Securing the Floodlight OpenFlow Controller

Rob Jellinek and Adam Vail

May 5, 2013

Abstract

Need to do the abstract

1 Introduction

In recent years the advent of software defined networking (SDN) has introduced a major disruption into the networking space. Traditional networks employ vast amounts of hardware that all need to be configured and managed manually by teams of trained professionals. Manual interaction with so many different pieces of hardware causes the network to be suceptible to costly human-errors. The need for a scalable network configuration and management system has led to SDN's rapid rise in popularity. SDN implementations today require switches that can take direction from a centralized controller. These switches can be software based, such as Open vSwitch [4], or hardware switches which offer traditional and SDN modes of operation. Switches deployed today for use in a software defined network generally communicate to the controller using the OpenFlow protocol.

OpenFlow [3] is a standardized protocol that facilitates coordination between the data plane, which resides in the switch, and the control plane, which resides in controller. The controller makes high level decisions, such as routing, dropping a packet, or forwarding a packet, and communicates these decisions using OpenFlow messages. The messages are stored in the switch's "flow table" which allows the switch to handle packets without repeatedly involving the controller.

We have extended the Floodlight OpenFlow controller [1] to incorporate security policy enforcement. Our implementation detects if an application attempts to modify a switch's flow table with a conflicting rule and disallows it. This prevents malicious applications from subverting security policy decisions but in place by network administrators. It also provides a useful debugging system which allows administrators to discover applications that exhibit conflicting behavior within their network.

2 Design

We have extended the Floodlight OpenFlow controller to ensure two conflicting OpenFlow rules can not be present on a switch at the same time. Since Floodlight has a global view of the network it can make informed decisions concerned which rules an application is allowed to install on a switch and which rules should be disalowed. Much of our work for Floodlight was inspired by FortNOX [5]. FortNOX created a security enforcement kernel for the NOX OpenFlow controller [2] which did similar things to what we have created in Floodlight. Unfortunately, NOX is no longer under development and FortNOX was never released to the public. The need for functionality similar to FortNOX in a widely supported controller drove us extend Floodlight. We also discovered several shortcomings of FortNOX which we adressed in our Floodlight implementation.

2.1 Alias Reduced Rules

In order to detect rule conflicts, we borrowed the idea presented in FortNOX of representing OpenFlow messages as *alias reduced rules* (ARR). An ARR is simply an expansion of

a rule's match headers along with its actions. This allows the controller to see the full effect the rule will have on flows moving through the switch, effectively incorporating both the input (the match) and the output (the effect of the actions) of a given rule. Our Floodlight module then compares candidate-rule ARRs and outgoing packets against a table of these ARRs stored in our application to determine if there is a conflict between an active rule in the switch's flow table and a candidate rule that an application is attempting to install. If no conflict is found then the candidate rule is allowed to be written to the switch. To illustrate a malicious rule that would be targeted by our application, we present a simple example of what FortNOX calls "dynamic-flow tunneling." Effectively, if one imagines a simple firewall implemented by a rule dropping all packets matching a given flow, a malicious application could insert a rule remapping that flow around the firewall rule, effectively circumventing it.

The firewall application installs a static flow rule upon switch connection that states:

$$(h1) \rightarrow (h8 : 80) \Rightarrow Action : drop \qquad (1)$$

In the attack, the subversive application dynamically installs a flow rule that will effectively allow traffic from `h1` to be delivered to `h8` by remapping the destination of packets being sent from `h1` to some dummy host `h7` to instead be delivered to the destination of `h2:80`. Clearly, `h1` can effectively communicate with `h8` by sending its packets to `h7` instead.

$$(h1) \rightarrow (h7 : 80) \Rightarrow Action : set(dst = h8), (port = 80) \qquad (2)$$

The ARR for rule 1 looks the same as the rule itself, combining its match with its actions. The ARR for rule 2 expands the rule into:

$$(h1) \rightarrow \{(h7 : 80)(h8 : 80)\} \Rightarrow Action : forward \qquad (3)$$

2.2 Detecting Conflict

Floodlight maintains a per switch set of ARRs that correspond to active rules in the switch's flow table. When an application attempts to add a rule to the switch's flow table, Floodlight creates an ARR for the candidate rule (cARR) and does a pairwise check of the candidate ARR with every ARR representing active rules in the switch's flow table (fARR). A table of rules currently in the switch is maintained in our Floodlight module. Rule timeouts are registered by using the OpenFlow callback functionality, which registers an event with the controller upon timeout. The conflict detection algorithm works as follows:

1. If the cARR and fARR have the same actions, then allow the candidate.
2. If the cARR and fARR have any intersection in their source sets, then take the union of the two source sets.
3. If the cARR and fARR have any intersection in their destination sets, then take the union of the two destination sets.
4. If both unions result in non-empty sets, then there is a conflict. Otherwise, allow the candidate rule and add cARR to the switch's set of active ARRs.

