The (In)Security of Topology Discovery in Software Defined Networks

Talal Alharbi
School of ITEE
The University of Queensland
Brisbane, Australia
Email:t.alharbi@uq.edu.au

Marius Portmann
School of ITEE
The University of Queensland
Brisbane, Australia
Email: marius@ieee.org

Farzaneh Pakzad
School of ITEE
The University of Queensland
Brisbane, Australia
Email:farzaneh.pakzad@uq.net.au

***Abstract*—Topology Discovery is an essential service in Software Defined Networks (SDN). Most SDN controllers use a de-facto standard topology discovery mechanism based on OpenFlow to identify active links in the network. This paper discusses the security, or rather lack thereof, of the current SDN topology discovery mechanism, and its vulnerability to link spoofing attacks. The feasibility and impact of the attacks are verified and demonstrated via experiments. The paper presents and evaluates a countermeasure based on HMAC authentication.**

***Keywords*—SDN, Security, Topology Discovery**

I. Introduction

An essential service in Software Defined Networking (SDN) [1] is *topology discovery*, which underpins higher level applications and services such as routing and forwarding. While there is no official standard for a SDN topology discovery mechanism, there is a de-facto standard, which is sometimes informally referred to as Open Flow Discovery Protocol (OFDP) [2], [3]. All major OpenFlow-based 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 [4]. The problem with OFDP is that it is insecure, as is demonstrated in this paper. We show how an attacker can poison the topology view of the SDN controller and create spoofed links by crafting special control packets and injecting them into the network via one or more compromised hosts. We further demonstrate and evaluate the impact of the link spoofing attack on higher layer services, using shortest path routing as an example. A key contribution of the paper is the discussion and evaluation of a simple and effective countermeasure.

II. Background - OpenFlow and OFDP

OpenFlow [5] is the predominant *southbound interface* protocol for SDN. It is a wire protocol that allows the SDN controller to configure switches, i.e. via the installation of packet forwarding rules. 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 OpenFlow standard is version 1.5 [5]. However, the following discussion of the topology discovery method and its security is version agnostic and relevant to all versions of OpenFlow.

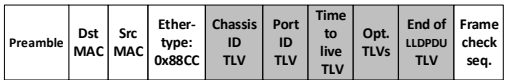

Fig. 1. LLDP Frame Structure

As part of the initial OpenFlow protocol handshake between the switches and the controller, the controller learns about the existence of switches and their key properties, i.e. their ports, MAC addresses, etc. As mentioned above, OpenFlow 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. OpenFlow 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 OpenFlow switch is forwarding a packet on a particular switch port, which can be either be physical ports, or can also be 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 a data packet to the controller, an SDN switch encapsulates the packet in an OpenFlow *Packet-In* message. OpenFlow 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 OpenFlow *Packet-In* and *Packet-Out* messages are essential for the topology discovery mechanism discussed in the following.

Topology discovery is an essential service in SDN and it underpins many higher layer services. When we refer to topology discovery, we actually mean *link discovery*, since the controller learns about the existence of network nodes (switches) during the OpenFlow handshake.

As mentioned above, *OFDP* (OpenFlow Discovery Protocol) [2], [3] is the de-facto standard for topology discovery in SDN, and is implemented by most SDN controller platforms. OFDP uses the frame format defined in the Link Layer Discovery Protocol (LLDP) (shown in Figure 1) [6], designed for link and neighbour discovery in Ethernet networks. However, with the exception of the frame format, OFDP has not much

978-1-4673-6770-7/15/$31.00 ©2015 IEEE

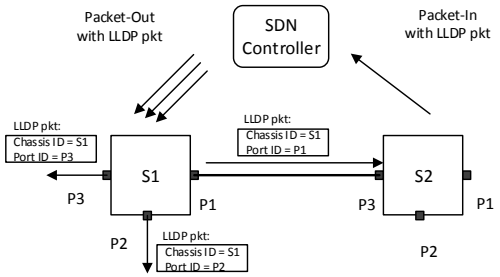

Fig. 2. Basic OFDP Example Scenario

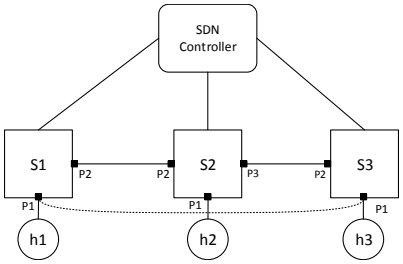

Fig. 3. Basic Attack Scenario

in common with LLDP. The LLDP payload is encapsulated in an Ethernet frame with the *EtherType* field set to 0x88CC. The frame contains an LLDP Data Unit (LLDPDU) (shaded in grey in Figure 1), 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 LLDPDU* TLV.

