SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks

Christian Röpke and Thorsten Holz

Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum
christian.roepke@rub.de, thorsten.holz@rub.de

Abstract The new paradigm of *Software-Defined Networking* (SDN) enables exciting new functionality for building networks. Its core component is the so called *SDN controller* (also termed *network operating system*). An SDN controller is logically centralized and crucially important, thus, exploiting it can significantly harm SDN-based networks. As recent work considers only flaws and rudimentary malicious logic inside SDN applications, we focus on rootkit techniques which enable attackers to subvert network operating systems. We present two prototype implementations: a SDN rootkit for the industry's leading open source controller OpenDaylight as well as a version with basic rootkit functions for the commercial and non-OpenDaylight-based HP controller. Our SDN rootkit is capable of actively hiding itself and malicious network programming as well as providing remote access. Since OpenDaylight intends to establish a reference framework for network operating systems (both open source and commercial), our work demonstrates potential threats for a wide range of network operating systems.

1 Introduction

Over the last few years, the new paradigm of *Software-Defined Networking* (SDN) has attracted a lot of attention from both industry and academia [6, 16, 22]. Industry is already adopting SDN technologies and sells SDN-ready products (*e. g.*, OpenFlow-enabled switches and SDN control software), tests feasibility in enterprise networks [11, 13], and introduces new ecosystems for SDN-based networks [10]. In academia, a new research area has evolved including security-related topics such as using SDN to enhance network security [1, 31] and developing countermeasures against SDN-specific attacks [12, 33]. Broadly speaking, SDN physically decouples the *control plane*, on which network control programs decide where traffic is sent to, from the *data plane*, on which packet forwarding hardware forwards traffic to the selected destination. Furthermore, SDN promotes open interfaces to both network devices and physically decoupled network operating systems. Considering the computer market

growth after introducing processor chips with open interfaces, SDN enables rapid innovation for the network market [21]. In such a network, the logically centralized network operating system is responsible to program network devices, thus, managing the entire programmable network. The SDN architecture promises hereby to be more dynamic, manageable and cost-effective compared to traditional networks. Concerning security, SDN-based networks benefit from automatism and standard protocols to provide a more adequate and faster reaction on security incidents.

Several security mechanisms for SDN-based networks and especially for network operating systems have been proposed recently. For example, Kreutz *et al.* [15] present security and dependability techniques to secure software-defined networks on a more generic level. FortNOX [29] is based on a security kernel and detects as well as resolves attempts to circumvent existing security flow rules. AvantGuard [33] implements countermeasures to mitigate denial-of-service attacks against SDN controllers launched by network clients. Additionally, Hong *et al.* [12] and Dhawan *et al.* [5] introduce methods to counter SDN-specific attacks which are similar to traditional ARP cache poisoning. With respect to malicious SDN applications, few papers [30, 32] study flaws and rudimentary malicious logic and apply sandbox techniques to restrict access to critical operations such as establishing network connections or executing shell commands.

In this paper, we go one step further and investigate rootkit techniques primarily specialized for network operating systems. Compared to previous work, we analyze how attackers can subvert network operating systems via sophisticated malicious SDN applications based on the design principles of rootkits. As a result, we present new challenges and two proof-of-concept implementations for the popular and greatly industry-supported open source OpenDaylight controller as well as for a closed source one. In particular, our SDN rootkit subverts the targeted network operating systems and provides adversely network programming from a remote host, while all carried out manipulations are actively hidden. In addition, we test our SDN rootkit against several available security mechanisms to prove the existence of this threat.

To summarize our work, we provide the following main contributions:

- We investigate sophisticated attacks against modern network operating systems and present new challenges regarding SDN rootkits. Moreover, we develop a new technique for introducing remote access in a SDN-specific fashion.
- We present proof-of-concept implementations of SDN rootkits. In particular, we provide a fully functional version for the industry's leading

open source OpenDaylight controller as well as a version with basic functionality for the HP controller. To the best of our knowledge, these are the first prototype implementations of SDN rootkits.
– We test our SDN rootkits against several detection and protection mechanisms and find that current security mechanisms cannot adequately stop SDN rootkits.

2 Background

Before we introduce the concept of SDN rootkits, we first provide background information necessary to understand the SDN-specific aspects.

2.1 Software-Defined Networking

The *Open Networking Foundation* (ONF) [23] is a user-driven organization which is greatly supported by industry. Among others, the ONF publishes SDN standards and defines the SDN architecture [25] by the following layers: (i) *infrastructure layer*, (ii) *control layer*, and (iii) *application layer* (see Figure 1). The infrastructure layer consists of *programmable network devices* which merely forward network packets. On top, *SDN control software* (currently implemented by a SDN controller / network operating system) operates on the control layer and programs the network devices via an open *control data plane interface* (also known as *southbound interface*). Probably the most widely used southbound protocol is *OpenFlow* [19] which facilitates both programming switches via flow tables and requesting their current state. Note that malicious network programming will take place in such flow tables. On the control layer, so called *network services* run inside the control software and provide access to SDN resources while hiding implementation details from the application layer. On the application layer, so called *business applications* operate on a global network view, leveraging network services via an open interface (also known as *northbound interface*). As we will see later in this section, attackers may operate on each of these layers.

