The Smaller, the Shrewder: A Simple Malicious Application Can Kill an Entire SDN Environment

Seungsoo Lee
KAIST, Republic of Korea
lss365@kaist.ac.kr

Changhoon Yoon
KAIST, Republic of Korea
chyoon87@kaist.ac.kr

Seungwon Shin
KAIST, Republic of Korea
claude@kaist.ac.kr

ABSTRACT

Security vulnerability assessment is an important process that must be conducted against any system before the deployment, and emerging technologies are no exceptions. Soft ware-Defined Networking (SDN) has aggressively evolved in the past few years and is now almost at the early adoption stage. At this stage, the attack surface of SDN should be thoroughly investigated and assessed in order to mitigate possible security breaches against SDN. Inspired by the necessity, we reveal three attack scenarios that leverage *SDN application* to attack SDNs, and test the attack scenarios against three of the most popular SDN controllers available today. In addition, we discuss the possible defense mechanisms against such application-originated attacks.

Keywords

Software-defined Networking; Attack; Security

1. INTRODUCTION

Software-Defined Networking (SDN) has aggressively grown over the past few years, and to date, there are many SDN controller products available on the market and the industry is even participating in open-source SDN controller projects. These SDN controllers are often referred to as network operating systems (NOS), because most cutting-edge controllers expose useful northbound APIs to enable the network application ecosystem in addition to managing the entire SDN environment in a centralized manner. ONOS and OpenDaylight, for instance, are popular open-source NOSes that are supported by the major companies, who are seriously considering the adoption of SDN.

When considering the adoption, the security of SDN components is obviously a mandatory aspect that should be thoroughly investigated, and accordingly, such area of study is gradually gaining attention. Kreutz et al. have introduced seven possible attack vectors against SDN [7]; however, they have not verified if these vectors are feasible against real SDN controller implementations. Meanwhile, there have been a few practical efforts to reveal and mitigate the security problems in real SDN controller implementations as well. For example, Shin et al. [16] revealed architectural weaknesses in POX and Floodlight controller and proposed a secure controller architecture that fundamentally eliminates the weaknesses. Furthermore, Hong et al. [4] unveiled the vulnerabilities in network element authentication mechanisms of OpenDaylight controller and contributed eliminating the potential threat.

As mentioned, despite the increased attention to the security of SDN, the attack surface of SDN is still understudied, and SDN controllers are prone to the security vulnerabilities that have not been disclosed yet. Hence, similar to the previously mentioned efforts, we also disclose three security vulnerabilities in three different SDN controllers: Floodlight, OpenDaylight (ODL) and Open Networking Operating System (ONOS). Here, we specifically attempt to reveal the design flaws, which are potentially powerful enough to put the entire network at risk, in the application platform of the controllers. In order to prove the feasibility and to examine the potential impact of each vulnerability, we introduce and demonstrate the attack scenarios in the real SDN environments.

The remainder of this paper is structured as follows. In Section 2, we describe how SDN application has become a serious attack vector that could potentially affect the entire SDN environment to emphasize the importance of the application layer security. Then, we demonstrate the attack scenarios against three different SDN controllers to show the effectiveness of each attack in Section 3. In Section 4, we discuss the possible defense mechanisms that protect the application layer from our attacks. Finally, we discuss related work in Section 5, and conclusion is described in Section 6.

2. BACKGROUND AND MOTIVATION

This section provides an introduction to network operating system, and motivating examples to illustrate how an SDN application might put the managed network at risk.

