H. Langweg, M. Meier, B.C. Witt, D. Reinhardt (Hrsg.): Sicherheit 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 95

SDN Ro^{2}tkits: A Case Study of Subverting A Closed Source SDN Controller

Christian Röpke[1]

Abstract: An SDN controller is a core component of the SDN architecture. It is responsible for managing an underlying network while allowing SDN applications to program it as required. Because of this central role, compromising such an SDN controller is of high interest for an attacker. A recently published SDN rootkit has demonstrated, for example, that a malicious SDN application is able to manipulate an entire network while hiding corresponding malicious actions. However, the facts that this attack targeted an open source SDN controller and applied a specific way to subvert this system leaves important questions unanswered: How easy is it to attack closed source SDN controllers in the same way? Can we concentrate on the already presented technique or do we need to consider other attack vectors as well to protect SDN controllers?

In this paper, we elaborate on these research questions and present two new SDN rootkits, both targeting a closed source SDN controller. Similar to previous work, the first one is based on Java reflection. In contrast to known reflection abuses, however, we must develop new techniques as the existing ones can only be adopted in parts. Additionally, we demonstrate by a second SDN rootkit that an attacker is by no means limited to reflection-based attacks. In particular, we abuse aspect-oriented programming capabilities to manipulate core functions of the targeted system. To tackle the security issues raised in this case study, we discuss several countermeasures and give concrete suggestions to improve SDN controller security.

Keywords: Software-defined networking; SDN controller security; SDN rootkits

1 Introduction

Academia and industry have brought forward Software-Defined Networking (SDN) for quite a long time already. The key driver was (and probably is) the OpenFlow protocol [Mc08] which was introduced in 2008. From then on, many studies have been presented which vary from OpenFlow improvements [Opa], numerous SDN controllers [Gu08, Mc, Fl, Opb, ON, Yo17], network simulation software [LHM10], hardware switches with OpenFlow support to various SDN applications [HPa]. In addition, SDN is on the big player's business agenda including network vendors such as Cisco and Juniper, software vendors such as Microsoft and Oracle, network operators such as Verizon and Deutsche Telekom, and data center operators such as Google and Facebook. Major reasons for this are the ability to solve pressing network

[1] Ruhr-Universität Bochum, Lehrstuhl für Systemsicherheit, Universitätsstrasse 150, 44801 Bochum, NRW, christian.roepke@rub.de

doi:10.18420/sicherheit2018_07

problems easier than in the past [FRZ13], the potential to reduce costs [Ho12, Va17], and opportunities to improve network security [Sh16].

In software-defined networks, SDN controllers play a crucial role [Op13, Op14]. They primarily connect network hardware (aka. SDN switches) and network software which is responsible for making forwarding decisions (aka. SDN applications). Programming SDN switches is performed either proactively by inserting flow rules into an SDN switch or reactively by reacting on packets, which are delegated to an SDN controller because of a missing flow rule. For the benefit of SDN applications, SDN controllers provide a global view of the network and allow operating on it in an abstract manner. Obviously, this control system is of high interest for an attacker as compromising it would allow network-wide adverse manipulations. As recent work [RH15b] concentrates on open source SDN controllers, it is still unclear whether closed source systems are affected equally serious. It is also uncertain if simply blocking Java reflection for SDN applications would solve the problem, or if an attacker can implement SDN rootkits using other techniques.

To complement existing work, we present two new SDN rootkits and use them as demonstrators to discuss these open issues. The first one is based on Java reflection which is enabled by default even for closed source SDN controllers. The second one takes advantage of Aspect-Oriented Programming (AOP) which must not be supported by an SDN controller in its default settings. However, we demonstrate how an attacker can add required AOP software via bypassing a corresponding security mechanism (i. e., signature validation). In order to run these attacks, we target the HP VAN SDN controller [HPb] as it is supposed to provide robust security [HPc] while several release generations indicate a certain degree of maturity.