To provide an analogy to operating systems like Linux, one can consider that SDN controllers provide interfaces and abstractions for software developers just like an OS. Thus, SDN controllers are also denoted as network operating system [8, 32]. Correspondingly, both types of SDN applications (namely, network services and business applications) can be considered as kernel applications and user applications, respectively. In this work, we use the terms NOS and SDN controllers interchangeably.

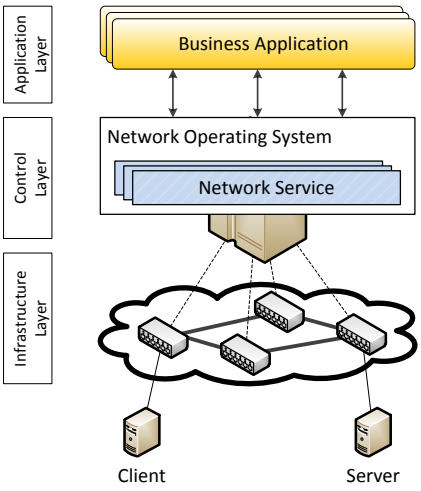

Figure 1. SDN Architecture

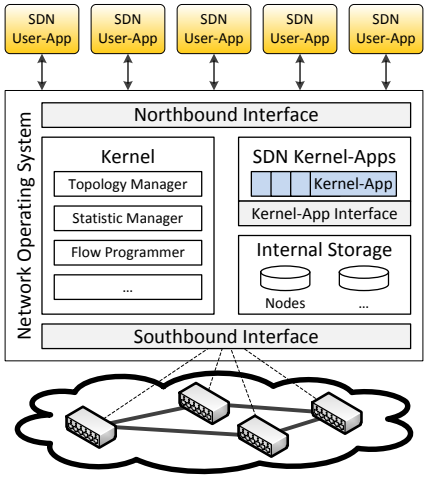

Figure 2. NOS Components

2.2 Network Operating System alias SDN Controller

Network operating systems are logically centralized and responsible for controlling the entire SDN, thus, playing a major role. As illustrated in Figure 2, the main tasks are providing (i) a global network view, (ii) network statistics, (iii) a northbound interface as well as a (iv) southbound interface. Another important service is the *flow rule programmer* which performs the actual network programming. A typical scenario would be reactive network programming: assuming that a switch cannot process a network packet due to a missing flow table entry, the switch delegates the forwarding decision to the NOS. The NOS considers the delegated information as well as the current network state and determines an adequate forwarding decision which is sent back to the requesting switch, typically, via a flow rule. Subsequently, such packets are forwarded by the switch according to the previously added flow rule. As we will see in Sections 3 and 4, the aforementioned services are the main objectives a SDN rootkit wants to manipulate.

Furthermore, many SDN controllers (*e. g.*, the ones released by HP and Cisco) support the installation of SDN kernel applications during runtime whereas others, such as Floodlight [7], are inflexible regarding this. According to Cisco [4], runtime flexibility is important for implementing business needs. Similarly, HP explicitly supports runtime flexibility and started the first SDN App Store which hosts SDN kernel applications (even from third-parties) which are supposed to be installed during

runtime. Note that SDN App Store compromises may become a relevant malware attack vector in future, like in the smartphone industry.

2.3 Motivating Examples

Since there are only a few SDN-based networks today (mostly for research purposes), malicious SDN applications have not been reported yet. Thus, we want to motivate the possibility of NOS compromises by the following three examples. For that purpose, we penetrate the HP controller [9] and aim to install a malicious SDN application despite the fact that this NOS refuses such installations if applications do not provide a valid signature.

First, we simply modify the HP controller's configuration file *virgo/configuration/osgi.console.properties*, enable the OSGi console for local access (`telnet.enabled=true`) and trigger the controller process to restart (*kill -HUP <virgo pid>*). As a result, a connection to the OSGi console of the SDN controller can be established via *telnet localhost 2401*. An attacker could now install arbitrary SDN applications despite the fact that they do not have a valid signature.

Second, we copy a malicious SDN application into the controller's plugin directory and modify the controller's configuration file *virgo/configuration/config.ini*. Adding an entry here tells the controller to start the previously copied SDN application during startup. After triggering the controller to restart, a configured malicious SDN application can be installed and started, again, without presenting a valid signature.

Third, we copy an arbitrary file to the SDN controller's directory *virgo/pickup*. Since the HP controller uses this directory for hot deployment, any malicious SDN application is automatically installed and started after a few seconds.

Similarly, code verification may present vulnerabilities [26, 27] or attackers could simply disable the SDN controller's security (CVE-2012-4681, CVE-2013-0422). Although we assume that security experts operate such systems, we want to motivate that NOS may present further vulnerabilities which could be exploited, *e. g.*, through targeted attacks as they have become prevalent in today's networks.