2.1 Network Operating System

As briefly mentioned in the previous section, a network operating system (NOS) manages the entire SDN environment as the control plane. Figure 1 illustrates the key components of a typical network operating system (NOS), and the components include the northbound interface, core services, internal storage and southbound interface. Obviously, the actual implementations of NOSes are different from each other; however, most of the well-known NOSes, such as OpenDay-

SDN-NFVSec'16, March 11 2016, New Orleans, LA, USA

DOI: http://dx.doi.org/10.1145/2876019.2876024

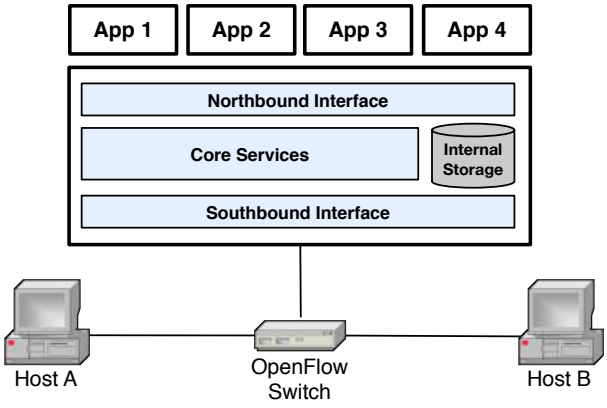

Figure 1: Network operating system architecture

light [1] and ONOS [10], are implemented according to such a typical architecture.

Specifically, the core services generally implement and provide essential SDN functions, including network flow and statistics management functionalities, to SDN applications via the northbound interfaces. The southbound interfaces are the data plane abstractions, while the internal storage collects and maintains useful and diverse network information for the components of the core and application layers.

In summary, not only successfully managing SDNs in a centralized manner, most NOSes implement a dynamic and flexible network application platform as if they were the traditional operating systems such as Windows or Linux.

2.2 Motivation

The crucial mission of SDN is to provide high-level network abstraction and programmability, and hence, SDN architecturally decouples the control plane from the data plane and expose high-level network abstractions to the application layer. Leveraging the abstractions, or Northbound APIs, anyone can easily develop and distribute SDN applications that implement useful and innovative network functions in the same way that one would develop applications for any other operating systems.

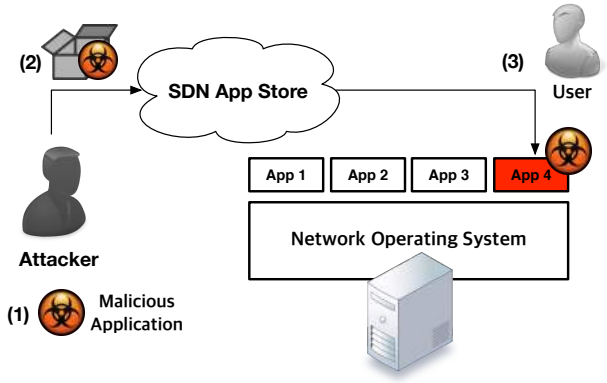

Figure 2: An attack scenario that involves an adversary distributing malicious SDN applications to compromise a network operating system

Accordingly, just like any other operating systems, NOSes might also have the vulnerabilities that are actually exploitable. Before we elaborate on the vulnerabilities (Section 3), we introduce an effective attack scenario that demonstrates how a malicious SDN application, which exploits the vulnerability of an NOS to compromise the NOS itself or the managed network, can be installed on the victim's system as shown in Figure 2. Before we elaborate the scenario, we assume that there is an SDN application store for a target SDN controller (e.g., HP SDN Store [5]). The reason the application store can exist is that by exposing their APIs to the outside, they give an opportunity to freely produce useful applications to developers, and users can dynamically load the useful applications into the SDN controller.

First, (i) an attacker implements a load balancer application that hides a malicious behavior such as sniffing network topology. Then, (ii) he uploads that application to the SDN app store. (iii) A user searches a load balancer application from the SDN app store, and downloads it. Finally, the user installs the SDN application, and executes it. The malicious application seems like a benign load balance application, but it plays malicious acts without any restriction.

The reason the malicious application can be installed and act is that there is no guideline about malicious behaviors of SDN applications. In addition, most SDN controllers do not adopt an inspection mechanism for SDN applications before executing them. On that point, Christian et al. introduced an SDNRootkit, which hides some malicious behaviors from the target SDN controller [14]. With the possibility, we will show three attack cases for three well-known SDN controllers in the next section.