2 Background

2.1 Java Reflection in SDN Rootkits

Java is an object-oriented programming language which includes that access to class internals can be restricted. By choosing an access modifier like *public* or *private*, a developer can specify which other classes can have access to a variable or method of a class. Especially, private variables and methods are not supposed to be accessible by objects of other classes. However, Java also provides a mechanism called reflection [Ora] which allows to bypass this mechanism. The already presented SDN rootkit [RH15b] takes advantage of this mechanism and modifies private fields as well as invokes private methods. In particular, private field manipulation was used for both removing rootkit artifacts from internal inventories and replacing internal services by malicious versions. In the latter case, source code of OpenDaylight was used to re-implement internal services with malicious filtering functions. To replace such an internal service, an object of a modified version was created which was then set to a private variable which was supposed to point to the original service.

In addition, private methods were invoked in order to bypass internal cache updates which are supposed, among others, to keep internal flow rule databases up-to-date. By skipping such updates, the SDN rootkit was able to hide manipulations from these databases, which are used by internal services as well as by SDN applications to monitor network changes.

2.2 Aspect-Oriented Programming

Aspect-oriented programming is a programming paradigm which complements object-oriented programming by enabling cross-cutting concerns [Ki97]. It brings flexibility and is especially capable of improving security which is in fact a cross-cutting concern [DWJP05, PS08, VBC01, DWVDD02]. So-called aspects act similar to classes while containing, among others, so-called advices and pointcuts. An advice specifies the code manipulation and a pointcut defines a point in the program flow where such an advice is supposed to be applied. A typical use case for AOP is adding log functionality, for example, to observe when a certain method is called and what arguments it receives. Instead of modifying all classes which use such a method, AOP enables to define an aspect which modifies all the existing code at once. For example, additional code can be executed before or after a method of interest. In case of Java, code manipulation is typically implemented on the byte code level by weaving manipulations either into already compiled classes (compile-time weaving) or while loading a class (load-time weaving). Thus, AOP for Java allows to modify closed source systems such as SDN controllers, for example, by adding security checks before accessing critical controller functions.

2.3 Attacker Model

Throughout the rest of this paper, we assume an attacker which is able to install a malicious SDN application. This can be achieved in several ways: (i) an attacker can lure administrators into installing such an application, (ii) a security vulnerability can be exploited to bypass the security mechanism taming such applications as a whole [Orb, Orc] or in parts [Orf, Ore, Ord], and (iii) an attacker can steal a certificate in order to correctly sign a malicious application[FMC11]. In addition, we allow this malicious SDN application to use SDN controller services in the same way other SDN applications can do. This includes, for example, reading the network topology (e. g., to look for an interesting target) and re-programming network devices (e. g., to manipulate the network).

3 Subverting the HP Controller

An SDN rootkit faces several challenges with respect to its primary goal, i. e., hiding its existence. First, it is obliged to hide the components which become visible during its

installation. For instance, SDN controllers typically add information to internal inventories such as a list of installed SDN applications and a list of applications, which are allowed to handle network packets. In addition, the HP controller provides a protection mechanism which is supposed to prevent the installation of unwanted SDN applications. In particular, an SDN application's signature is validated during the installation procedure in order to prevent unsigned software.

Second, an SDN rootkit typically wants to hide malicious network manipulations such as added malicious flow rules. As providing the current network state is a core function of SDN controllers, also the HP controller provides a corresponding interface. An attacker must manipulate this service in order to hide the presence of adverse network manipulations, for example, from a monitoring application.

Third, network manipulations typically trigger network events which are observed by SDN controllers. As this can reveal malicious network manipulations, an attacker is obviously interested in hiding corresponding artifacts as well. In case of OpenFlow, for example, *flow_mod* messages are generated when (re)-programming SDN switches and *flow_removed* events are sent to inform SDN controllers about flow rule removals. In order to monitor such network events, the HP controller provides two separate listener services. On the one hand, a so-called *flow listener* allows to monitor flow_mod OpenFlow messages. On the other hand, a so-called *message listener* enables the observation of various messages including flow_mod and flow_removed.

Fourth, we face additional challenges with respect to closed source SDN controllers. On the one hand, it is more difficult to understand the internal functioning of an SDN controller. As this is particularly important to identify interesting spots within an SDN controller, it can complicate subverting critical functions significantly. On the other hand, abusing Java reflection in the way used by previous work [RH15b] does not work anymore. For example, replacing a controller component by implementing a malicious version which is based on the SDN controller's source code (see Section 2.1) is not possible. This reflection-based technique was, however, essential to hide critical SDN rootkit artifacts.