2.4 Attacker Model

We assume that a sophisticated attacker could compromise a SDN App Store or distribute manipulated SDN software otherwise. Such a SDN application (*e. g.*, a fake update) could exploit a vulnerable NOS aiming at bypassing different protection systems like verifying the SDN application's

digital signature [26,27] or disabling the NOS's security (CVE-2012-4681, CVE-2013-0422). As a consequence, an attacker is able to compromise the NOS via a malicious SDN application.

Considering that attackers can already write empty files to the SDN controller's file system (CVE-2014-8149) or provoke crashes of SDN services during network packet parsing (CVE-2015-1166), another attack scenario would be to exploit vulnerabilities in a NOS through a network client. Assuming further NOS vulnerabilities, an infected network client can install SDN malware on the SDN controller by sending specially crafted network packets containing the payload. For instance, an infected network client sends a specially crafted packet to a switch which delegates it to the NOS due to a missing flow rule. During packet parsing, a vulnerability is triggered within the NOS and the packet's payload is written to the NOS's file system to a certain directory. As a result, the NOS considers the written file as SDN application and installs it as part of its hot deployment task.

3 SDN Rootkits

Traditional rootkits (*e. g.*, for Windows or Linux) mainly aim at hiding their artifacts in order to remain undetected as long as possible. A mature technique to achieve that is the hooking of selected control data inside an OS (*e. g.*, the system call table) in order to transfer the control flow to the rootkit. While this has remained almost unchanged over the last years, a new technique has been presented recently [34]. It is no longer based on aforementioned control data but uses non-control data for the control transfer. In future networks, however, we face new challenges and require suitable techniques which consider the specifics of SDN-based networks.

3.1 Challenges

In the following, we present SDN rootkit challenges necessary to understand that we can later on provide adequate countermeasures.

Hiding SDN Rootkits From a NOS. Hiding rootkit artifacts from a NOS works similar to hiding them from a commodity OS. Since NOSs are often designed as service-oriented systems, *e. g.*, based on OSGi, the type of artifacts to hide as well as the data structures holding such artifacts differ with respect to traditional rootkits. For example, a SDN rootkit necessarily consumes services in order to perform malicious actions, but

it wants to hide itself from the same services to remain undetected. At the same time, however, the service must know its consumers to notify them about new events (which is typically needed by SDN rootkits). Thus, the challenge is to hide rootkit artifacts (*e. g.*, from a service) while it remains capable of performing malicious actions (*e. g.*, based on service notifications).

Hiding a Malicious Network State From the NOS. New and specific to SDN is hiding malicious network programming. This includes adding of malicious flow rules as well as pretending the existence of previously removed (security) flow rules. To make malicious network programming effective, corresponding flow rules must be present (or not) at least in the SDN-enabled switches. Hence, the challenge is to manipulate the NOS's view on the network state despite the fact that it has direct access to the switch's internals as this is part of the SDN paradigm.

Hiding Unwanted Remote Access Communications. Remote access to malicious functions is highly desirable by attackers. Reasonably, such communications should not differ much from normal communication. In SDN-based networks, it is normal to exchange messages between the NOS and the associated switches. However, it is not part of the SDN architecture to provide a connection between network clients (residing in the user network) and the NOS (residing in the management network). Thus, the challenge is to establish a connection which by design should not be possible and which on top does not significantly differ from normal control traffic.

Resolving Non-Existent Rule Conflicts. While hiding malicious flow rules, network operators may want to add similar but legitimate flow rules. For example, assume a legitimate flow rule which matches on the same packet header fields as a malicious one but contains different actions. Adding this legitimate flow rule would cause a rule conflict which must not occur in order to keep malicious behavior hidden. Hence, the challenge is to control the NOS's rule conflict mechanism and to handle possible rule conflicts.

Faking Non-Existent Network Statistics. Assuming that an attacker has already removed a flow rule (*e. g.*, a security rule which blocks traffic from a certain IP address) the flow rule's existence must be pretended. Otherwise, the corresponding security application or a network

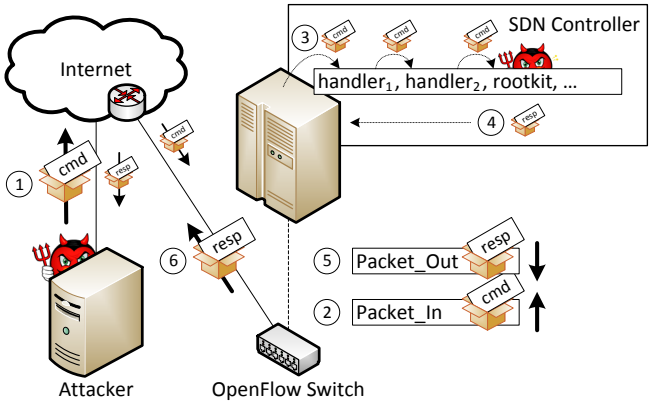

Figure 3. Remote Access via Packet-In and Packet-Out Messages

operator can discover the misconfiguration, thus, revealing the manipulated network state. The challenge is to fake reasonable statistics for the removed flow rules.