3. ATTACK CASE

In this section, we present three attack scenarios against three well-known open-source SDN controllers; (i) Floodlight, (ii) ONOS and (iii) OpenDaylight. In order to verify the effectiveness of each attack scenario, we have tested each one in a real SDN environment.

3.1 Assumption

As discussed in Section 2, there are diverse vectors that malicious SDN application could be deployed to the victim controllers because the modern controllers do not authenticate the applications nor perform application analysis prior to the deployment. Hence, in our attack scenarios, we assume that a network administrator has already installed malicious applications. Such a malicious application delivery scheme has been introduced and discussed in previous work [14] as well.

3.2 Floodlight Case

When a *PACKET_IN* message arrives at the Floodlight controller, the controller passes the message to the applications that are interested in such type of message. When passing the message to the applications, the controller does not just broadcast the message to all the applications, rather it sequentially passes the message to each application in a certain order. During this process, (i) the message reception order of applications should never be manipulated by any entity other than the trusted ones (e.g., the core services), and (ii) the integrity of *PACKET_IN* messages should be guaranteed.

Floodlight, however, does not guarantee the integrity of the message reception order of applications nor the control message, and a malicious application may leverage such vulnerability to launch an attack against the controller itself or the network. We introduce a working attack scenario that a malicious application exploits this vulnerability to manipulate the network behavior (Figure 3).

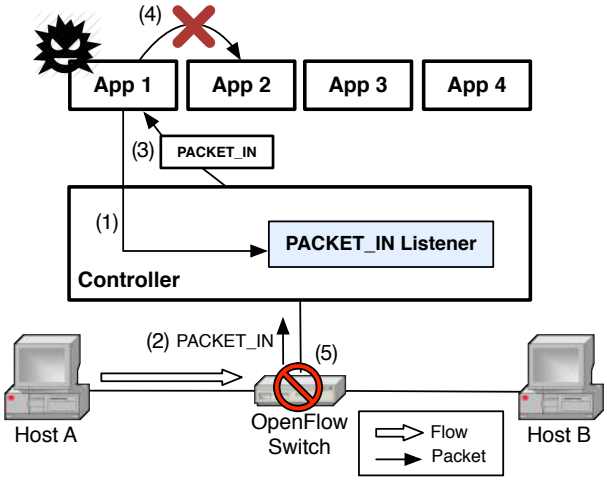

Figure 3: The steps of the Floodlight attack case

1. A malicious application (App 1) accesses the PACKET_IN listener of Floodlight and modifies the order of the listeners so that the listener of the malicious one could be the first one to receive PACKET_INs.
2. Then, Host A attempts to communicate with Host B, and accordingly, the OpenFlow switch sends a PACKET_IN to the controller.
3. The PACKET_IN listener, manipulated by App 1, checks the order, and then passes the message to App 1 before passing it to any other applications.
4. Once the malicious application receives the message, it removes the entire payload value of the message and hands the forged message over to the next application (App 2). However, since the payload field has been wiped out by the malicious application, the next application that tries to access the payload throws an (unexpected) exception.
5. Since the controller cannot handle the exception properly, the switch is disconnected from the controller, and as a result, Host A fails to communicate with Host B.

In order to verify if such an attack scenario is effective against the real SDN controllers, we implemented a Floodlight application that launches this attack. Like any other SDN controllers, Floodlight provides rich northbound services, and one of the services, called `IFloodlightProviderService`, allows an SDN application to not only add itself to the list of the PACKET_IN listeners, but also access and manipulate the ordered list of the PACKET_IN listeners. Leveraging such capabilities of the service, it is fairly easy to implement the malicious application described in Figure 3. We implemented the application to perform these tasks during the application initialization phase so that the application can automatically become the first one to receive PACKET_IN messages whenever it is installed. Furthermore, `IFloodlightProviderService` even implements the functionality to manipulate the control message contents, and to implement the malicious application, we leverage `remove()` method to eliminate the payload of each PACKET_IN message, and let other applications to process the message by simply returning `Command.CONTINUE` value.