3.1 Abusing Java Reflection

As mentioned before, our first SDN rootkit uses Java reflection to subvert the HP controller. As solving aforementioned challenges varies in difficulty, we describe this rootkit's solutions depending on its difficulty and start with the most difficult one. The most challenging parts are to understand the internal functioning of the HP controller and to find suitable spots, which must be manipulated in order to hide malicious network manipulations. Based on the HP controller's programming guide, its API documentation, and by means of decompiling byte code, an attacker is able to solve these challenges despite the controller's complexity. For example, the documentation says that *getFlowStats()* is responsible for listing installed flow rules and that it is provided by a so-called *ControllerService*, which is implemented

by a class called *ControllerManager.class*. To list interesting private fields of this class (e. g., *FlowTrk flowTrk* and *ListenerService ls*), we use Java reflection. Note that an attacker can also take advantage of a Java decompiler such as Procyon[Mi] to ease this process. According to documentation, a flow tracker is a sub-component of the controller service that is responsible to manage reading and writing of flow rules. In particular, it is involved in case a controller component or an SDN application calls getFlowStats() in order to request a list of currently installed flow rules. The documentation does not say much about a listener service, but it seems that it is responsible for managing a list of message listeners which, for example, can receive OpenFlow multipart response messages. This is indicated by a private variable called *msgHandlers* which contains, among others, the message listener of the aforementioned flow tracker. Such OpenFlow messages are particularly important in this context as they also contain a list of current flow rules. Based on these insights, we abuse Java reflection and subvert critical functions of the HP controller without re-using its source code. Figure 1 illustrates the mechanism we implement to hide malicious flow rules, to fake the existence of removed legitimate flow rules, and to hide adversely modified flow rules.

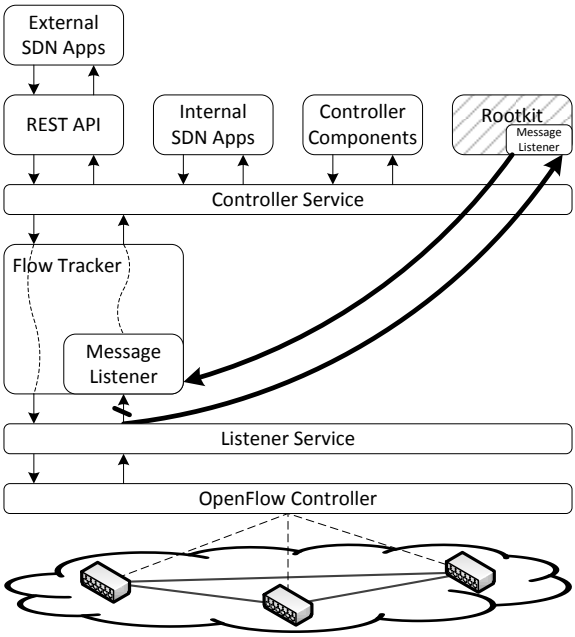

Fig. 1: Hiding network manipulations

In detail, we hook the control flow when getFlowStats() is called as this function is responsible for returning the current network state in terms of installed flow rules. The normal control flow starts when an SDN application (or a controller component) calls this controller service's function. In turn, this service uses its flow tracker sub-component and passes this request towards the network. The flow tracker in turn uses the listener service which in turn uses a component called *openflow service* that sends OpenFlow multipart

request messages to the network. As a result, the corresponding SDN switch returns a multipart response message including all of its flow rules including the malicious ones. Inside the HP controller, receiving such an OpenFlow message generates an event which is passed to the aforementioned flow tracker's message listener. Normally, this flow tracker would pass the event data to the controller service, which returns the requested flow rules to the component which was calling getFlowStats() in the first place. However, we hook into this control flow in order to enable our rootkit to filter out rootkit related data *before* it is passed to the controller service. To achieve this, we implement a message listener which any SDN application is allowed to do. Since the flow tracker's message listener is called before the one of our rootkit, filtering out data is not effective yet. For this purpose, we must manipulate the control flow in a way that the rootkit's message listener is executed before or instead of the flow tracker's one. By applying Java reflection we manipulate the private list of message handlers (i. e., *msgHandlers*), which is managed by the listener service, and replace the flow tracker's message listener object by our rootkit's listener. From now on, all event data, which is supposed to reach the flow tracker's message listener, is in fact passed to our rootkit. In order to keep the HP controller functioning, we also invoke the flow tracker's message listener (i. e., the event function of its message listeners) after we filter out rootkit related data. Thus, we are able to to hide malicious network manipulations from all controller components as well as from all potential SDN security applications.