In SDN, link discovery is initiated by the controller. A basic example scenario is shown in Figure 2. The SDN controller creates a dedicated LLDP packet for each port on each switch. All these LLDP packets have their *Chassis ID* and *Port ID* TLVs initialised accordingly. The controller then uses a separate OpenFlow *Packet-Out* message to send each of the LLDP packets to the switch, *S1* in this case. Every *Packet-Out* message also includes an *action*, which instructs the switch to forward the packet via the corresponding port.

Switches are pre-configured with a rule which states that any received LLDP packets are to be sent to the controller via an OpenFlow *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 Figure 2. Switch *S2* sends the LLDP packet to the controller in a *Packet-In* message, which also contains additional meta data, such as the ingress port where the packet was received *(P3)*, as well as the *Chassis ID* of the switch sending the *Packet-In* message, which is *S2* in this case. 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. [1]

III. OFDP Link Spoofing - Basic Vulnerability

The basic security problem with 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. As a result, it is relatively easy for an attacker to inject fabricated LLDP control messages into the network, thereby corrupting the topology information of the controller. We illustrate this via the example shown in Figure 3. Here, we assume that host *h1* has been compromised by an attacker, who aims to create a fake link between switches *S1* and *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 1, 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 forwards it to the controller in a *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*, i.e. *(S1, P1)* in our scenario.

3) The controller receives the *Packet-In* message and identifies the source of the LLDP packet from the TLVs in the payload as *(S3, P1)*. The information about the other end of the link is taken from the meta-data of the *Packet-In* message, and is identified as *(S1, P1)*. Hence, the controller concludes (wrongly) that there exists a link between *(S3, P1)* and *(S1, P1)*.

In order to validate the feasibility of the link spoofing attack experimentally, we used Mininet [7], 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 vSwitch [8], a software OpenFlow switch which is supported in Mininet. We used the POX controller platform and its implementation of OFDP, i.e. the *openflow.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 [9]. The experiments were run on a 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 4 shows the debug output of the POX controller. 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 5 lines are from the POX *openflow.discovery* component which implements OFDP. Each of these lines indicates the detection of a unidirectional link, caused by the reception of a corresponding LLDP packet at the controller. The first 4 of the 5 lines show the detection of the bidirectional links between *(S1, P2)* and *(S2, P2)*, as well as between *(S3, P1)* and *(S1, P1)*. The key part is the last line, which appears after we run the attack by injecting the fabricated

[1] We have shown in [2] how the efficiency of this can be greatly improved.

```
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
```

Fig. 4. POX Debug Information

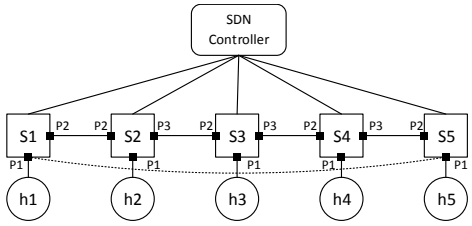

Fig. 5. Linear Topology for Routing Experiment

LLDP packet from host *h1* 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 was successful.

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 meta data 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 links 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* to *S3* in our example, he/she needs to control both *h1* and *h3*, as we will discuss later. We also performed the above attack using the Ryu SDN controller, with identical results.

IV. Impact on Routing

Routing is a key network application that relies on the controller having an up to date and accurate topology view. Here, we demonstrate and discuss the impact of the link spoofing attack on routing. We consider the simple linear topology shwon in Figure 5. 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 5. As described earlier, 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.

Prior to launching the attack we verified that there is connectivity among all host pairs using *ping*. After launching the attack from host *h1*, which injected the fabricated LLDP packet, we verified that the topology discovery service had indeed detected a link from *(S5, P1)* to *(S1, P1)*. Since the routing POX component *l2_multi.py* only considers bidirectional links, the above attack has no immediate impact on connectivity. In this case, for the attacker to be able to disrupt network connectivity, he/she needs to spoof a bidirectional link. In order to achieve this, an attacker needs to have control over two hosts, since only one end point of the spoofed link, i.e. the source, can be chosen by the attacking host.

In this new attack scenario, we assume the attacker controls hosts *h1* and *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, *h5* injects an LLDP packet with the source set to *(S1, P1)*, creating the link in the reverse direction. We ran the pairwise *ping* test again, and now we observed that the attack on connectivity had been successful. We see that all host pairs which are connected via a shortest path that includes the spoofed link *(S1-S5)*, are disconnected. In this basic scenario, connectivity of almost 30% of all the paths is disrupted by the creation of a single spoofed link. We have verified this for a range of other topologies, but are unable to present the detailed results due to lack of space.