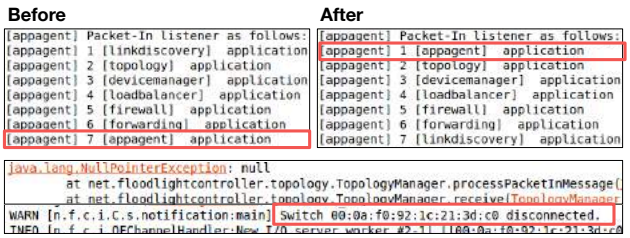

Figure 4: The result of the Floodlight attack case

Figure 4 illustrates the impact of the actual attack launched by our application. As shown in Figure 4 (Before), our application (appagent) is the last application to receive PACKET_IN messages; however, once the application has manipulated the list of PACKET_IN listeners, it becomes the first one on the list as shown in Figure 4 (After). Then, the application removes the payload from all the PACKET_INs received, and it passes the manipulated message to the next application, which is `TopologyManger` in this case. When the topology application attempts to access and process the message, it fails to handle the nullified payload value, and thus throws a NullPointerException, which leads the controller to disconnect the switch that sent the message.

3.3 ONOS Case

Compared with Floodlight, ONOS provides more advanced features to improve the SDN environment in many ways. For example, ONOS allows users to dynamically and programmatically configure various ONOS components via one of the Northbound services, called `ComponentConfigService` to enhance the configurability of ONOS. This feature is especially useful when users need to adjust the properties of various ONOS components (e.g., threshold values, switches to enable certain features) to fulfill different network requirements, and it allows the users to accomplish this without reimplementing each component.

Meanwhile, this feature introduces a new security threat to the network. Noting that minor change in the configuration of an ONOS component may completely change the network behavior, such tunable parameters should only be manipulated by the trusted entities via `ComponentConfigService`. Here, we show that such unrestrictedly configurable SDN controller environment may put the network at serious risk. The possible attack scenario is illustrated in Figure 5.

1. A malicious application (App 1) accesses the service configuration management among available services in ONOS. The malicious application gets the property list of a target application.
2. Then, the malicious application manipulates a certain property of the target application through a configuration manager that is one of ONOS services.
3. Host A sends a packet to Host B, but there is no a

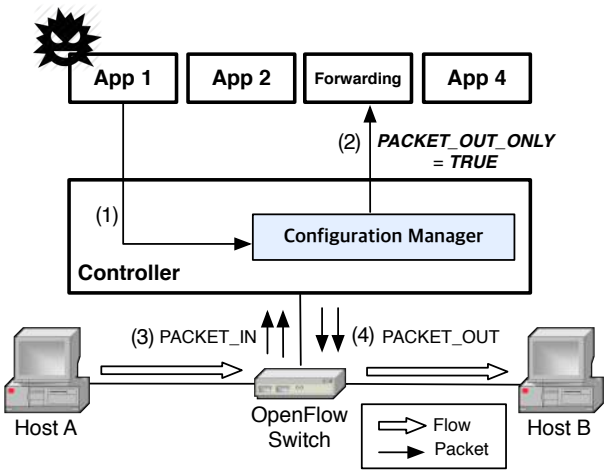

Figure 5: The steps of ONOS attack case

matched flow rule for the request. Therefore, a switch generates a PACKET_IN message and sends it to the ONOS controller.

4. However, the target application cannot make a flow rule for that message to the switch because the property is changed. Therefore, the target application only sends PACKET_OUT message to the switch.

We implemented a malicious ONOS application that attacks a network as shown in Figure 5. In the actual implementation, we attempt to manipulate the configuration of the `ReactiveForwarding` application via `ComponentConfigService`; specifically, we set the PACKET_OUT_ONLY parameter value to be *true*. Setting this parameter might be useful in some certain networks with special requirements; however, in our experiment, we aim to abuse this tunable parameter to degrade the overall performance of the target network.