The remaining challenges (i. e., hiding rookit components and hiding from network events) are solved as follows. In order to hide artifacts generated through a normal installation procedure, we avoid using the controller's web interface, but use the controller's deploy folder *virgo/pickup*. As this folder can be used for hot deployment tasks, the HP controller automatically attempts to install all files which are copied to this folder. When installing our rootkit like this, it neither appears in the list of installed SDN application nor does the HP controller's signature validation mechanism raises any alert. Note that other popular SDN controllers such as OpenDaylight [Opb] and ONOS [ON] also support such a deplay folder. Hiding this bundle from controller internal functions can be achieved as presented in previous work [RH15b]. With respect to hiding from network events, the HP controller provides two valuable services: (i) flow listeners and (ii) message listeners. Both are suitable to monitor critical network messages such as flow_mod and flow_removed. These can be used, for example, by a monitoring application in order to observe what flow rules are actually added, modified, and deleted. By comparing this view of the network with data provided by getFlowStats(), differences in terms of hidden manipulations can be revealed. Unfortunately, there are two drawbacks regarding these components. First, the flow listener function does not work correctly in version 2.7.10 (and probably earlier versions) but only in the newest version (i. e., 2.7.18). Second, registered flow listeners get informed about such critical network events only if a special flow programming function is used (i. e., sendConfirmedFlowMod()). For example, the controller's REST API uses this function which is based on OpenFlow barrier response messages. However, in case we call *sendFlowMod()* no such barrier request messages are generated after sending a flow_mod message. Thus, our rootkit can add, modify, and delete flow rules without notifying a

flow listener. In a similar way, registered message listeners get also notified only in case sendConfirmedFlowMod() is used, which our SDN rootkit does not.

To evaluate this first SDN rootkit, we run several tests. First, we test default monitoring capabilities of the HP controller and potential monitoring capabilities of SDN security applications. Former tests include reviewing the list of SDN applications provided via the controller's web interface, and observing the list of flow rule statistics which is provided by the web interface as well as by the controller's REST API. For the latter tests, we implement an SDN application with comprehensive monitoring capabilities. As a result, we find that our rootkit is capable of hiding its artifacts from all these tests. In addition, we not only test the rootkit against the initially targeted HP controller version (i. e., 2.7.10) but also against other versions (i. e., v2.6.11, v2.7.10, v2.7.16, and the newest version 2.7.18). To attack all these versions, only a few changes are necessary. Regarding v2.6.11, we recompile our rootkit with the SDK shipped with this controller version. With respect to v2.7.18, we change a single letter in the rootkit code as in this controller version a variable's name changes from *ls* to *lm*.

3.2 Abusing Aspect-Oriented Programming

Our second SDN rootkit uses aspect-oriented programming instead of Java reflection. Since by default the HP controller only provides limited AOP support via the Spring AOP framework [Ro], we install and use AspectJ [Xe] which is more powerful. To achieve this, we include AspectJ-related software within the rootkit package. Then, during the installation of this rootkit the HP controller is configured appropriately. In particular, the rootkit extracts AspectJ files to the correct folders, manipulates HP controller configuration files to add AspectJ support, and finally triggers a controller restart. As adding AspectJ support is achieved during runtime, an attacker can used it as this capability were enabled by default. Note that an attacker may find a way to use the AOP support, which is shipped with the HP controller, instead of AspectJ.

Based on insights gained during the implementation of the first rootkit, we solve several challenges in the same way. In particular, finding suitable spots within the HP controller, hiding rootkit artifacts generated during its installation, and hiding from network events are solved in the same fashion as described before (see Section 3.1). The remaining and more interesting challenge is replacing Java reflection by AOP as this would allow an attacker to subvert SDN controllers although Java reflection is prohibited. For this purpose, our second rootkit uses AspectJ to replace entire methods in order to hook the HP controller's normal control flow. Instead of manipulating a list of message listeners via Java reflection, this eases subverting SDN controllers significantly.

