Files

1786eaa

Go to file

.github
CodeQL_Queries
Conferences
Meetup
SecurityExploits
Android
Ansible
Apache
CImg
Chrome
Facebook
Microsoft
SANE
Ubuntu
apple
freedesktop
libssh/pubkey-auth-bypass-CVE...
attacker
server
README.md
libssh2
polkit
rsyslog
strongSwan
vivo-project
README.md
docs
mastodon
.gitignore
.gitmodules
CODE_OF_CONDUCT.md
CONTRIBUTING.md
LICENSE.md
README.md

securitylab / SecurityExploits / libssh / pubkey-auth-bypass-CVE-2023-2283 /

kevinbackhouse PoC for libssh CVE-2023-2283 79b4e6f · last year History

Name	Last commit message	Last commit date
..		
attacker	PoC for libssh CVE-2023-2283	last year
server	PoC for libssh CVE-2023-2283	last year
README.md	PoC for libssh CVE-2023-2283	last year

README.md

Public key authentication bypass in libssh (CVE-2023-2283)

CVE-2023-2283 is an authentication bypass vulnerability in libssh, which, under certain conditions, may enable a remote attacker to gain unauthorized access to another user's account via ssh login.

This demo uses docker to simulate two computers, named "libssh-server" and "libssh-attacker". On libssh-server, we run `ssh_server_pthread`, which is a simple ssh server application that is included as an example with the libssh source code. The server is configured to allow public key authentication with an ED25519 key, but the attacker does not know the private key. The attacker instead authenticates by triggering the vulnerability.

The vulnerability is triggered when `ssh_server_pthread` hits an out-of-memory condition at precisely the right moment. If libssh is running on a 64-bit server with plenty of RAM then it is very unlikely that an attacker will be able to generate enough memory pressure to cause an out-of-memory error, which means that the vulnerability is unlikely to be exploitable. The goal of this demo is, instead, to show that the vulnerability is exploitable if libssh is running in a memory-constrained environment such as a memory-constrained container, which we believe is a realistic scenario for a real-life libssh deployment. The demo uses `ulimit` to set a 256MB memory limit on the ssh server.

Network setup

Create a docker network bridge, to simulate a network with two separate computers.

```
docker network create -d bridge --subnet 172.18.0.0/16 libssh-demo-netwo
```

Server setup

Build the docker image:

```
docker build server -t libssh-server --build-arg UID=`id -u`
```

Start the container:

```
docker run --rm --network libssh-demo-network --ip=172.18.0.10 -it libss
```

If you want to be able to debug the libssh server, then you need to start the container with some extra command line arguments:

```
docker run --rm --network libssh-demo-network --ip=172.18.0.10 --cap-add
```

Inside the container, run these commands to create ssh keys for the server:

```
mkdir ~/testkeys
ssh-keygen -P "" -t ecdsa -f ~/testkeys/id_ecdsa
ssh-keygen -P "" -t rsa -f ~/testkeys/id_rsa
```

Start the server:

```
ulimit -v 262144  # 256MB
~/libssh/build/examples/ssh_server_pthread -p 2022 -r ~/testkeys/id_rsa
```

Note: ssh servers normally listen on port 22, but root privileges are required to listen on 22, so this demo uses port 2022 instead. Use `sudo` if you want to change the port number to 22. The `sudo` password in this docker container is "x".

Attacker setup

Build the docker image:

```
docker build attacker -t libssh-attacker --build-arg UID=`id -u`
```

Start the container:

```
docker run --rm --network libssh-demo-network --ip=172.18.0.11 -it libss
```

If you want to be able to debug the client, then you need to start the container with some extra command line arguments:

```
docker run --rm --network libssh-demo-network --ip=172.18.0.11 --cap-add
```

The attacker uses a modified version of libssh. The modifications are in the file named `diff.txt` and are applied during the `docker build` step.

Run the malicious client like this:

```
~/libssh/build/examples/ssh-client -p 2022 victim@172.18.0.10 ~/id_ed255
```

The vulnerability is triggered when the ssh server has an out-of-memory error at the exact right moment, which means that the PoC is unreliable. It runs in a loop until it's successful, which can often take several minutes. You may also need to run several instance of the PoC simultaneously to generate enough memory pressure on the server. I suggest using `tmux` to open three terminals and start 3 instances of the PoC. When one of the PoCs succeeds, it creates a file named "success.txt", which notifies the other instances that they should stop.

Note: the PoC sometimes accidentally triggers a SIGSEGV in the server due to an unrelated null-pointer dereference bug. If this happens, you will need to restart the `ssh_server_pthread` process.