Before

```
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=1.05 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=1.00 ms
64 bytes from 10.0.0.2: icmp_seq=13 ttl=64 time=1.00 ms
64 bytes from 10.0.0.2: icmp_seq=14 ttl=64 time=1.02 ms
64 bytes from 10.0.0.2: icmp_seq=15 ttl=64 time=1.01 ms
```

After

```
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=4.42 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=4.28 ms
64 bytes from 10.0.0.2: icmp_seq=13 ttl=64 time=4.57 ms
64 bytes from 10.0.0.2: icmp_seq=14 ttl=64 time=3.98 ms
64 bytes from 10.0.0.2: icmp_seq=15 ttl=64 time=4.78 ms
```

Figure 6: The result of ONOS attack case

In order to compare the network performance before and after the attack, we measure the network latency by running ping tests on a network host as shown in Figure 6. After the attack, since the ReactiveForwarding application forwards the traffic on a packet-by-packet instead of flow-by-flow basis, every single packet on the network experiences the control path delay, ultimately degrading the overall network performance. Furthermore, this attack also adds a significant burden on the controller and other SDN applications because it fundamentally leads the network devices to generate a PACKET_IN message for each network packet.

3.4 OpenDaylight Case

Similar to other SDN controller implementations, OpenDaylight also provides diverse northbound services to encourage the developers to implement useful and innovative network functions, and applications running on OpenDaylight can easily register services or directly call them to use their network functions. For example, if an application wants to see whether a certain host is in a network, it registers the `IHostFinder` service called by `HostTracker` (registering services). If an application wishes to decode data packets, it set `DataPacketService` in the `Activator` (using services). In addition, applications can make dependencies on specific services. Similar to previous attack cases, Since OpenDaylight allows applications to change the services of other applications dynamically without constraint, the attacker can also leverage this ability. The operational scenario of this case is presented in Figure 7.

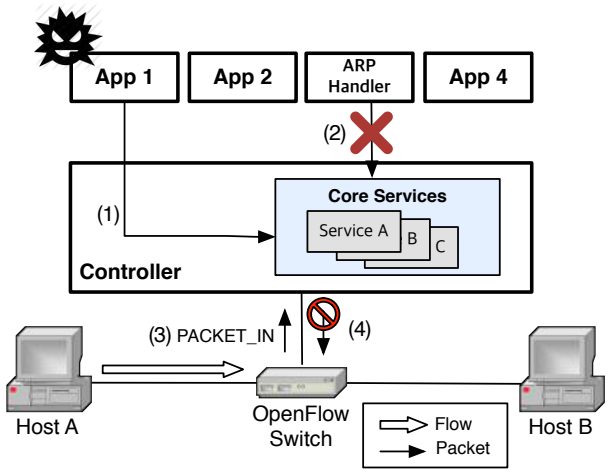

Figure 7: The steps of OpenDaylight attack case

1. A malicious application (App 1) accesses a core service management of OpenDaylight, and gets the information about the services used by a target application.
2. Then, the malicious application removes all services used by the target application so that the target application cannot use any service provided by the controller.
3. After the malicious application removes those services, Host A sends a packet to the Host B. But, there is no matched flow rule, and the switch sends PACKET_IN message to the controller.
4. However, the target application cannot handle PACKET_IN message, so the controller does not understand topology information about the network.

Our target application is `ArpHandler` that is a default application running on OpenDaylight and manages ARP request and reply packets. We implemented a malicious application for the scenario. The malicious application gets the registered services of the target application in `Activa-`

tor class through `getServiceProperties()`. Then, the malicious application calls `unregister()` to delete the whole services of the target application.

The ArpHandler registers three services: `IHost Finder`, `IListenDataPacket`, and `ICacheUpdateAware` as shown in Figure 8. Using these services, the ArpHandler can manage ARP packets passing a network. In the case that a malicious application tries to unregister those services, it can successfully unregister all the services of the ArpHandler. Then, the ArpHandler cannot receive any messages from the controller, and thus it cannot provide any useful information to other applications. Since ARP packets play a role as an initiator for network communications, the network managed by the controller loses its functionality.