Figure 2 shows how this is achieved to hide malicious flow rules. First, we define a *pointcut* to specify a point during the execution of the *getFlowStats* method. Then, we replace the original method by our own one by using an *around* advice. Now, in case OpenFlow

```
pointcut pc(DataPathId d,TableId t): call(* *.getFlowStats(DataPathId,TableId)) && args(d,t);
List<MBodyFlowStats> around(DataPathId d, TableId t) : pcName(d, t) {
  // get actual network state
  List<MBodyFlowStats> orig = proceed(d, t);
  // create filtered data set
  List<MBodyFlowStats> filtered = new ArrayList<MBodyFlowStats>();
  for (int i = 0; i < orig.size(); i++) {
    for (int j = 0; j < flows_to_hide.size(); j++) {
      // if current flow rule should NOT be hidden, add to filtered
    }
  }
  return filtered;
}
```

Fig. 2: AspectJ Example of Hiding Adverse Network Manipulations

messages are received by the HP controller, this replacement can handle them. For the purpose of hiding malicious flow rules, we compare the received list of installed flow rules with a list of flow rules to hide. In case of a hit, we filter it out. Finally, we return a set of filtered flow rules to the caller of getFlowStats(), which is typically an SDN application or a controller component. As a result, our AOP-based rootkit is able to hide malicious network manipulations from the HP controller successfully. In order to demonstrate the effectiveness of this rootkit, we re-run the tests performed to evaluate our first rootkit. This includes reviewing the list of SDN applications and observing the list of flow rule statistics both provided by the controller's web interface as well as by the controller's REST API, and running an SDN application which implements monitoring capabilities.

4 Limitations and Discussion

Since our SDN rootkits take advantage of Java-specific capabilities, these attacks primarily affect Java-based SDN controllers. However, SDN controllers written in other programming languages also suffer from malicious SDN applications [Sh14]. In particular, AOP support is not limited to Java and, thus, can enable attackers to replace critical code, for example, for C++-based SDN controllers [OS]. In addition, our second rootkit currently requires the HP controller's hot deployment mechanism in order to install additional software. As many closed source SDN controllers are based on OpenDaylight [SD15], which also supports such a mechanism, an attacker can abuse this in a similar fashion. Furthermore, we target only one closed source SDN controller as attacking various systems would go beyond the scope of this work. Nevertheless, we argue that several other systems are equally affected as many closed source SDN controllers use OpenDaylight as a basis, which is currently neither able to prevent Java reflection nor AOP operations.

To tackle the raised security concerns, we suggest improvements which are specific to the HP controller, on the one hand, and generally applicable for SDN controllers, on the other hand. With respect to the HP controller, flow and message listeners should be enabled to receive flow programming related events independent of barrier messages. This would

allow a more realistic observation of messages which are related to network manipulations. Furthermore, validating signatures of software should be activated for the entire controller platform. In particular, we strongly recommend to cover the controller's hot deployment folder *virgo/pickup*. Moreover, we suggest to ship the controller with default monitoring capabilities which is capable of finding obvious inconsistencies. In addition, a mechanism should be provided which protects critical controller mechanisms from being hijacked. Particularly, invoking message listeners must be protected in a way that listeners of SDN applications are not processed before the ones of controller components.

In more general, SDN controller vendors may be interested in implementing security improvements such as (i) putting SDN applications into a sandbox, (ii) tracking reflection and AOP related critical operations, and (iii) comparing the actual network state with the state provided to SDN applications. In particular, several sandbox systems have been proposed already [RH15a, Sh14, CTB16, Yo17]. With such a system, our Java reflection and AOP based attacks can be prevented, for example, by denying access to corresponding critical Java operations. However, it is worth noting that the use of such operations is not malicious per se and, thus, SDN applications can use them in a benign manner as well. Hence, we suggest to track corresponding critical operations in order to prevent only a malicious utilization. Another possibility is to compare the actual network state provided by network devices with the network view provided to SDN applications. Hereby, discrepancies such as hidden flow rules can be easily revealed by a dual-view comparison [Ta17].

