THE (IN)SECURITY OF TOPOLOGY DISCOVERY IN OPENFLOW-BASED SOFTWARE DEFINED NETWORK

Talal Alharbi, Marius Portmann and Farzaneh Pakzad

School of ITEE, The University of Queensland, Brisbane, Australia

ABSTRACT

Networking (SDN) is a new paradigm for configuring, controlling and managing computer networks. In SDN's logically centralized approach to network control, a reliable and accurate view of the network topology is absolutely essential. Most SDN controllers use a de-facto standard topology discovery mechanism based on OpenFlow to identify active links in the network. This paper evaluates the security, or rather lack thereof, of the current SDN topology discovery mechanism. We discuss and demonstrate its vulnerability to a simple link spoofing attack, which allows an attacker to poison the topology view of the controller. The feasibility of the attack is verified and demonstrated via experiments, and its impact on higher layer services is evaluated, via the example of shortest path routing. The paper finally discusses countermeasures, and implements and evaluates the most promising one.

KEYWORDS

Software Defined Network, Topology discovery, Security, POX

1. INTRODUCTION

Software Defined Networking (SDN) is a new approach to manage computer networks that have recently gained tremendous momentum [1, 2, 3].One of the essential concepts of SDN is the separation of the control and data plane. The control' intelligence', which determines how packets are being forwarded, is removed from routers and switches (forwarding elements), and placed in a (logically) centralised entity, i.e. the SDN controller.

Figure 1 illustrates the general SDN architecture [4]. The bottom layer, i.e. the infrastructure layer, consists of a set of connected forwarding elements, i.e. SDN switches, which provide basic packet forwarding functionality (data plane).

The middle layer is the control layer, consisting of a logically centralized SDN controller, implementing the functionality of a Network Operating System (NOS) [5].The NOS deals with and hides the distributed nature of the physical network and provides the abstraction of a network graph to higher layer services, which sit at the application layer of the SDN architecture [6]. This abstraction makes the network much more programmable and simplifies the implementation and deployment of new network services and applications.

The SDN controller manages and configures individual SDN switches by installing forwarding rules via the so called south bound interface [7].The pre-dominant standard for this is Open Flow[8], which we will focus on in this paper. At the top of the SDN architecture is the application layer, where high level network policy decisions are defined and applications and services such as traffic engineering, routing, security, etc. are implemented .The interface between the Application layer and the control layer is referred to as the northbound interface

DOI: 10.5121/ijnsa.2018.10301

[7].In contrast to the southbound interface, there is currently no well-established standard for this [9].

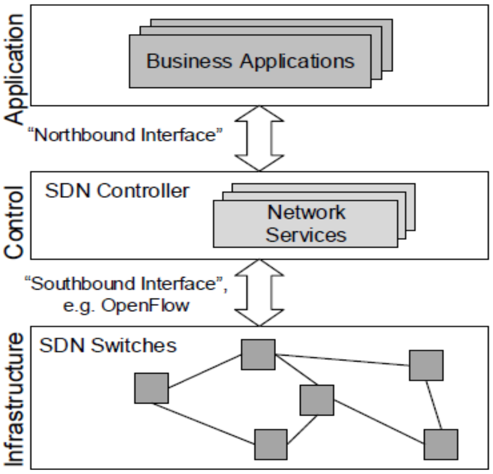


Figure 1. SDN Architecture [4]

In the SDN architecture, it is essential for higher layer services such as routing to have an accurate and up to date view of the network topology. Providing network topology information to the application layer is one of the key network services for which a NOS, i.e. SDN control layer is responsible for.

While there is no official standard for an SDN topology discovery mechanism, there is de-facto standard, which is sometimes informally referred to as Open Flow Discovery Protocol (OFDP) [10, 11]. All major SDN controllers implement it in essentially the same way, most likely due to the fact that it has been adopted from NOX, the original SDN controller [5].

The problem with OFDP is that it is fundamentally insecure, as is demonstrated in this paper. We show how an attacker can poison the topology view of the SDN controller and creates proofed links by crafting special control packets and injecting them into the network via one or more compromised hosts.

We show the feasibility of the attack, both via net work emulation as well as test-bed experiments, for both POX[12]and Ryu[13], two widely used SDN controller platforms. We further demonstrate and evaluate the impact of the link spoofing attack on higher layer services. We use shortest path routing as a case study and show that an attacker can relatively easily cause significant disruption of network connectivity. The impact and level of connectivity disruption is quantified for two example scenarios. A final contribution of the paper is the discussion and evaluation of countermeasures against the vulnerability.

The remainder of the paper is organized as follows. Section 2 presents the relevant background on SDN and Open Flow, and describes in detail the operation of OFDP, the current state-of-the-art topology discovery mechanism in SDN. Section 3describesthevulnerabilityof OFDP and demonstrates a number of link spoofing attacks. Section 4 investigates the impact of the attack on higher layer services, based on the example of routing. Section5 discusses counter measures, Section6 discusses related works, and Section7 concludes the paper.

2. BACKGROUND

2.1 OPENFLOW

Open Flow [3]is the predominant south bound interface protocol for SDN. It provides the interface between the infrastructure layer and the control layer, as shown in Figure 1.Open Flow is a wire protocol that allows the SDN controller to configure switches, i.e. via the installation of packet forwarding rules [14, 15,16] .The protocol also allows switches to notify the controller about special events, e.g. the receipt of a packet that does not match any installed rules.

At the time of writing this paper, the latest edition of the Open Flow standard is version1.5 [17].However, the following discussion of the topology discovery method and its security is version agnostic and relevant to all versions of Open Flow.

Open Flow switches are assumed to be configured with the IP address and TCP port number of their assigned SDN controller. At initialization, switches contact the SDN controller via this address and port and establish a secure Transport Layer Security (TLS) connection. As part of the initial protocol handshake, the controller sends an Open Flow OFPT FEATURES REQUEST message to each switch, requesting configuration information, including the number of switch ports and corresponding MAC addresses.