3.2 A New Technique For Remote Access

A novel and specific aspect of SDN is establishing a connection between the attacker's host and the SDN rootkit for remote access purposes. Note that such connections are neither part of the SDN architecture nor desirable in any way. However, in the following we present a new technique how this can be achieved for the probably most widely used southbound protocol, *i. e.*, OpenFlow. In OpenFlow, so called *packet-in messages* are sent by OpenFlow-enabled switches in order to delegate forwarding decisions of occurring network packets to the NOS. In response, the NOS determines an adequate decision by taking both packet information and the current network state into account. It then sends this decision back to the requesting switch via a so called *packet-out message*, typically resulting in adding a new flow rule. As illustrated in Figure 3, attackers can misuse that standard behavior in order to establish a remote access communication.

On the one hand, attackers could send specially crafted packets for which a switch cannot determine a forwarding decision, thus, it delegates such packets to the NOS. On the other hand, an attacker can install hidden flow rules on each switch which exactly match on the attacker's packets, thus, causing delegation of such packets to the NOS. Inside the NOS, the packet-in message including the attacker's packet (or only the

packet's headers) is passed to a registered set of packet-in message handlers. Depending on the NOS, such messages are passed to aforementioned handlers in a sequential fashion allowing each handler to pass the message to the next handler if it cannot provide an adequate forwarding decision. To avoid that other handlers generate responses to the attacker's packets (or even drop the corresponding packet-in message), the SDN rootkit must either ensure that it is the first handler which is called for processing a new packet-in request or that the previous handlers keep passing the packet until the SDN rootkit's packet-in handler is reached.

If the attacker's packets reach the rootkit's handler, it can simply parse the commands (*e. g.*, encoded in the packet's header or the payload) and execute the corresponding malicious actions. Some commands such as *adding a new malicious flow rule* may not necessarily require feedback whereas others like *download the network topology* do. For the latter commands, the SDN rootkit's packet-in handler can create packet-out messages including previously collected information (such as the network topology) which are sent back to the switch. According to the given packet-out settings, the switch forwards the network packet towards the attacker's host. Since such control communications between the NOS and associated switches are normal, attackers can establish a remote access connection which likely remains stealthy.

4 Prototype Implementation

Based on the concept introduced in the last section, we now provide two prototype implementations, one for the industry's leading open source OpenDaylight controller, and another one for the HP controller. OpenDaylight is the foundation for several enterprise NOSs of large companies such as Cisco, IBM, Brocade and Extreme Networks whereas the HP controller is considered as representative for commercial network operating systems. More specifically, we use OpenDaylight's Helium base release (SR1) and the HP controller of version 2.3.5. The current implementation of our SDN rootkit supports the following functions: hiding the rootkit's artifacts, hiding malicious network programming (including rule conflict resolution and faking statistics) as well as providing remote access.

4.1 Rootkit Hiding

For OpenDaylight, there are two main interfaces to display information about NOS internals such as a list of installed SDN applications or running NOS services: a console and a web interface. Since we consider our

rootkit to be installed as a normal SDN application, we focus on hiding artifacts which occur inside the NOS during the installation process. It mainly includes the creation of a unique object for each SDN application which is added to the protected list of installed SDN applications. This object is further on used, for example, to register a new NOS service or to consume an already registered service to implement its functionality. Accordingly, this unique object is added to a protected list of registered services and to a protected list of consumers of an already registered NOS service, respectively. As these lists are protected, they are not supposed to be modified by SDN applications.

However, we utilize Java reflection and manipulate each of these protected lists to remove the objects which are added during the installation. This works fine for the first two lists, *i. e.*, the lists of installed SDN applications and the one of registered NOS services. But if we remove the rootkit's object from the consumer list of a NOS service, the SDN rootkit remains unable to react on service events. Therefore, we additionally replace the OpenDaylight's service registry which is responsible for notifying the associated consumers of a NOS service. This is currently implemented by replacing the service's object by an object of our own registry service. As a result, the SDN rootkit can consume NOS services while it remains hidden from the NOS.

Since other SDN controllers such as the HP SDN controller are built upon the same execution environment as OpenDaylight, the presented artifact hiding works on many other NOS as well. In particular, we implement this basic rootkit functionality also for the HP controller.