5 Related Work

The problem of malicious SDN applications was first raised by Porras et al. [Po12]. The authors presented a new technique which enables attackers to bypass existent flow rules by exploiting the standard OpenFlow instructions *set* and *goto*. Shin et al. [Sh14] and Röpke et al. [RH15a, RH16] extended this work by presenting SDN applications which can harm SDN controllers by implementing rudimentary malicious logic. For this purpose, the authors exploited the fact that many SDN controllers run their SDN applications within the same execution environment. In addition, Röpke et al. [RH15b] demonstrated a more sophisticated malicious SDN application which is able to compromise an SDN controller and, thus, an entire SDN via abusing Java reflection. Complementary to aforementioned research, we adopt Java reflection-based attacks to a closed source SDN controller and, additionally, present an AOP based technique to compromise SDN controllers. Although manipulating Java programs via AOP is not new in general [Wi09], abusing AspectJ to subvert SDN controllers is new in the context of SDN.

6 Conclusion

In this paper, we present two new SDN rootkits which aim to subvert a closed source SDN controller. The first one extends existing Java reflection based attacks while the

second one abuses aspect-oriented programming techniques to subvert critical controller functions. Both rootkits are able to compromise the HP controller in a default setup and both attacks do subvert this controller on such a deep level that even multiple release versions are affected. Clearly, this shows that an SDN rootkit is not only a severe threat to open source SDN controllers but also to closed source ones. We also demonstrate that preventing SDN applications from accessing reflection-related operations is not enough to protect SDN controllers. Further attack vectors must be considered. Moreover, we find that using Java reflection to subvert SDN controllers heavily depends on a concrete implementation. Thus, compromising another open or closed source SDN controller probably raises new challenges with respect to finding suitable spots for hooking the control flow. To prevent such attacks, we finally discuss several countermeasures including concrete suggestions to improve security of the targeted SDN controllers.

References

[CTB16] Chandrasekaran, Balakrishnan; Tschaen, Brendan; Benson, Theophilus: Isolating and Tolerating SDN Application Failures with LegoSDN. In: ACM Symposium on SDN Research. 2016.

[DWJP05] De Win, Bart; Joosen, Wouter; Piessens, Frank: Developing Secure Applications through Aspect-Oriented Programming. Aspect-Oriented Software Development, 2005.

[DWVDD02] De Win, Bart; Vanhaute, Bart; De Decker, Bart: Security through Aspect-Oriented Programming. Advances in Network and Distributed Systems Security, 2002.

[Fl] Floodlight. `www.projectfloodlight.org/floodlight/`, Accessed: 2018-02-13.

[FMC11] Falliere, Nicolas; Murchu, Liam O; Chien, Eric: W32. stuxnet dossier. White paper, Symantec, 2011.

[FRZ13] Feamster, Nick; Rexford, Jennifer; Zegura, Ellen: The Road to SDN. ACM Queue: Tomorrow's Computing Today, 2013.

[Gu08] Gude, Natasha; Koponen, Teemu; Pettit, Justin; Pfaff, Ben; Casado, Martín; McKeown, Nick; Shenker, Scott: NOX: Towards an Operating System for Networks. ACM SIGCOMM Computer Communication Review, 2008.

[Ho12] Hoelzle, Urs: OpenFlow @ Google. Open Networking Summit, 2012.

[HPa] HP Open Ecosystem Breaks Down Barriers to Software-Defined Networking. `www.hp.com`, Accessed: 2018-02-13.

[HPb] HP VAN SDN Controller. `www.hp.com`, Accessed: 2018-02-13.

[HPc] HP Virtual Application Networks SDN Controller: The building block of HP SDN ecosystem. `www.hp.com`, Accessed: 2018-02-13.

[Ki97] Kiczales, Gregor; Lamping, John; Mendhekar, Anurag; Maeda, Chris; Lopes, Cristina; Loingtier, Jean-Marc; Irwin, John: Aspect-Oriented Programming. In: European conference on object-oriented programming. 1997.

[LHM10] Lantz, Bob; Heller, Brandon; McKeown, Nick: A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In: ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking. 2010.

[Mc] POX SDN Controller. github.com/noxrepo/pox/, Accessed: 2018-02-13.

[Mc08] McKeown, Nick; Anderson, Tom; Balakrishnan, Hari; Parulkar, Guru; Peterson, Larry; Rexford, Jennifer; Shenker, Scott; Turner, Jonathan: OpenFlow: Enabling Innovation in Campus Networks. ACM SIGCOMM Computer Communication Review, 2008.