This initial handshake informs the controller the existence of the nodes (switches) in the network, but it does not provide any information about the active inter-switch links. Gathering this information is the role of OFDP.

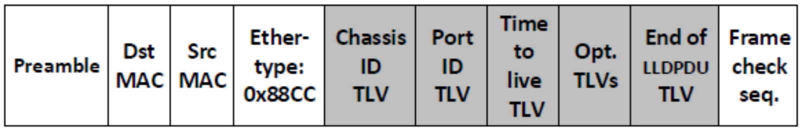
As mentioned above, Open Flow allows controllers to access and configure the forwarding rules (flow tables) in SDN switches, and these rules provide fine grained control over how packets are forwarded through the network. Open Flow switches support a basic match-action paradigm, where each incoming packet is matched against a set of rules, and the corresponding action or action list is executed. The supported match fields include the switch ingress port and various packet header fields, such as IP source and destination address, MAC source and destination address, etc.

One of the main actions supported by an Open Flow switch is forwarding a packet on a particular switch port. Switch ports can either be physical ports or canal so is one of the following virtual port types: ALL (sends packet out on all physical ports), CONTROLLER (sends packet to the SDN controller), FLOOD (same as ALL, but excluding the ingress port).

To send data packet to the controller, an SDN switch encapsulates the packet in an Open Flow Packet-In message .For example, this is used if the switch receives a packet for which it has no matching forwarding rule. Open Flow also supports a Packet-Out message, via which the controller can send a data packet to a switch, together with instructions (action list) on how to forward it. Both Open Flow Packet-In and Packet-Out messages are essential for the topology discovery mechanism discussed in the following section [3].

2.2 TOPOLOGY DISCOVERY

Topology discovery is an essential service in SDN and under pins many higher layer services. In this context, when we refer to topology discovery, we actually mean link discovery, since the controller learns about the existence of network nodes (switches)by other means, as discussed above.



Preamble	Dst MAC	Src MAC	Ether- type: 0x88CC	Chassis ID TLV	Port ID TLV	Time to live TLV	Opt. TLVs	End of LLDPDU TLV	Frame check seq.

Figure 2. LLDP Frame Structure

Open Flow switches themselves do not support any topology (link) discovery functionality, and it therefore needs to be implemented as a service at the controller. There is currently no official standard for topology discovery in SDNs based on Open Flow. However, there is a de-facto standard, since all the major SDN controller platforms implement topology discovery in essentially the same way, derived from the topology discovery mechanism in NOX[5].This mechanism is sometimes referred to in formally as OFDP(OpenFlow Discovery Protocol) in [10, 11],and for lack of an official term, we will use it in this paper.

OFDP uses the frame format defined in the Link Layer Discovery Protocol (LLDP) [18], designed for link and neighbor discovery in Ethernet networks. However, with the exception of the frame format, OFDP has not much in common with LLDP, and operates quite differently.
The format of an LLDP frame, as used in OFDP, is shown in Figure 2.TheLLDP payload is encapsulated in an Ethernet frame with the Ether Type field set to 0x88CC.The Ethernet frame contains an LLDP Data Unit (LLDPDU) (shaded in greyinFigure2),which has a number of type-length-value(TLV)fields. The mandatory TLVs include Chassis ID, which is a unique switch identifier, Port ID, a port identifier, and a Time to live field. These TLVs can be followed by a number of optional TLVs, and an End of LLDPDUTLV.

Open Flow switch, due to its lack of control intelligence and autonomous operation, cannot initiate the sending and processing of link discovery packets, as is the case for traditional Ethernet switches, and as specified in the LLDP standard[18].In SDN, link discovery is initiated by the controller. How this works in OFDP is illustrated via a basic example scenario shown in Figure3.Initially The SDN controller creates a dedicated LLDP packet for each port on each switch, in our example, a packet for portP1, a packet for port P2 and one for port P3on switchS1. All these LLDP packets have their Chassis ID and Port IDTLVs initialized accordingly.

The controller then uses a separate Open Flow Packet-Out message to send each of the LLDP packets to switch S1. Every Packet-Out message also includes an action, which instructs the switch to forward the packet via the corresponding port. For example, the LLDP packet with Port ID = P1 will be sent out on port P1, the packet with Port ID = P2 on port P2, etc.

Switches are pre-configured with a rule which states that any received LLDP packets are to be sent to the controller via an Open Flow Packet-In message. As an example, we consider the LLDP packet which is sent out on port P1 on switch S1 and is received by switch S2 via port P3 in Figure3.According to the pre-installed rule, switch S2 sends the LLDP packet to the controller, encapsulated in a Packet-In message. This Packet-In message also contains additional meta data, such as the ingress port where the packet was received, as well as the Chassis ID of the switch sending the Packet-In message. This information, combined with information about the origin switch and port, contained in the payload of the LLDP packet (Chassis ID and Port ID TLVs) can be used by the controller to infer the existence of a link between (S1, P1) and (S2, P3).

This process is repeated for every switch in the network, i.e. the controller sends a separate Packet-Out message with a dedicated LLDP packet for each port of each switch, allowing it to discover all available links in the network. The entire process is repeated continuously, with a typical discovery interval of 5 seconds [12].

Most current SDN controller platforms such as NOX [5], POX [12], Floodlight [19], Ryu [13] and Beacon [20] implement the OFDP discovery mechanism as described above. A study of the Source code of the different implementations reveals only very minor variations, for example in regard to the timing of the sending of the Packet-Out messages, or the encoding of Chassis ID and Port ID information in LLDP packets.