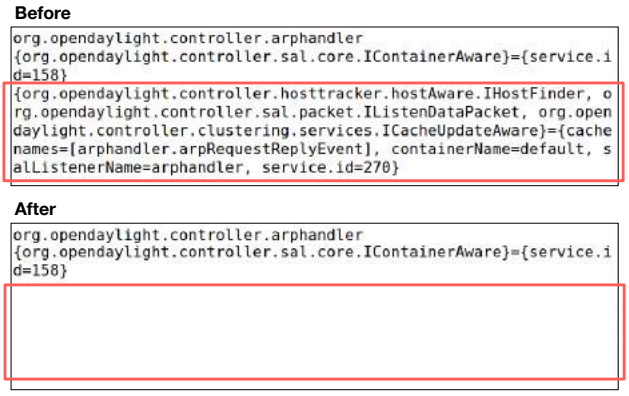

Figure 8: The result of OpenDaylight attack case

4. DEFENSE

The reason the attack cases that are introduced in Section 3 are probable is that SDN applications running on each controller have too powerful authority without any restriction. Therefore, if a malicious SDN application is installed on the target controller, it is possible to even subvert the target SDN environment.

Here, we introduce two defense mechanisms that could protect such attacks; (i) permission checking and (ii) static or dynamic analysis.

4.1 Permission Checking

There are some studies about the policy checking for SDN application [9, 12, 16, 11]. Among them, Noh et al. suggested a permission checking model for an SDN controller [9]. First, they defined five states on which a network application can arrive based on OpenFlow protocol version 1.1. Also, they described two primary permission sets for SDN applications. One is the OpenFlow message permission set, and the other permission set is the controller resource permission set. OpenFlow message permission set has a list each type of OpenFlow message as a permission rule. The controller resource permission set means whether an application can access resources managed by the controller such as an internal storage or not.

Figure 9 shows the example operation for permission checking model. Each application has their own permission file, which each application has to specify what it uses, similar to Android applications. For example, an application (App1) states that they need an authority to install flow rule so that it can send FLOW_MOD message to the switch. The network administrator can check the permissions that applications have, which are downloaded from untrusted sources. In addition, they implemented an additional layer on Floodlight controller and it can monitor whether the applications do disallowed permission.

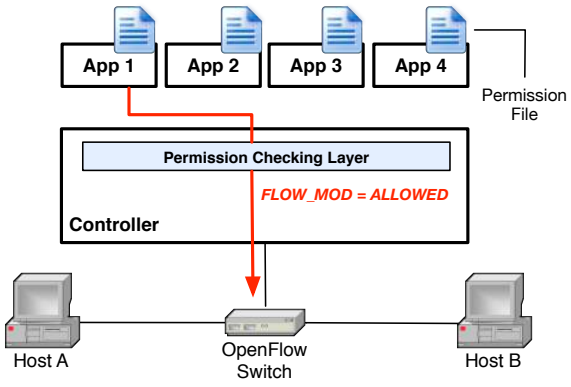

Figure 9: Permission checking layer

4.2 Static/Dynamic Analysis

Only adopting permission checking mechanism may be insufficient for making the network environment secure. Another method to examine whether an SDN application is secure or not is static or dynamic analysis for SDN applications. Before the network administrator downloads and executes SDN applications, the applications can be analyzed statically or dynamically manner. This mechanism is not new, but it is needed to secure the network environment.

For static analysis of SDN applications, the source code of a target application should be provided. Then, we need to calculate the Control Flow Graph (CFG) (e.g., Soot [8]) and get API calls list. Based on that information, we could make a decision whether the target application is malicious or not. Also, for dynamic analysis to be effective, the target application must be executed with sufficient test inputs (e.g., control messages such as PACKET_IN) to produce interesting behavior. With software testing measures such as code coverage, we could observe an adequate slice of the applications of possible behaviors.

Before those analyses for SDN applications, there should be a definition of malicious behavior for each controller because each implementation of the controller is different. However, the problem is that there is insufficient information about the malicious behavior for SDN applications.