[Mi] Procyon. bitbucket.org/mstrobel/procyon/, Accessed: 2018-02-13.

[ON] Open Network Operating System. onosproject.org, Accessed: 2018-02-13.

[Opa] OpenFlow Switch Specification. www.opennetworking.org, Accessed: 2018-02-13.

[Opb] OpenDaylight Project. www.opendaylight.org, Accessed: 2018-02-13.

[Op13] Software-Defined Networking: The New Norm for Networks. www.opennetworking.org.

[Op14] SDN Architecture Overview (ONF TR-504). www.opennetworking.org.

[Ora] Java Reflection API. docs.oracle.com/javase/7/docs/technotes/guides/reflection/, Accessed: 2018-02-13.

[Orb] Security Alert for CVE-2012-4681. www.oracle.com, Accessed: 2018-02-13.

[Orc] Security Alert for CVE-2013-0422. www.oracle.com, Accessed: 2018-02-13.

[Ord] Sun Alert 1000148.1. download.oracle.com/sunalerts/, Accessed: 2018-02-13.

[Ore] Sun Alert 1000560.1. download.oracle.com/sunalerts/, Accessed: 2018-02-13.

[Orf] Sun Alert 1000975.1. download.oracle.com/sunalerts/, Accessed: 2018-02-13.

[OS] AspectC++. www.aspectc.org, Accessed: 2018-02-13.

[Po12] Porras, Philip; Shin, Seungwon; Yegneswaran, Vinod; Fong, Martin; Tyson, Mabry; Gu, Guofei: A Security Enforcement Kernel for OpenFlow Networks. In: ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking. 2012.

[PS08] Phung, Phu H; Sands, David: Security policy enforcement in the OSGi framework using aspect-oriented programming. In: 2008 32nd Annual IEEE International Computer Software and Applications Conference. 2008.

[RH15a] Röpke, Christian; Holz, Thorsten: Retaining Control Over SDN Network Services. In: International Conference on Networked Systems. 2015.

[RH15b] Röpke, Christian; Holz, Thorsten: SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In: Symposium on Recent Advances in Intrusion Detection. 2015.

[RH16] Röpke, Christian; Holz, Thorsten: On Network Operating System Security. International Journal of Network Management, 2016.

[Ro] Spring Framework Reference Documentation. docs.spring.io/spring/docs/current/spring-framework-reference/html/, Accessed: 2018-02-13.

[SD15] SDxCentral: SDN Controllers Report. www.sdxcentral.com, 2015.

[Sh14] Shin, Seungwon; Song, Yongjoo; Lee, Taekyung; Lee, Sangho; Chung, Jaewoong; Porras, Phillip; Yegneswaran, Vinod; Noh, Jiseong; Kang, Brent Byunghoon: Rosemary: A Robust, Secure, and High-Performance Network Operating System. In: ACM SIGSAC Conference on Computer and Communications Security. 2014.

[Sh16] Shin, Seungwon; Xu, Lei; Hong, Sungmin; Gu, Guofei: Enhancing Network Security through Software Defined Networking (SDN). In: Proceedings of the 25th International Conference on Computer Communication and Networks. 2016.

[Ta17] Tatang, Dennis; Quinkert, Florian; Frank, Joel; Röpke, Christian; Holz, Thorsten: SDN-Guard: Protecting SDN controllers against SDN rootkits. In: Network Function Virtualization and Software Defined Networks (NFV-SDN), 2017 IEEE Conference on. 2017.

[Va17] Vahdat, Amin: Cloud Native Networking. Open Networking Summit, 2017.

[VBC01] Viega, John; Bloch, JT; Chandra, Pravir: Applying Aspect-Oriented Programming to Security. Cutter IT Journal, 2001.

[Wi09] Williams, Jeff: Enterprise Java Rootkits: Hardly anyone watches the developers. BlackHat USA, 2009.

[Xe] The AspectJ Programming Guide. eclipse.org/aspectj/doc/next/progguide/index.html, Accessed: 2018-02-13.

[Yo17] Yoon, Changhoon; Shin, Seungwon; Porras, Phillip; Yegneswaran, Vinod; Kang, Heedo; Fong, Martin; O'Connor, Brian; Vachuska, Thomas: A Security-mode For Carrier-grade Sdn Controllers. In: Anual Computer Security Applications Conference. 2017.