

- In-kernel Virtual Machine
- Userspace code running in kernel
  - Change behaviour during runtime
    - In our case: Packet processing
  - Provably safe
- Many entry-points
  - Device and socket packet filtering
Why Not Write Directly in eBPF?

- Lots of boilerplate
  - Redundant verification
    - Can be inferred automatically from P4 code
  - Map structure must be defined beforehand
    - Can be inferred automatically from P4 code
  - Internal test, without defining headers:
    - P4: 70 LOC
    - eBPF: 160 LOC
P4

- From the website:
  - P4 programs specify how a switch processes packets.
- Domain specific language
- Specifies how to process packets
- Protocol Independent
- Compiled to different backends and architectures

```c
// Parse packet
parser ingress_parser(packet_in packet, out headers_t hdr,
                        inout metadata_t meta, inout standard_metadata_t std_meta) {
    state start {
        packet.extract(hdr.ethernet);
        // ...
    }
}
```
P4

- Missing compilation options
- P4 has a reference implementation switch, but
  - Slow
    - Userspace
    - Geared towards flexibility, not speed
  - Unstable
    - At least, compared to the Linux kernel
- So let’s compile to eBPF!
But there is a P4->eBPF compiler!

- Current eBPF compilation is feature-poor
  - Can’t change packet structure
  - No redirection
    - Packets either pass or drop
- Translates to C
  - High level language
    - Optimisations are less precise
  - Another layer of indirection
    - Programme structure information loss
    - More code; more room for bugs
So Let’s Write a P4 Compiler
LLVM

LLVM Compiler Infrastructure

- Compiler and toolchain technologies
- Modular and reusable
Skip the Middleman

- Translate Directly to LLVM IR
  - Better support for optimisation
    - Add entry-points and placeholders
    - Example: Strong use of `mem2reg`
    - Example: `undef` -> 0 replacement
  - Better control over generated code
    - Can add debug symbols and code locators
  - Less computation
    - Meaning less room for bugs
The Technicalities
The Technicalities

● So we wrote our code in P4
  ○ Implemented identity
    ■ For Zero-Trust
  ○ Implemented security
    ■ For Zero-Trust and SDP
  ○ Implemented the kitchen sink

● Now what?
The Technicalities

Code
The Technicalities - Code (L2)

```c
control l2(inout headers_t hdr, inout metadata_t meta, inout standard_metadata_t istd) {
    action set_dst_lport(bit<32> port) {
        meta.dst_lport = port;
    }

table l2_forward_tbl {
    key = {
        hdr.ethernet.dstAddr: exact;
    }
    actions = {
        set_dst_lport;
    }
}

apply {
    l2_forward_tbl.apply();
}
```
The Technicalities

Compilation
Compilation

```
Make
  P4 Code
    p4app.p4
    P4 compiler
  LLVM IR
    p4app.bc
    <arch>_ebpf.bc
    Link with architecture
    Optimisations:
      * O3
      * tolerate undefined
    LLVM Compiler
  eBPF VM
    p4app.o
```
The Technicalities

Loading (an example)
“Physical” Network Structure

- Single P4 Controller namespace
- Each network element in its own namespace
  - Connected with a veth pair
- Where to load compiler output?
  - P4 Controller only
  - Leg of veth pair
- We’re good to go!
You Promised Zero-Trust and SDP!

- Write the logic in P4
  - For Zero-Trust
  - For SDP
  - For the pygmy marmoset passing the packets
- Compile it
  - That’s where this work comes in
- Load it in the gateway
Summary
Conclusion

- Wrote a Zero-Trust and SDP Gateway in P4
- Wrote a compiler: P4 -> LLVM
- Use LLVM optimisations
- It works. Managed to compile, run, and test
Thank You!

Questions?
Thank You!

(Come Again)