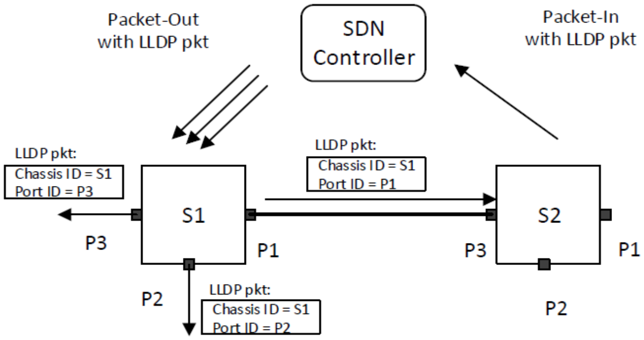


Figure 3. Basic OFDP Example Scenario

3. OFDP LINK SPOOFING - BASIC VULNERABILITY

The basic security problem with the current SDN topology discovery mechanism (OFDP) is that there is no authentication of LLDP control messages. Any LLDP packet received by the controller is accepted and link information contained in it is used to update the controller's topology view. More specially, OFDP lacks the following two checks:

- OFDP does not check or enforce that LLDP packets are only received via switch ports that are connected to another switch. Instead, LLDP packets from host ports are also accepted and are forwarded to the controller.

- There is no authentication or integrity check of LLDP control messages. The controller has no way of verifying the origin of the packets.

As a result, it is relatively easy for an attacker to inject fabricated (spoofed)LLDP control messages into the network, and thereby corrupting the topology information of the controller. We illustrate this via a simple example, shown in Figure 4. In this scenario, we assume that host h1 has been compromised by an attacker, who aims to create a fake link between switches S1and S3.

The attack can be broken down into the following steps:

1. Host h1 injects an LLDP packet via port P1 on switch S1, where h1 is attached. The injected packet follows the structure shown in Figure 2, but with the Chassis ID TLV set to S3, and the Port ID set to P1.
2. Switch S1 receives the LLDP packet from h1, and following its installed rule, forwards the packet to the controller, encapsulated in an Open-Flow Packet-In message. Switch S1 adds information to the Packet-In message, i.e. its own Chassis ID and the Port ID of the ingress port via which the LLDP packet was received at switch S1. In our scenario, this information is (S1, P1).

The controller receives the LLDP packet plus the additional information added by S1. It identifies the source of the LLDP packet, and therefore the origin of the link from the TLVs in the payload as (S3,P1). The information about the other end of the link is taken from the metadata of the Packet-In message, and is identified as (S1, P1). From this information, the controller concludes (wrongly) that there exists a link between (S3, P1) and (S1, P1)

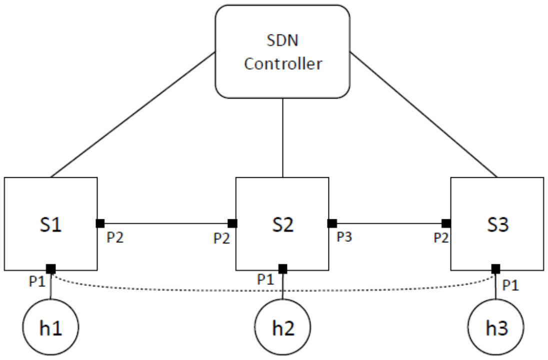
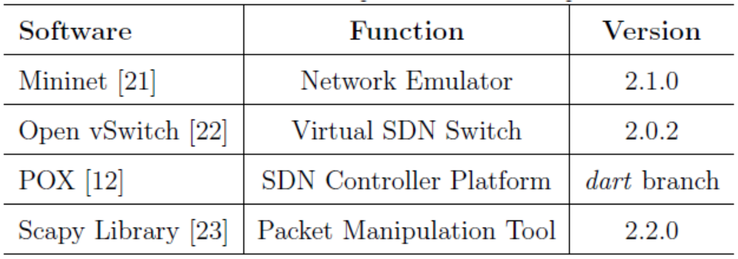


Figure 4. Basic Attack Scenario

Table 1. Software used in Implementation and Experiments.



Software	Function	Version
Mininet [21]	Network Emulator	2.1.0
Open vSwitch [22]	Virtual SDN Switch	2.0.2
POX [12]	SDN Controller Platform	*dart* branch
Scapy Library [23]	Packet Manipulation Tool	2.2.0

3.1 EXPERIMENTAL VALIDATION – MININET

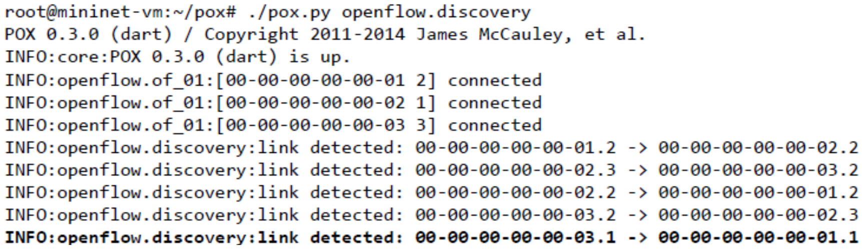
In order to validate the feasibility of the link spoofing attack experimentally, we used Mininet [21], a Linux based network emulator which allows the creation of a network of virtual SDN switches and hosts, connected via virtual links. We used Open v Switch [22], a software Open Flow switch which is supported in Mininet.

In our initial experiment, we used the POX controller platform and its implementation of OFDP, i.e. the open flow. discovery component. In order to craft the special LLDP packet for the attack, we wrote a packet generator in Python, based on the Scapy library [23]. Table 1 summarizes the relevant software tools that we used in our experiments. The Mininet experiments were run on a standard PC, with a 3 GHz Intel Core 2 Duo CPU with 4 GB of RAM, running Ubuntu Linux with kernel version 3.13.0.

Figure 5 shows the debug output of the POX controller, and in particular the open flow. Discovery component which implements OFDP. No other POX component was running in this experiment. From the output, we see that our three switches have connected to the controller, with Chassis ID of 00-00-00-00-00-01 for switch S1, 00-00-00-00-00-02 for switch S2 and 00-00-00-00-00-03 for switch S3. This debug output is generated by the main POX component.