5. RELATED WORK

There have been several studies [6, 2, 15, 4, 14] that focus on attack possibility of SDN. Benton et al. point out that failures due to lack of TLS adoption by vendors for the OpenFlow control channel because of the performance issue can make attack vectors such as man-in-the-middle attack and denial of service attack available [2]. Kreutz et al. argue that the new features of SDN such as a centralized controller and programmability of managing network environment introduce new threats and address possible threat

vectors that make SDN vulnerable [6]. Shin et al. discuss the feasibility of attacking SDN by introducing flow table saturation attacks [15]. Hong et al. address the feasibility of compromising network topology services of controllers that can cause serious problems such as black hole routing and making fake link [4]. Christian et al. implemented an SDNRootkit, which hides some malicious behaviors from a target SDN controller [14]. The SDNRootkit can install malicious flow rules and get the network information managed by the SDN controller. However, these malicious behaviors can be feasible by Java reflection not vulnerabilities of a target SDN controller.

In addition, some researchers have directly indicated there are specific issues such as policy conflict, permission checking, access control, and sharing relationships [13, 16, 3, 12, 11]. Porras et al. argue that a flow rule generated by an SDN application is possible to conflict with other flow rules. Then, the conflicted rules can make a new rule that is possible to evade application policies [13]. Shin et al. introduce a robust and secure controller, which is called Rosemary, to prevent applications from vulnerabilities that can cause controller corruptions [16]. Chandrasekaran et al. address the reliability and fault tolerance problems of SDN as showing the possibilities of controllers and network failures due to SDN applications [3]. Porras et al. have proposed a security-enhanced version of Floodlight, which has some key features that include authentication, role-based authorization, a permission model for the data plane access, flow rule conflict resolution and more. Also, SM-ONOS controller (i.e., Security-Mode ONOS) provides APIs permission checking mechanism [11].

6. CONCLUSION

In this paper, we describe three attack cases on well-known controllers (i.e., ONOS, OpenDaylight and Floodlight). Those attack cases may be critical to the SDN environment. In addition, each controller has different implementation concept, so there are many malicious behaviors and vulnerabilities depending on controller type. Therefore, the network administrator has to inspect the security of the controller and SDN applications before they construct SDN environment.

We hope that the vulnerability cases described in Section 3 will contribute to communities and vendors who wish to make their SDN products more secure and to the industry who want to verify them before the adoption or deployment stage.

Acknowledgments

This work was supported by the ICT R&D program of MSIP/ IITP, Republic of Korea [B0190-15-2011, Korea-US Collaborative Research on SDN/NFV Security/Network Management and Testbed Build].

7. REFERENCES

[1] A Linux Foundation Collaborative Project. OpenDaylight SDN Controller. http://www.opendaylight.org.

[2] K. Benton, L. J. Camp, and C. Small. Openflow vulnerability assessment. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN'13)*. ACM, 2013.

[3] B. Chandrasekaran and T. Benson. Tolerating sdn application failures with legosdn. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets'14)*. ACM, 2014.

[4] K. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[5] HP. Hp sdn dev center: Sdn app store. http://www.hp.com/go/sdndevcenter.

[6] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1):14–76, 2015.

[7] D. Kreutz, F. M. V. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013.

[8] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[9] J. Noh, S. Lee, J. Park, S. Shin, and B. B. Kang. Vulnerabilities of network os and mitigation with state-based permission system. *Security and Communication Networks*, 2015.

[10] ON.LAB. Open network operating system. http://http://onosproject.org/.

[11] ON.LAB. Security-mode onos. https://wiki.onosproject.org/display/ONOS/Security-Mode+ONOS.

[12] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran. Securing the software-defined network control layer. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS), San Diego, California*, 2015.

[13] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks (HotSDN'12)*, 2012.

[14] C. Röpke and T. Holz. Sdn rootkits: Subverting network operating systems of software-defined networks. In *Research in Attacks, Intrusions, and Defenses*, pages 339–356. Springer, 2015.

[15] S. Shin and G. Gu. Attacking software-defined networks: A first feasibility study (short paper). In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013.

[16] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, November 2014.