4.2 Malicious Network Programming

To implement malicious network programming, we basically manipulate the OpenDaylight's view on the network. This view is either cached in protected flow rule databases or based on information which is directly received from the network through a so called *read service* and a so called *flow programmer service*. The most interesting databases are: *StaticFlows*, *originalSwView* and *installedSwView*. *StaticFlows* contains static and pro-actively installed flow rules which can be managed via OpenDaylight's web interface. For example, we use this to add security flow rules, *e. g.*, to drop network packets coming from a certain host. The databases *originalSwView* and *installedSwView* contain the software view of the network. While the former one manages the flow rules which are requested to be installed, the latter one contains the flow rules which are actually installed on the switches. The latter two databases are typically

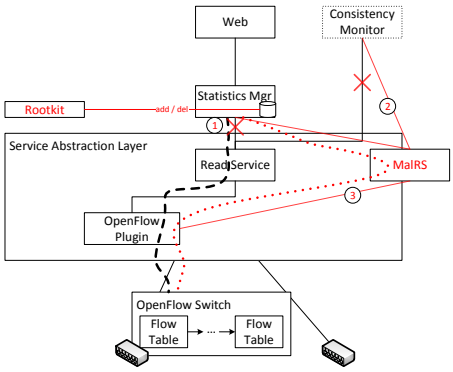

Figure 4. Read Service Manipulation

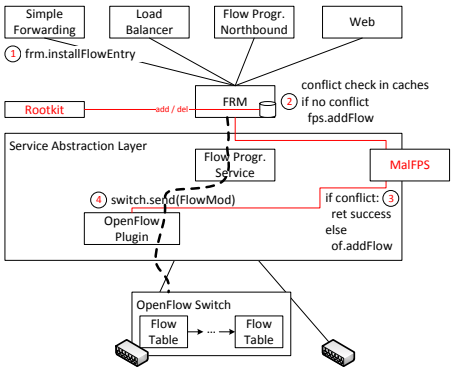

Figure 5. Flow Progr. Manipulation

used for reactive programming, for example, by the OpenDaylight's load balancing service.

In addition, OpenDaylight provides direct access to the programmable network through a read service (which allows to read information stored on a network switch) and a flow programmer service (which enables adding and removing flow rules on such network devices). Figure 4 illustrates how an OpenDaylight service such as the *statistic manager* can directly access switch information such as current flow statistics (dashed line). Thereby, it uses the NOS's read service and sends OpenFlow messages towards the programmable switches in order to request flow rule statistics such as the the byte count or the packet count of a certain flow rule. Figure 5 depicts how OpenDaylight's *Forwarding Rules Manager* (FRM) uses the flow programmer service in order to add and remove a certain flow rule to/from programmable switches (dashed line).

Currently, we implement hiding of malicious flow rules as well as pretending the existence of previously removed security flow rules by invoking the forwarding rules manager's internal functions *addEntryInHw* and *removeEntryInHw*. Inside the FRM, these functions are called after no rule conflicts were detected and after the flow rule caches got updated. However, we facilitate Java reflection and bypass cache updates in order to directly add and remove flow rules to/from the network. Furthermore, we manipulate the view on the network's statistics by replacing the OpenDaylight read service by our own one. As illustrated in Figure 4, we disconnect the statistics manager (and other read service consumers) and replace it by our own version. Then, we connect our malicious read service to the OpenFlow plugin and thereby control the readable view on

the network (dotted line). This is used, for example, to skip statistics of malicious flow rules and to add fake statistics of removed flow rules.

With respect to faking removed flow rules, we also need to consider that a flow rule removal triggers a flow remove event which must be handled to reduce attention. Currently, we wait a few moments after removing a security flow rule and allow the FRM to handle such events. Meaning that the FRM removes the corresponding entries from the affected flow rule caches. Then, we manipulate these flow rule caches and insert fake entries as the corresponding entries would have never been removed.

Concerning flow rule conflicts, we must also control OpenDaylight's rule conflict mechanism. Currently, we replace OpenDaylight's flow programmer service by our own one which enables to check for rule conflicts before OpenDaylight's FRM can do it. In case a network operator adds a new flow rule which is in conflict with a malicious one, our replacement resolves the conflict as follows. First, the malicious flow rule is temporarily stored and removed from the network. Then the new security flow rule is added normally such that the network operator can test its effectiveness, *e. g.*, by sending ping requests. Finally, the malicious flow programmer service waits a certain amount of time, replaces the security flow rule by a fake entry, and restores the previously removed malicious flow rule.

4.3 Remote Access

We realize remote access capabilities to our SDN rootkit functions by implementing the previously introduced technique which is based on OpenFlow's packet-in and packet-out messages. OpenDaylight provides a sequential processing of packet-in events. Meaning that in case of such an event, the corresponding message including the network packet is passed to the packet-in handler which was registered first. This handler can decide to mark that message as being handled or it can pass it to the next handler, *i. e.*, the packet-in handler which was registered second. This continues until either a packet-in handler marks a packet-in message as being handled or all packet-in handlers are called.

In case of OpenDaylight, all default packet-in handlers pass each packet-in message to the next packet-in handler. Therefore, we currently implement remote access capabilities by means of an OpenDaylight packet-in handler which is normally registered but actively hidden by previous rootkit functionality. Commands like `addMalFlow`, `removeSecFlow` and `listTopology` are currently encoded by the TCP source port and parameters like the flow name, matching fields or actions are encoded within the URL of a HTTP GET request. While the former two commands do

not require any feedback, the latter command requires to send back the collected information. Beside receiving packet-in messages, each OpenDaylight packet-in handler is also allowed to create and send packet-out messages. We use this to send back confidential information towards the attacker's host.