The last five lines of the output are from the open flow. Discovery component. Each line indicates the detection of a unidirectional link, caused by the reception of a corresponding LLDP packet at the controller. For example, the first of these lines indicates that a link from (S1, P2) to

(S2, P2), i.e. from port P2 on switch S1 to Port P2 on switch S2, has been detected. The next line indicates a link from (S2, P3) to (S3, P2). The following two lines indicate the detection of the same links in the reverse direction. This is consistent with our topology, as shown in Figure 4.
The interesting part in Figure 5 is the last line (in bold), which appears after we run the attack by injecting the fabricated LLDP packet from hosth1 to switch S1. The line indicates that a non Existent link from (S3, P1)to (S1, P1) is detected by the controller, and hence the link spoofing attack has been successful. When we look at the topology view of the controller, which is stored as a list of links, we see that the link has indeed been added.



```
root@mininet-vm:~/pox# ./pox.py openflow.discovery
POX 0.3.0 (dart) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.3.0 (dart) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
INFO:openflow.of_01:[00-00-00-00-00-03 3] connected
INFO:openflow.discovery:link detected: 00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-02.3 -> 00-00-00-00-00-03.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-03.2 -> 00-00-00-00-00-02.3
INFO:openflow.discovery:link detected: 00-00-00-00-00-03.1 -> 00-00-00-00-00-01.1
```

Figure 5. POX Debug Information (Mininet)

It is important to note that the attacker can spoof the origin of the link (switch and port) arbitrarily, simply by setting the relevant LLDP TLVs accordingly. However, the link destination information is added as metadata to the Packet-In message by the ingress switch, and hence cannot be changed by the attacker. For our example, this means that the spoofed link that h1 can create are limited to the set of unidirectional links terminating at port P1 on switch S1.If an attacker wants to create a spoofed bidirectional link, say from S1 toS3 in our example, he/she needs to control both h1 and h3. We will discuss this in more detail in Section 4.

We also performed the above attack with the Ryu SDN controller, with identical results. However, we had to slightly modify our LLDP packet generation script, since Ryu uses a different encoding format for the Chassis IDand Port ID values in the LLDP packet.

.

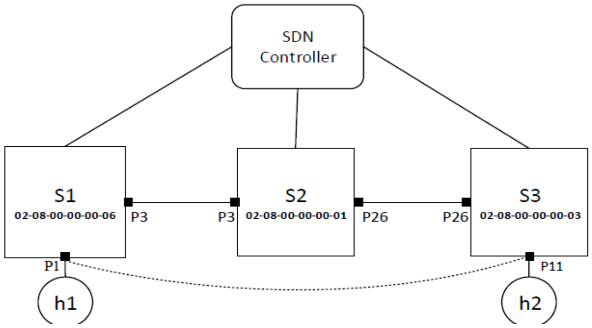
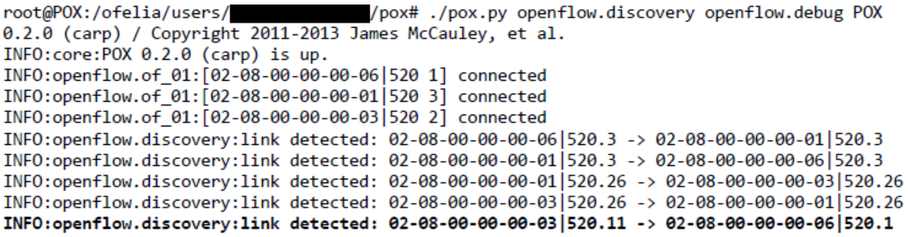


Figure 6. Basic Attack Scenario OFELIA

3.2 Experimental Validation – OFELIA

In addition to our emulation-based experiments using Mininet, we also conducted the same experiment on the OFELIA SDN test-bed [24]. OFELIA is a federated SDN test-bed distributed across a number of sites, or islands, in a range of European countries, including the UK, Switzerland, Germany, Belgium, Spain and Italy. Each island is equipped with a range of SDN hardware switches, supporting the Open Flow 1.0 standard. The model of the switches used in our experiment was NEC IP8800//S3640-24T2XW. We used the resources located at the OFELIA island in Trento, Italy.

Our goal was to replicate the topology shown in Figure 4. We managed to do this, with the exception of a small detail. OFELIA provides only 3virtual machines to experimenters, of which one is used for the controller, leaving only two for the use as hosts. Figure 6 shows our OFELIA topology, with the corresponding Chassis IDs and Port IDs. The main difference to Figure 4 is That the host attached to S2 is missing, which is not a problem, since it does not play an active role in this attack scenario.



```
root@POX:/ofelia/users/[redacted]/pox# ./pox.py openflow.discovery openflow.debug POX
0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[02-08-00-00-00-06|520 1] connected
INFO:openflow.of_01:[02-08-00-00-00-01|520 3] connected
INFO:openflow.of_01:[02-08-00-00-00-03|520 2] connected
INFO:openflow.discovery:link detected: 02-08-00-00-00-06|520.3 -> 02-08-00-00-00-01|520.3
INFO:openflow.discovery:link detected: 02-08-00-00-00-01|520.3 -> 02-08-00-00-00-06|520.3
INFO:openflow.discovery:link detected: 02-08-00-00-00-01|520.26 -> 02-08-00-00-00-03|520.26
INFO:openflow.discovery:link detected: 02-08-00-00-00-03|520.26 -> 02-08-00-00-00-01|520.26
INFO:openflow.discovery:link detected: 02-08-00-00-00-03|520.11 -> 02-08-00-00-00-06|520.1
```

Figure 7. POX Debug Information (OFELIA)

After configuring the topology, we conducted the same experiment as discussed in Section 3.1, with the same POX controller code and configuration. We also used the same packet injection code, with a small modification account for the different interface name on host h1.

Figure 7 shows the debug output from POX. We see that bidirectional links are created between port P3 on switch S1 (02-08-00-00-00-06), and port P3 on switch S2 (02-08-00-00-00-01), as well as between port P26 on switch S2 (02-08-00-00-00-01), and port P26 on switch S3 (02-08-00-00-00-03).The last line again indicates that the attack was successful, and that there was a link created between S1 and S3. We have also verified the creation of the fake link in the controller's topology view. As with Mininet, we have replicated the attack for Ryu, with the same results.