As an example, consider the ARRs we created above where the static firewall rule is currently in the switch (fARR) and the subversive application's remapping rule is the candidate (cARR):

1. The two ARRs have different actions. fARR = drop and cARR = forward. Therefore we need to check the source and destination sets of the two ARRs.
2. The two source sets intersect, therefore we take the union of the sets:
$$(h1) \bigcap (h1) \Rightarrow (h1) \qquad (4)$$
3. The destination sets intersect (h8:80), therefore we take the union of the sets:
$$(h8 : 80) \bigcap (h7 : 80)(h8 : 80) \Rightarrow (h7 : 80)(h8 : 80) \qquad (5)$$

This leaves the two sets non-empty, and Floodlight sees that these two rules are in conflict. In the case where a rule has wildcarded fields, the union includes the "widest" possible rule. For example, if the static firewall rule wanted to drop all traffic to `h8` regardless of destination port then the union would be:

$$(h8) \bigcap (h7:80)(h8:80) \Rightarrow (h7:80)(h8) \quad (6)$$

2.3 Critiques of FortNOX and Implementation Challenges

In designing our implementation of alias reduced rules for rule conflict avoidance, we encountered a number of apparent shortcomings and blind spots in the FortNOX implementation that would still allow a subversive application to insert conflicting rules without detection. Our critiques, and the corresponding challenges are detailed below.

Note that all of our critiques are based solely on the information provided in [5], as the reference implementation of FortNOX was never made public due to concerns about its viability and completeness.[1] A reference implementation of SE-Floodlight (secure Floodlight) by the same authors, has been proposed for release in summer 2013. While it is possible that SE-Floodlight will address our concerns, there is currently no implementation available on which to base more specific critiques.

2.4 Examining Packet Out

FortNOX focuses only examining and filtering out conflicting *flow rules* from being inserted into switches. However, packet rewriting and manipulation is not limited to flow rules inserted into the switches themselves. Rules can be inserted that, when matched, forward the packet to the controller. This could happen because a new flow has arrived at a switch for which there is not yet a corresponding flow rule, or because an idle or hard timeout means that a previous flow rule has been discarded from the switch. At this point, a subversive application can arbitrarily rewrite a packet *in the controller* in order to subvert rules, before sending it back to the switch to be forwarded. Even if another application in the controller installs a corresponding rule during the same session in the controller, the damage has already been done.

In order to defend against malicious flow subversion at the controller level, our implementation examines the actions of both attempted flow-table insertions, and of `PACKET_OUT` events. That is, we examine outgoing packets to see if they conflict with any of our alias reduced rules in our in-memory mapping of allowed actions. This prevents malicious actions by applications both at the flow-rule level and the controller level.

2.5 Malicious Tunneling Across Switches

One of the more substantial issues with the FortNOX approach is that it only verifies that a candidate rule is not in conflict with the current ruleset of a *single* switch's rules. Enforcement at the level of a single switch is suitible only to thwart the simplest attacks, such as the single-rule dynamic tunneling attack outlined in 2.1.

However, when a Floodlight application can install rules into multiple switches and/or manipulate packets flowing through multiple switches, comparing new rule insertions with each switch ruleset in isolation is not sufficient to detect conflicts. Consider a simple extension to the dynamic tunneling attack, as follows:

[1] We confirmed this in communication with FortNOX author Phil Porras.

References

[1] Floodlight openflow controller. `http://www.projectfloodlight.com`. Accessed: 2013-02-26.

[2] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev. 38*, 3 (July 2008), 105–110.

[3] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[4] Pfaff, B., Pettit, J., Amidon, K., Casado, M., Koponen, T., and Shenker, S. Extending networking into the virtualization layer. In *HotNets* (2009).

[5] Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., and Gu, G. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks* (New York, NY, USA, 2012), HotSDN '12, ACM, pp. 121–126.