If installed on a NOS, such a SDN rootkit can be utilized in order to control entire networks. For example, an attacker could remotely send the command `listTopology`. After identifying internal servers which are not supposed to be accessed by external hosts, the attacker could send another command such as `addMalFlow`. Assuming correct network re-programming, at this point the attacker can access internal servers from remote, *e. g.*, to perform further privilege escalation. Network re-programming could also be used to copy internal traffic, thus, forwarding that copy to the attacker. Since SDN is supposed to allow generic network programming, utilizing SDN rootkit functions is a powerful attack.

5 Evaluation

We evaluate our proof-of-concept implementations by performing several tests which aim at revealing rootkit artifacts while performing malicious network programming from a remote host. Therefore, we use standard system tools available for OSGi-based SDN controllers, rule conflict checks included in OpenDaylight, and a state-of-the-art policy checker called NetPlumber [13]. In our evaluation, we use OpenDaylight (Helium release, SR1) and the HP controller (version 2.3.5) as NOSs as well as the popular SDN evaluation tool called *Mininet* [17] which is able to emulate even large networks of OpenFlow-enabled switches.

5.1 Evaluation Setup

Figure 6 illustrates our evaluation environment. Beside a NOS, we run several security mechanisms, *i. e.*, a firewall application, a consistency monitor and (as previously mentioned) a real time policy checker.

We use OpenDaylight's web interface to manually add firewalling flow rules particularly in order to deny the attacker's host h_1 to communicate with host h_2. A consistency monitor frequently tests the consistency between the real network state and the state stored in the internal flow rule caches. As policy checker, we use NetPlumber which sits between the NOS and the switches. It checks at real time if adding a new flow rule

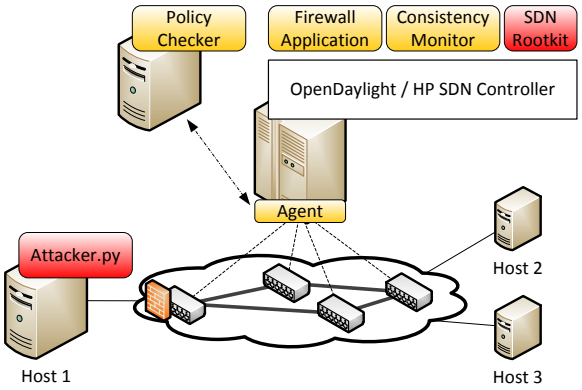

Figure 6. Evaluation Setup

or removing an existing flow rule would violate a certain security policy, and blocks programming attempts in case of detected violations.

5.2 Testing Artifact Hiding

We test artifact hiding capabilities on both network operating systems. For that purpose, we use several standard system tools provided by the NOS's execution environment which are supposed to reveal the artifacts of unwanted SDN kernel applications such as our SDN rootkit. In particular, we perform the commands `bundles` and `ss` which list installed kernel applications. Additionally, we use the `services` command to display details about all registered services and the `t` command to display information about running threads. Furthermore, we run the `services` and `status` commands to list the kernel applications which consume a registered service. After the installation of our SDN rootkit, however, none of these standard tools show a rootkit artifact on the tested systems.

5.3 Testing Malicious Network Programming

First, we test if the malicious flow rule, which is needed for enabling the remote access to our SDN rootkit, is not only installed on already connected switches, but also on switches which are possibly added during runtime. This malicious flow rule matches on the attacker's remote control packets and, thus, delegates the packets to the SDN controller and forwards them towards the rootkit's packet-in handler. We test this by adding a new switch via the Mininet console followed by performing remote access commands by a host which is only connected to that new switch.

Next, we remotely remove the pre-installed security flow rule which drops network traffic from the attacker's host h_1 to the internal host h_2. This is followed by automatically adding a corresponding fake entry including fake statistics. We verify the manipulated view on the network state by checking OpenDaylight's web console as well as the consistency monitor results which show the same flow rule as it was present before the manipulation. Additionally, we perform a ping test inside of Mininet which is launched by the attacker's host and verify host reachability in spite of a visible security flow rule.

Then, we remotely add a hidden malicious flow rule enabling the attacker to communicate with a server. As result, the attacker's host h_1 becomes able to connect to the server host h_3 whereas the corresponding flow rule does neither appear in OpenDaylight's flow rule caches, in its web console nor is it visible to our consistency monitor.

Finally, we simulate adding a new security flow rule (drop traffic between hosts h_1 and h_3) which would normally trigger a rule conflict because a similar malicious flow rule exists. Since the SDN rootkit is capable of managing this, no rule conflict alert is raised. To the contrary, we can successfully check the effectiveness of this new security flow rule by performing a ping test between host h_1 and host h_3. After some time, however, our SDN rootkit replaces this security flow rule by a fake entry and re-activates the previously disabled malicious flow rule. During this period of time, a network operator could recognize the corresponding rootkit thread which waits a few moments until it performs the aforementioned flow rule replacement and re-activation. Note that at this point another efficiency test would reveal the manipulation.