Our experiments have demonstrated the basic vulnerability of OFDP, the predominant SDN topology discovery mechanism. As mentioned earlier, topology discovery is an essential network service provided in SDN, upon which a lot of other services and applications rely. In the following section, we will discuss the potential impact of the vulnerability on such higher layer services, using the example of shortest path routing.

4. IMPACT ON ROUTING

Routing is a key network application that relies on the controller having an up to date and accurate topology view. Shortest path routing in SDN is relatively trivial, compared to traditional networks. The challenge of dealing with a physically distributed system is done by the topology discovery component, implemented by the controller platform. Given a network topology as a graph, shortest path routing is essentially just computing a shortest path between source and destination nodes, e.g. via Dijkstra's algorithm [25].In the following, we will evaluate and quantify the impact of the attack on routing via two topology examples.

4.1 Linear Topology

For our next experiment, we consider a simple linear topology with 5switches and a single host attached to each switch, as shown in Figure 8. As before, we assume host h1 is the attacker, which in this case injects a fabricated LLDP packet with the aim of creating a false (unidirectional) link between (S5, P1) and (S1, P1). This spoofed link is shown as a dashed line in Figure 8. As described in the previous section, the attacker simply needs to set the Chassis ID to S5, and the Port ID TLV to P1 in the fabricated LLDP packet for this attack.

We created this topology in Mininet and used the layer 2 shortest path routing component in POX (l2 multi.py) for our experiment. This POX component computes shortest paths among node pairs using the Floyd-Warshall algorithm [26].

Prior to launching the attack, we performed a ping test among all host pairs, in order to verify the connectivity. This was achieved via the ping all command in Mininet. We saw that we had 100% connectivity, and each host could reach every other host. After launching the attack from host h1, Which injected the fabricated LLDP packet, we verified that the topology discovery service had indeed added a link from (S5, P1) to (S1, P1) to the controller topology database.

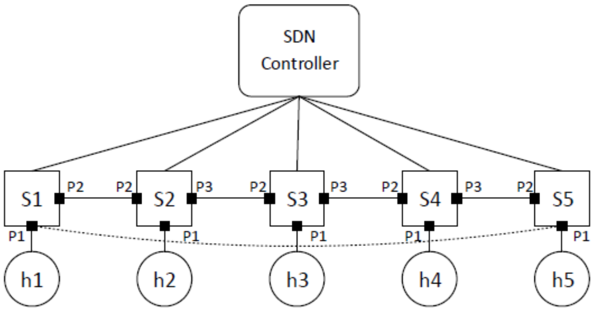


Figure 8. Linear Topology for Routing Experiment

However, after running pin fall again, we still saw a connectivity of 100%.Inspection of the source code of l2 multi.py reveals that only bi-directional links are considered for path computation, which is why the attack was unsuccessful. In this case, in order for the attacker to be able to disrupt network connectivity, he/she needs to spoof a bidirectional link. This is impossible to achieve with control over only a single host, due to the fact that only one end point of the spoofed link, i.e. the source, can be chosen by the attacker. The other end of the link is determined by the ingress port and switch where the LLDP packet is injected by the attacker. Another way of expressing this is that an attacker controlling a single host can create unidirectional links starting at any source, but they all have to terminate at the switch and port where the attacking host is connected.

In order to establish a bidirectional link, an attacker needs to control at least two hosts. In this new attack scenario, we assume the attacker has compromised and controls hosts h1and h5. As in the previous case, h1 injects an LLDP packet with the Chassis ID and Port ID set to S5 and P1,creating the unidirectional link from (S5, P1) to (S1, P1). In addition, h5injects an LLDP packet with the source set to (S1, P1), creating the link in the reverse direction.

We ran the pair wise ping test again, and the result is shown in Table 2.Theleftmost column indicates the source, and the topmost row indicates the destination of the ICMP echo request message sent by ping. A '1' in the table indicates that there is connectivity between the two hosts in the corresponding row and column, and a '0' indicates a lack of connectivity. We can see that connectivity between a pair of hosts is disrupted when the spoofed link (S1-S5) is part of the shortest path between the two hosts, as expected.

Table 2. Connectivity After Attack.

ping dst ping src	h1	h2	h3	h4	h5
h1	■	1	1	0	0
h2	1	■	1	1	0
h3	1	1	■	1	1
h4	0	1	1	■	1
h5	0	0	1	1	■

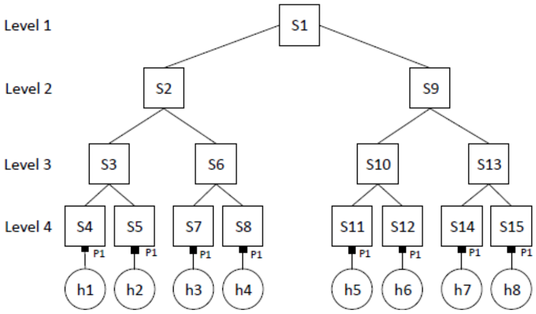


Figure 9: Tree Topology for Routing Experiment

This attack on the topology discovery mechanism significantly disrupt connectivity for routing. In this scenario almost 30% of all links are disrupted by the creation of a single spoofed bidirectional link. We have also replicated the experiment in OFELIA, with an identical outcome.

While often shortest path routing assumes bidirectional links, such as the POX component we have considered here, this is a design decision rather an absolute requirement. Other implementations, such as [27], also consider unidirectional paths in their computation of shortest paths. In this case, an attacker only needs to control a single host, and is able to disrupt connectivity via creating unidirectional spoofed links.

4.2 TREE TOPOLOGY

We also considered a tree topology, as shown in Figure 9. The topology consists of 15 switches, organized in a binary tree of depth 4 and fan-out2, with a host attached to each leaf switch. In this experiment, we tried to quantify the connectivity disruption impact of the link spoofing attack.

