

Securing the Floodlight OpenFlow Controller

Adam Vail and Rob Jellinek

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 susceptible 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 en-

forcement. Our implementation ensures that flow rules that exist within a switch are not violated. We detect if an application attempts to modify a switch's flow table with a conflicting rule and disallow it. We also force the controller to check all packet out transmissions against the active flow rules in a switch to ensure the controller does not allow unintentional "routing around" of flow rules. This prevents malicious applications from subverting security policy decisions put 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 Motivation

Gaining a full understanding of exactly what flow rules get inserted into a switch is very difficult from an administrators standpoint. Applications can be custom made by groups within an enterprise or from some other third party. With many applications needing to be run on a network, it isn't realistic for an administrator to know every detail of every application that is being run. There is no guarantee that any two applications, especially when created by a third party, don't logically overwrite each other's rules by accident. OpenFlow 1.0 specifies that switches must have support to detect and send an error message when there is an attempt to install a flow rule that has the same or overlapping headers as a rule currently in the flow table. This is only done when the ADD request to install the rule has the `CHECK_OVERLAP` flag set. Otherwise, unless the rules have exactly the same headers, the switch will allow the new rule to be installed "side-by-

side” with the old rule. When packets come into the switch, they will take the action of the most specific rule that they can match in the flow table. If the two rules have exactly the same headers than the new rule will overwrite the old rule.

Flow rules are installed in switches dynamically and are dependent on the traffic flowing through the network. This can make it extremely difficult for an administrator to investigate undesirable behavior in their network. The difficulty is exacerbated by the fact that rules timeout on a frequently. By the time an administrator inspects the flow table of a switch, its state could already have already changed.

Thus, it is extremely useful for an administrator to know when two or more applications are installing rules which logically conflict with each other. Since all rules are written to the switches using the controller, it is the controller which has a world view of all the flow tables. This allows the controller to make the decision if a rule should be allowed to be written to a switch or disallowed and flagged for administrator attention. An administrator can debug unintended network behavior faster and resolve conflicts between separate applications.

While unintentional misconfiguration of applications running on a controller are a definite possibility, there is also the case where an application author is being intentionally malicious. In order to illustrate this point we created a simple scenario with two conflicting applications. Figure figure number depicts our topology. While this is a somewhat contrived situation (due to the limited number of hosts and simplicity of the applications), the ideas can be extended to larger, real world applications.

Our first application acts like a simple “firewall” and installs a static rule in every switch to drop any traffic from **h1** to **h8** on port 80. The second application subverts the static rule by dynamically remapping **h1**’s flow to be redirected towards **h8**. This is done by having **h1** send packets towards a dummy host that is connected to the same switch as **h8** using port 80 (the port number is arbitrary, it could be any high number port as long as the subversive application and the sender are in agreement). When **h8**’s switch

sees packets destined for the dummy host it dynamically installs a rule to remap the destination of those packets to **h8** on port 80. Since a flow can only match a single rule in a flow table, the switch remaps the packets’ destination and forwards them on to **h8** without ever checking them against the firewall’s static rule. As with network debugging, the controller has a complete view of network and is therefore in a position to arbitrate which rules should be installed on a switch and which should be disallowed due to logical conflicts with existing rules.

3 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 concerning which rules an application is allowed to install on a switch and which rules should be disallowed. 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 motivated us to extend Floodlight. We also discovered several shortcomings of FortNOX which we addressed in our Floodlight implementation.