V. Countermeasure

As mentioned above, a key problem of OFDP is the lack of any authentication of LLDP packets. We propose to address this by adding a cryptographic Message Authentication Code (MAC) to each LLDP packet, providing both authentication and packet integrity. We have implemented this mechanism in POX using HMAC, a keyed-hash based message authentication code [10]. The MAC is computed as follows:

$$HMAC(K, m) = h((K \oplus opad)|h((K \oplus ipad)|m))$$

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 '$\oplus$' denotes the XOR operation. $opad$ and $ipad$ are constant padding values [10].

It is important to note that the basic HMAC is vulnerable to replay attacks, which we can demonstrate for the scenario shown in Figure 3. 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 host *h3* 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 identifier (or *nonce*) to ensure that each HMAC value is unique. This message identifier needs to be sent as cleartext to the receiver as part of the message, causing additional overhead. We therefore use an alternative approach. We replace the static secret key K with a dynamic value $K_{i,j}$, 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

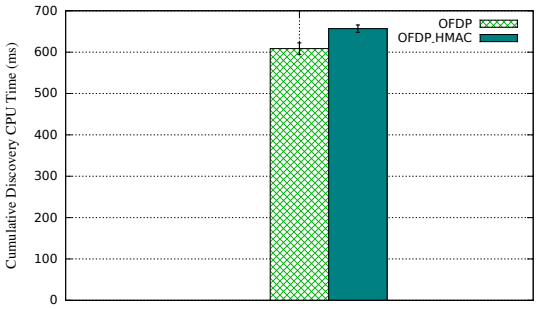

Fig. 6. Computational Overhead of HMAC in OFDP

value of the random numbers $K_{i,j}$. This is virtually impossible if we generate $K_{i,j}$ with sufficient entropy. In addition, any wrong guess by an attacker is 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 via the controller keeping track of the key used for each 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 [10], since HMAC does not rely on the collision resistance property [11].

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.

We have evaluated the computational overhead on the controller caused by this mechanism. For this, we used a 21 node tree topology in Mininet, with fan-out 4 and depth 2, and ran both the original POX OFDP mechanism and OFDP_HMAC. Figure 6 shows the total cumulative controller CPU time over a period of 300 seconds. A topology discovery round was initiated every 5 seconds, which is the default value in POX. The experiment was run 20 times and the figure shows the mean and the 95% confidence interval. 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 for the increased level of security.

VI. 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 topology discovery. The paper [12] discusses a range of attacks against SDN, and proposes *SPHINX*, a generic SDN attack alert system, which compares network behaviour with predefined or learned 'normal' behaviour, 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, nor does it explore the impact of the attack on network connectivity. In [13], 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 and evaluate the impact of the link spoofing attack on routing and hence network connectivity, as we have done in our paper. The authors of [13] also discuss potential countermeasures against the link spoofing attack, and also suggest a 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.

VII. Conclusions

Topology discovery is an essential service in SDN. 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 and demonstrated the vulnerability of OFDP to link spoofing attacks, which only require an attacker to have control over one or more hosts (physical or virtual) in the network. We have demonstrated the feasibility of the attack and its impact on network connectivity, using the example of routing. Finally, we discussed and evaluated a potential countermeasure based on authentication of LLDP control messages using HMAC.

Acknowledgment

This work is supported by Majmaah University through the Saudi Arabian Culture Mission in Australia.

References

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, p. 20, 2013.

[2] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska, "Efficient topology discovery in software defined networks," in *IEEE ICSPCS, 2015*.

[3] *GENI Wiki*. [Online]. Available: http://groups.geni.net/geni/wiki/OpenFlowDiscoveryProtocol

[4] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, Jul. 2008.

[5] *Open Flow Standard*. [Online]. Available: https://www.opennetworking.org/sdn-resources/onf-specifications/openflow

[6] "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)*, pp. 1–204, Sept 2009.

[7] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*.

[8] *Open vSwitch*. [Online]. Available: http://openvswitch.org

[9] *Scapy Library*. [Online]. Available: http://www.secdev.org/projects/scapy/doc/usage.html

[10] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," *IETF RFC 2104*, pp. 1–11. , February 1997.

[11] S. Turner and L. Chen, "Updated security considerations for the md5 message-digest and the hmac-md5 algorithms," *IETF RFC 6151*, 2011.

[12] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks," in *NDSS'15*, February 2015.

[13] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *NDSS'15*, February 2015.