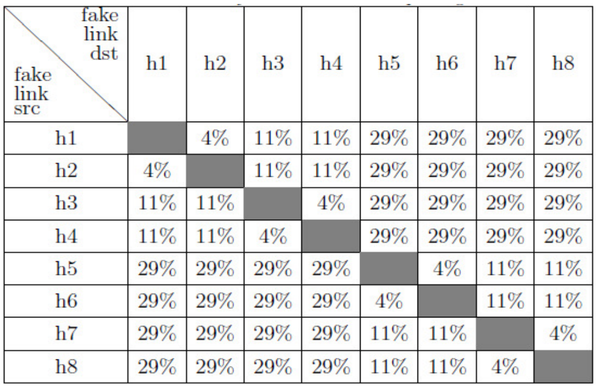
We created bidirectional fake links between all host pairs, one at the time. For each of those scenarios, i.e. for each case with a different fake link in the network, we ran a complete ping test (ping all) between all host pairs, resulting in a total of 56 pings. We then considered how many of those pings failed due to the creation of each individual fabricated link. The results are shown in Table 3, which shows the percentage (rounded to the nearest integer) of connectivity loss, i.e. failed pings, due to each of the possible fabricated links. For example, we see that if a single fake link is generated between hosts h1 and h8, 29% of the connectivity between host pairs is disrupted.

We observe that there are only 3 distinct values in the table, 4%, 11%and 29%, which correspond to 3 different scenarios in our example. We get4% (rounded) in the case where the fake link is between 2 hosts that are attached to the same switch at level 3 in the topology, for example hosts h1and h2, or hosts h3 and h4, etc. In this case, the spoofed link forms part of the shortest path for only 2 out of the total of 56 ping paths, i.e. from h1 toh2, and h2 to h1.For the case where the fake link is between two host connected via a common switch at level 2 of the topology, e.g. hosts h1 and h3 via switchS2, we observe that the fake link is part of the shortest path of 6 out of the total of 56 source -destination pairs, resulting in a loss of 11% connections.

Finally, if a fake link is created between hosts that are located in different main branches of the tree, i.e. if they are connected via switch S1 at level 1of the topology, 29% (16 out of 56) connections are disrupted. This is due to the fact that the spoofed link provides a shorter path or shortcut for a larger number of source -destination pairs.

The exact numbers are obviously specific to this particular example. However, it demonstrates the potential power of the link spoofing attack in terms of connectivity disruption for routing in SDN. By creating multiple spoofed links, the impact would obviously be compounded. The example also shows that by targeting the location of the spoofed link(s) in the network topology, an attacker can maximize the impact.

Table 3. Connectivity Loss Due to Link Spoofing Attack.



fake link dst fake link src	h1	h2	h3	h4	h5	h6	h7	h8
h1	■	4%	11%	11%	29%	29%	29%	29%
h2	4%	■	11%	11%	29%	29%	29%	29%
h3	11%	11%	■	4%	29%	29%	29%	29%
h4	11%	11%	4%	■	29%	29%	29%	29%
h5	29%	29%	29%	29%	■	4%	11%	11%
h6	29%	29%	29%	29%	4%	■	11%	11%
h7	29%	29%	29%	29%	11%	11%	■	4%
h8	29%	29%	29%	29%	11%	11%	4%	■

4.3 DISCUSSION

We have demonstrated the vulnerability of OFDP translates into the vulnerability of routing, which relies heavily on the topology discovery services provided by the controller. An attacker can relatively easily disrupt network connectivity, since the attacker only needs to gain control over a small number of hosts. This is typically much easier to achieve than gaining control over network infrastructure devices, i.e. switches or controllers. Since all the open source controller platforms that we have investigated implement OFDP in essentially the same way, the vulnerability discussed in this paper is shared by all of them, and hence the problem is significant.

We have used the example of routing to demonstrate the potential impact of the link spoofing attack, but it is clear that other services that rely on the topology discovery service are also vulnerable to the attack. We have done some preliminary investigation into the spanning tree component in POX and found it to be vulnerable as well. This is critical, given the fact that a lot of services rely on the spanning tree mechanism, such as the L2 learning switch component in POX.

5. COUNTERMEASURES

As mentioned previously, the vulnerability of OFDP is due to a lack of any checks about the origin of received LLDP packets. We discuss two basic approaches to address this.

5.1 CONTROLLER CHECKS

The link spoofing attack, as presented in this paper, could be rendered impossible if LLDP packets were only accepted via switch ports that connected to other switches. This check could be implemented via installing a simple rule on each switch, or alternatively, it could be performed at the controller. The problem with this approach is that it assumes knowledge about each port on each switch, and to what type of node it is connected to. In a network with a dynamic topology, there is no simple and secure way to keep track of that, to the best of our knowledge. For example, services that keep track of hosts, such as the host tracker component in POX, are themselves vulnerable to attacks [28], and hence cannot be assumed to provide reliable information.

5.2 LLDP PACKET AUTHENTICATION

The problem of lacking the authenticity of LLDP messages can relatively easily be overcome by adding a cryptographic Message Authentication Code (MAC) to each LLDP packet that the controller sends out. Such a MAC provides authentication as well as integrity for each packet. We have implemented this mechanism in POX using HMAC, a keyed-hash based message authentication code [29]. The MAC is computed as follows:

$$\mathrm{HMAC}(K,m) = h((K opad)|h(K\ |ipad)|m)$$

K is the secret key, and m is the message over which the HMAC is calculated. In our case, m consists of the relevant LLDP TLVs, i.e. the Chassis ID and the Port ID. h() is a cryptographic hash function, '|' denotes concatenation and denotes the XOR operation. opad and ipad are constant padding values [29].

It is important to note that the basic HMAC is vulnerable to replay attacks. In the scenario shown in Figure 4, a replay attack can be used against a topology discovery mechanism protected with a basic HMAC. The attack requires control over two hosts, e.g. hosts h1 and h3. As part of the normal OFDP protocol, host h1 will receive LLDP packets with Chassis ID set to S1 and Port ID set to P1. Here, we assume the LLDP packet is secured with a HMAC, computed over the relevant LLDP TLVs, using the secret key K. Host h1 can then send this LLDP packet to its colluding partner hosth3 via an out-of-band channel. h3 then injects the packet to switch S3 via port P1. Since the packet has a valid MAC, the controller accepts it, and a spoofed link from (S1, P1) to (S3, P1) is successfully created. The reverse link can be created in the same way. We have implemented this attack and verified its feasibility.