3.1 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 ARR 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:

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

$$(h1) \rightarrow (h8 : 80) \Rightarrow Action : drop \quad (2)$$

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.

$$(h1) \rightarrow (h7 : 80) \Rightarrow Action : set(dst = h8), (port = 80) \quad (3)$$

$$(h1) \rightarrow (h7 : 80) \Rightarrow Action : set(dst = h8), (port = 80) \quad (4)$$

$$(h1) \rightarrow (h7 : 80) \Rightarrow Action : set(dst = h8), (port = 80) \quad (5)$$

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:

$$(h1) \rightarrow \{(h7 : 80)(h8 : 80)\} \Rightarrow Action : forward \quad (6)$$

The ARR for rule 1 looks essentially the same as the rule itself. The ARR for rule 2 expands the rule into:

$$(h1) \rightarrow (h7 : 80)(h8 : 80) \Rightarrow Action : forward$$

3.2 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 ARR_s have different actions. fARR = drop and cARR = forward. Therefore we need to check the source and destination sets of the two ARR_s.
2. The two source sets intersect, therefore we take the union of the sets:

$$(h1) \cap (h1) = (h1)$$

3. The destination sets intersect (h8:80), therefore we take the union of the sets:

$$(h8 : 80) \cap (h7 : 80)(h8 : 80) = (h7 : 80)(h8 : 80)$$

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:

$$(h8) \cap (h7 : 80)(h8 : 80) = (h7 : 80)(h8)$$

3.3 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.

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

3.4 Examining Controller-Based Rule Evasion

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.

3.5 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 rule-set of a *single* switch. Enforcement at the level of a single switch is sufficient only to thwart the simplest attacks, such as the single-rule dynamic tunneling attack outlined above in 2.1. However, when a Floodlight application can install rules into multiple switches and/or manipulate packets flowing through multiple switches, installing a candidate ARR requires verifying that its installation would not create a flow that would subvert any existing flow restrictions throughout the

network. In other words, to adequately protect the *network*, enforcement must be considered on the level of the whole network, not just a single switch.

Consider a simple extension to the dynamic tunneling attack, where the legitimate rule is once again $(h1)(h8 : 80) : DROP$. Then the subversion rules we be as follows:

$$Sw1 : (h1) \Rightarrow (h6) : set(src = h6, dest = h7) \quad (7)$$

$$Sw1 : (h7) \Rightarrow (h6) : set(src = h6, dest = h1) \quad (8)$$

$$Sw2 : (h6) \Rightarrow (h7) : set(dest = h8) \quad (9)$$

$$Sw2 : (h8) \Rightarrow (h6) : set(src = h7) \quad (10)$$

Assuming the malicious application installs each of these rules in turn, each is considered as a candidate ARR (cARR) and checked pairwise against existing rules according to the procedure described in Subsection 2.2 above. Recall that the algorithm builds the union of source and destination rules and actions between the cARR and each fARR in the flow table, and detects a conflict if actions differ and the intersection of the sources and destinations of the ARRs are non-empty. In this case though, testing the insertion of the rules into Sw1 would yield a empty destination set since $h8$ does not appear in the cARRs' destination sets. Similarly, $h1$ does not appear in the cARRs' source sets for the rules inserted into Sw2, so they would be inserted successfully.

Our implementation follows that of FortNOX in blocking the insertion of conflicting rules on a single switch. However, blocking the insertion of malicious dynamic tunneling across switches in a large network and with an arbitrary topology does not admit a simple solution. One approach might be to simply follow....

4 Related Work

As mentioned earlier, much of our work was inspired by FortNox. Although, there have been

other approaches to security in software defined networks as well.

Ethane [?] was a predecessor and driving force behind the development of software defined networking as we see it today. It was born out of the need to be able to manage and enforce fine-grained policies across an entire network. The design of Ethane consists of a central controller and dumb switches, which are the building blocks of SDN today. This system was designed with security in mind since enforcing security in an enterprise network is essentially a set of policies that need to be managed across the entire network. Ethane provides ease of management for network policies, but it does not address how a controller is to protect against conflicting policies introduced by administrators and third party applications.

FlowVisor [?] allows for a single physical network to be divided into logical "slices." Each slice received its own controller and therefore runs its own set of application of the controller. This provides isolation between the different applications and guarantees that they do not misbehave with each other. While this is desirable, it is not practical to expect an enterprise to create a new network slice, and thus a new controller, for every third party application. The management of these different entities could get very complicated very quickly. Also, FlowVisor does not provide any security within a slice. Therefore, if there is a need to run several applications on the same network slice, there is no guarantee that they will cooperate with each other.

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.