5.4 Bypassing Policy Checkers

Current policy checkers such as VeriFlow [14] and NetPlumber reasonable sit between the NOS and the associated switch network. Their main task is to block the adding of a new flow rule or the removal of an existing flow rule which would result in a policy violation. Therefore, VeriFlow was implemented inside the SDN controller (due to performance reasons) whereas NetPlumber only simulated network state changes via loading flow rules from local files. Since NetPlumber's implementation is independent from a certain SDN controller (except its agent which is needed to trigger the policy checker), we use this system in our evaluation to perform additional policy checks.

To allow OpenDaylight to run such policy checks before adding or removing flow rules, we implement the NetPlumber agent within Open-

Daylight. Reasonably we add policy checks after running OpenDaylight's internal rule conflict tests. Thus, our NetPlumber agent is able to intercept the flow rule programming process of OpenDaylight and consults the NetPlumber policy checker before a new rule is added to the network.

In case of our SDN rootkit, however, these policy checks can be bypassed. In particular, the SDN rootkit directly calls the internal functions *addEntryInHw* and *removeEntryInHw*. Thus, it avoids using the functions *addEntry* and *removeEntry* where OpenDaylight's rule conflict checks as well as the policy checks are implemented. Consequently, malicious flow rules can be added even if a policy violation would be detected normally.

5.5 Bypassing Sandbox Systems

As proposed recently [30, 32], sandbox systems help in case SDN applications present flaws or rudimentary malicious logic. In our work, we test if such a system is also able to provide adequate protection against our SDN rootkit. Therefore, we re-implement the latter sandbox system for the OpenDaylight controller as well as for the HP controller. If activated in detection mode, we observe a flood of permission entries including *java.lang.RuntimePermission accessDeclaredMembers* and *java.lang.reflect.ReflectPermission suppressAccessChecks*. Our SDN rookit depends on these permissions and uses corresponding sensitive operations to apply Java reflection. Since legitimate SDN controller components also use Java reflection, correct identification of the calling component is important.

As we implement the SDN rootkit in a separate file, identifying it as potential source for the compromise attack is possible. In protection mode, such permissions can be denied from our SDN rootkit file which results in protecting the SDN controllers from being compromised. However, sandbox configuration can be difficult in practice and misconfiguration is likely to happen (as we will discuss later). Thus, SDN rootkits could use the allowed set of critical operations for a compromise attack. Moreover, if injected into the file of a privileged SDN service (such as a SDN controller core service), a SDN rootkit could utilize these privileges to compromise the NOS. Another option is to simply disable sandboxing as a whole (see CVE-2012-4681 and CVE-2013-0422), thus, performing malicious actions without restrictions.

6 Discussion

Our evaluation shows that the examined SDN controllers do not provide adequate protection against the SDN rootkit. Policy checkers and sandbox systems, however, might be able to tackle the problem if integrated and configured correctly.

With respect to policy checkers (and other external monitoring systems), it is necessary to integrate these entirely independent from the SDN controller. For example, if the agent of the examined policy checker would physically sit between the SDN controller and the associated switches, manipulation of the SDN controller's network view would not be enough to bypass the policy checking. However, the remaining question is whether an attacker, who is able to install a SDN rootkit, is also able to compromise such an external monitoring system.

Concerning sandbox systems, correct configuration is mandatory, otherwise a SDN rootkit may simply use mistakenly granted permissions to run malicious actions. In case of the re-implemented sandbox system, special knowledge of the Java programming language is needed to understand the security implications of occurring sensitive operations. This is similar in case of Rosemary [32] since operators must decide on the system call level if a specific action is allowed, which also requires special knowledge to understand the security implications. Although we assume experts operating SDN controllers, we should consider that network operators might not have such knowledge. In particular, since 50% to 80% of network outages occur due to human error [20], misconfiguration is likely to happen in practice. A possible solution could be provided by high-level permissions such as *read topology*, *add flow rule* or *register for packet-in message handling*. Such a design was proposed by Wen *et al.* [35] and is about to be implemented for the ONOS controller [24]. While such high-level permissions could ease correct sandbox configuration in practice, operators must still guess if such permissions are used in a benign or malicious way.

For future work, we suggest systems similar to current breach detection systems, thus, preventing malicious behavior inside the network operating system. In addition, analysis systems (*e. g.*, NICE [3]) are desirable for current SDN controllers, especially for the Java/OSGi-based ones, since the SDN controllers of many large companies are build upon Java and OSGi. Such an analysis system could be integrated, for instance, in the process of adding SDN applications to a SDN App Store.

7 Related Work

The problem of malicious SDN applications was first addressed by Porras *et al.* [29]. They provide a security kernel for the NOX controller with the focus on detecting and resolving dynamic flow rule tunneling. An enhanced version [28] provides additional security features such as role-based authorization, an OpenFlow-specific permission model and inline flow-rule conflict resolution. While these efforts concentrate on the data plane protection provided on control layer, our SDN rootkit is able to bypass control layer security and, therefore, to compromise the data plane.