The traditional approach to prevent replay attacks in HMAC is via the use of a unique message identified (or nonce) over which the HMAC is computed ,to ensure that each HMAC value is unique. This message identifier needs tobe sent as clear text to the receiver as part of the message, causing additional overhead.

We therefore use an alternative approach in our implementation. Instead of using a unique message identifier, we replace the static secret key *K* with a dynamic value *Ki,j* , which is randomly chosen for every single LLDP packet*i*, in every topology discovery round *j*. The best chance for an attacker to compute a valid MAC and launch a successful link spoofing attack, is via guessing the correct value of the random numbers *Ki,j* . This is virtually impossible if we use a high quality random number generator that provides sufficient entropy. In addition, any wrong guess by an attacker can easily be detected by the controller.

In order to verify the authenticity of a received LLDP packet and compute its HMAC value, the controller needs to know the corresponding value of *K i,j*. This is achieved by the controller keeping track of which key is used for which packet. The combination of Chassis ID and Port ID provides the necessary identifier. We used MD5 as our hash functions. While MD5 has been shown to be vulnerable to a range of collision attacks, it can still be considered sufficiently secure in the context of HMAC [29], since HMAC does not rely on the collision resistance property [30].It would obviously be trivial to replace MD5 with another hash function such as SHA3.

We have implemented this HMAC based mechanism in the topology discovery component in POX. To accommodate the MAC, we defined a new, optional TLV in the LLDP packet. We have conducted extensive tests and have verified that OFDP with the added HMAC (OFDP _HMAC) is indeed able to detect the injection of any fabricated LLDP packets from an attacker.

When creating an LLDP packet, the controller chooses a value N, computes HMAC, and adds the result to the TLV. Whenever the controller receives an LLDP packet via a Packet-In message, it computes the MAC over the relevant TLVs in the LLDP packet, using the corresponding value of *K i,j*. If the computed value matches the MAC in the packet, the authenticity of the packet is validated, and its content is used to update the controller's topology view. If not, the packet is discarded, and an alarm can be raised.

We have also evaluated the computational overhead on the controller caused by this mechanism. For this, we used a 2-node tree topology in Mininet, with fan-out 4 and depth 2, and ran both the original OFDP mechanism in POX, as well as OFDP _HMAC. Figure 10 shows the total cumulative controller CPU time used by each version, over an experiment period of 300 seconds. The experiment was repeated 20 times, and the figure shows the 95% confidence interval. The absolute values are not that interesting, since they are hardware dependent. However, we see that in relative terms, the overhead of HMAC adds an extra 8% in CPU load, to the low computational cost of the topology discovery mechanism. We believe this is an acceptable cost to pay for the increased level of security.

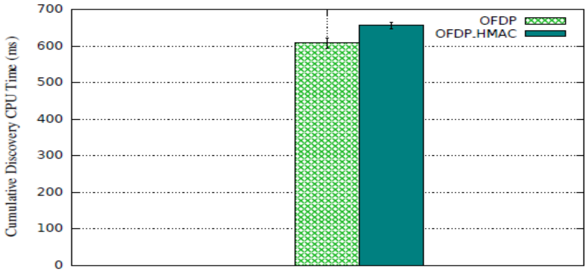


Figure 10. Computational Overhead of HMAC in OFDP

6. Related Works

There have been a number of works that address various security aspects of SDN. However, only very few recent papers have addressed the security of the core SDN service of topology discovery.

The paper [31] discusses a range of attacks against SDN, and proposes SPHINX, a generic SDN attack alert system, which compares network behavior with predefined or learned `normal' behavior, defined as policies. The paper also mentions the possibility of attacks against topology discovery via spoofing of LLDP packets, as discussed in our paper. The paper does not address the specific technical details of the attack, such as provided in our paper, nor does it explore and quantify the impact of the attack on network connectivity.

In [28], the authors also discuss a range of attacks against SDNs, including ARP spoofing attacks as well as link spoofing attacks. Due to the wide scope of the paper, it does not specifically consider, evaluate or quantify the impact of the link spoofing attack on routing and hence network connectivity, as we have done in our paper. Furthermore, in contrast to our paper, the experiments in [28] are limited to software switches only. The authors of [28] also discuss potential countermeasures against the link spoofing attack, and also suggest an HMAC based packet authentication mechanism. However, their proposed method uses a static secret key, without a nonce, for the computation of the HMAC, and is therefore vulnerable to replay attacks, as discussed in the previous section. Finally, a preliminary version of our paper has been published in [32].

7. CONCLUSIONS

Topology discovery is an essential service in SDN, and a variety of other services and applications, such as routing, rely on it. In this paper, we have discussed OFDP, the current de-facto standard of topology discovery in SDN, implemented by most SDN controller platforms. We have discussed the vulnerability of OFDP to link spoofing attacks, which only requires an attacker to have control over one or more hosts (physical or virtual) in the network.

Through experiments in both emulated network scenarios using Mininet, and a SDN test-bed (OFELIA), we have demonstrated the feasibility of the attack. We have shown that the controller's topology view can be successfully poisoned by the attacker, and non-existent links can be added to the controller's topology database. We have further evaluated the impact of this attack on the operation of an SDN, in particular with the example of routing.

We also discussed potential countermeasures and implemented a simple mechanism that provides authentication using a hash-based message authentication code, using HMAC. We have discussed and verified its ability to prevent the link spoofing attack discussed in this paper. Finally, we have measured the computational cost of implementing the measure.

REFERENCES

[1] N. McKeown, Software-defined networking, INFOCOM keynote talk.

[2] N. Feamster, J. Rexford, E. Zegura, The road to SDN, Queue 11 (12) (2013) 20.

[3] OpenFlow Switch Specification, Available from: https://www.opennetworking.org/software-defined-standards /specifications/ [28 April 2018].

[4] O. N. Fundation, Software-defined networking: The new norm for networks, ONF White Paper 2 (2012) 2-6.

[5] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, NOX: towards an operating system for networks, ACM SIGCOMM Computer Communication Review 38 (3) (2008) 105-110.

[6] S. Shenker, M. Casado, T. Koponen, N. McKeown, et al., The future of networking, and the past of protocols, Open Networking Summit 20 (2011) 1-30.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: enabling innovation in campus networks, ACM SIGCOMM Computer Communication Review 38 (2) (2008) 69-74.

[8] Open Networking Foundation, Available from: https://www.opennetworking.org [28 April 2018].

[9] M. Jammal, T. Singh, A. Shami, R. Asal, Y. Li, Software defined networking: State of the art and research challenges, Computer Networks 72 (2014) 74-98.

[10] F. Pakzad, M. Portmann, W. L. Tan, J. Indulska, Efficient topology discovery in software defined networks, in: Proc. of Signal Processing and Communication Systems (ICSPCS), 8th International Conference on, IEEE, 2014, pp. 1-8.

[11] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, I. Seskar, GENI: A federated testbed for innovative network experiments, Computer Networks 61 (2014) 5-23.

[12] S. Kaur, J. Singh, N. S. Ghumman, Network programmability using POX controller, in: Proc. of ICCCS International Conference on Communication, Computing & Systems, IEEE, no. s 134, 2014, p. 138.

[13] Ryu SDN Controller, Available from: https://osrg.github.io/ryu [28 April 2018].

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

[15] H. Kim, N. Feamster, Improving network management with software defined networking, IEEE Communications Magazine 51 (2) (2013) 114-119.

[16] F. Hu, Q. Hao, K. Bao, A survey on software-defined network and openflow: From concept to implementation, IEEE Communications Surveys & Tutorials 16 (4) (2014) 2181-2206.

[17] OpenFlow Switch Speci_cation Version 1.5.1 (Protocol Version 0x06), Available from: https://www.opennetworking.org/wpcontent/uploads/2014/10/openow-switch v1.5.1.pdf [28 April 2018].

[18] IEEE Standard for Local and Metropolitan Area Networks {Station and Media Access Control Connectivity Discovery, IEEE Std 802.1AB-2009 (Revision of IEEE Std 802.1AB-2005) (2009) 1-204.

[19] Floodlight SDN Controller, Available from http://www.projectoodlight.org/floodlight [28 April 2018].

[20] D. Erickson, The beacon openflow controller, in: Proc. of the second ACM SIGCOMM workshop on Hot topics in software defined networking, ACM, 2013, pp. 13-18.

[21] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software defined networks, in: Proc. of the 9th SIGCOMM Workshop on Hot Topics in Networks, ACM, 2010, p. 19.

[22] Open vSwitch, Available from: http://www.openvswitch.org [28 April 2018].

[23] Scapy Library, Available from: http://www.secdev.org/projects/scapy/doc/usage.html [28 April 2018].

[24] M. Su~ne, L. Bergesio, H. Woesner, T. Rothe, A. Kopsel, D. Colle, B. Puype, D. Simeonidou, R. Nejabati, M. Channegowda, et al., Design and implementation of the OFELIA FP7 facility: The European OpenFlow testbed, Computer Networks 61 (2014) 132-150.

[25] S. Skiena, Dijkstras algorithm, Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley (1990) 225-227.

[26] R. W. Floyd, Algorithm 97: shortest path, Communications of the ACM 5 (6) (1962) 345.

[27] Using Bellman-Ford to Find a Shortest Path, Available from: http://csie.nqu.edu.tw /smallko/sdn/bellmanford.htm [28 April 2018].

[28] S. Hong, L. Xu, H. Wang, G. Gu, Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures, in: Proc. of Annual Network and Distributed System Security Symposium (NDSS'15), 2015.

[29] H. Krawczyk, R. Canetti, M. Bellare, HMAC: Keyed-hashing for message authentication, IETF RFC 2104.

[30] S. Turner, L. Chen, Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, IETF RFC 6151.

[31] M. Dhawan, R. Poddar, K. Mahajan, V. Mann, SPHINX: Detecting security attacks in software-defined networks, in: Proc. of Annual Network and Distributed System Security Symposium (NDSS'15), 2015.

[32] T. Alharbi, M. Portmann, F. Pakzad, The (In) Security of Topology Discovery in Software Defined Networks, in: Proc. of Local Computer Networks (LCN), IEEE 40th Conference on, 2015, pp. 502-505.

AUTHOR DETAILS

Talal Alharbi

Talal is a PhD candidate in Computer Science with a concentration on Network Security under the supervision of A/Prof. Marius Protmann at the University of Queensland (UQ), Brisbane, Australasia. He received his master's degree in Network Security and System Administration from Rochester Institute of Technology (RIT), Rochester, USA, and joined the School of Information Technology and Electrical Engineering (ITEE) in 2014. He obtained two advanced certificates from RIT focused on network planning and design and information assurance. Talal is an instructor at Majmaah University, Majmaah, KSA and prior to coming to UQ, he worked as a Vice Dean of IT Deanship for Systems and E-services. For his doctoral thesis, he is conducting a comprehensive investigation on security of Software Defined Network (SDN).

Marius Portmann

Marius Portmann is an Associated Professor at the School of Information Technology & Electrical Engineering at the University of Queensland, Brisbane, Australia. He has PhD in Electrical Engineering from the Swiss Federal Institute of Technology (ETH), Zürich, Switzerland. His research interests include Software Defined Networks, Pervasive Computing and Information Security.

Farzaneh Pakzad

Farzaneh Pakzad is a Researcher at University of Melbourne, Melbourne, Australia. She has PhD in Software Defined Wireless Mesh Networks from the University of Queensland, Brisbane, Australia. Farzaneh has an MSc degree in Computer Engineering from the Islamic Azad University, Arak, Iran. Her main research interests include wireless networks, Software Defined Networks and Network Security.