Shin *et al.* [32] and Röpke *et al.* [30] provide sandbox techniques to protect the control layer against SDN applications which either present flaws or rudimentary malicious logic. Each of the counteracting sandbox mechanism is based on low-level permissions (*i. e.*, on system call level and on sensitive Java function level) and, therefore, requires special knowledge of the regarding operating systems or the Java programming language. As network operators may have little knowledge regarding these system-level aspects, the possibility of misconfiguration in practice evokes the question of sophisticated attacks through malicious SDN applications.

High-level permission models have been discussed earlier in the literature [35] whereas implementation of similar models in SDN controllers is targeted recently [24, 28]. We believe that these do not entirely solve the problem of sophisticated malicious SDN applications, since operators must still guess whether a critical operation is used in a benign or malicious way, independent of the level's degree.

Finally, we believe that an analysis of SDN applications combined with policy checking at runtime can potentially reveal sophisticated malicious SDN applications. However, current policy checkers [13, 14] focus rather on finding network invariants than on detecting SDN rootkits. Similarly, the analyis system NICE [3] concentrates on finding bugs in OpenFlow applications but not on revealing SDN malware. Approaches for precise analysis of Java reflections [2, 18] could assist for preliminary analysis.

8 Conclusion

Software-defined networking is an emerging technology which crucially depends on secure network operating systems. Commercial network operating systems are often built upon Java and OSGi, for instance, in case of either large enterprises such as Cisco and HP or open source versions such as the industry's leading open source OpenDaylight controller and

the carrier-grade ONOS controller. Taking this into consideration, we investigate sophisticated malware attacks against such modern NOS. In particular, we identify new challenges regarding SDN rootkits and develop a new technique for providing remote access by leveraging traditional rootkit techniques and combining these with the SDN specific features. Additionally, we demonstrate via our prototype implementations that modern NOS are vulnerable to sophisticated SDN malware. Even in the presence of available security mechanisms, we are able to subvert and compromise this Achilles' heel of SDN-based networks. Hence, sophisticated SDN malware should be included in the threat model of NOS instead of only considering rudimentary malicious logic.

References

1. S. T. Ali, V. Sivaraman, A. Radford, and S. Jha. A Survey of Securing Networks using Software Defined Networking. *IEEE Transactions on Reliability*, 64(3), 2015.
2. E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *International Conference on Software Engineering*, ICSE, 2011.
3. M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012.
4. Cisco. Extensible Network Controller. `www.cisco.com/c/en/us/products/collateral/cloud-systems-management/extensible-network-controller-xnc/data_sheet_c78-729453.html`.
5. M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Symposium on Network and Distributed System Security*, 2015.
6. N. Feamster, J. Rexford, and E. Zegura. The Road to SDN. *ACM Queue: Tomorrow's Computing Today*, 2013.
7. Floodlight. `floodlight.openflowhub.org`.
8. N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, 2008.
9. Hewlett-Packard. HP VAN SDN Controller. `www.hp.com`.
10. Hewlett-Packard. HP Open Ecosystem Breaks Down Barriers to Software-Defined Networking. `www.hp.com`, 2013.
11. U. Hölzle. OpenFlow @ Google. Open Networking Summit, 2012.
12. S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Symposium on Network and Distributed System Security*, 2015.
13. P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.
14. A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

15. D. Kreutz, F. Ramos, and P. Verissimo. Towards Secure and Dependable Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.
16. 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*, 2015.
17. B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2010.
18. B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *Asian Conference on Programming Languages and Systems*, APLAS, 2005.
19. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 2008.
20. J. Networks. Whats behind network downtime? `www-935.ibm.com/services/au/gts/pdf/200249.pdf`, 2008.
21. Nick McKeown. How SDN will shape networking. Open Networking Summit, 2011.
22. B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys & Tutorials*, 2014.
23. ONF. Open Networking Foundation. `www.opennetworking.org`.
24. ONOS. Open Network Operating System. `http://onosproject.org/`.
25. Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. White paper, Open Networking Foundation, 2012.
26. Oracle. Sun Alert 1000560.1. `www.oracle.com`, last update in 2008.
27. Oracle. Sun Alert 1000148.1. `www.oracle.com`, last update in 2010.
28. P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran. Securing the Software-Defined Network Control Layer. In *Symposium on Network and Distributed System Security*, 2015.
29. P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A Security Enforcement Kernel for OpenFlow Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2012.
30. C. Röpke and T. Holz. Retaining Control Over SDN Network Services. In *International Conference on Networked Systems*, 2015.
31. S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Symposium on Network and Distributed System Security*, 2013.
32. 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 *ACM SIGSAC Conference on Computer and Communications Security*, 2014.
33. S. Shin, V. Yegneswaran, P. Porras, and G. Gu. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *ACM Conference on Computer and Communications Security*, 2013.
34. S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz. Dynamic hooks: hiding control flow changes within non-control data. In *USENIX Security Symposium*, 2014.
35. X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang. Towards a Secure Controller Platform for OpenFlow Applications. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.