Write-up Secure Green Server (Fault injection)

France Cybersecurity Challenge (FCSC) 2022

erdnaxe

Contents

1 Secure Green Server - Fault injection challenge

495 points were awarded for the resolution of this challenge, only 3 people managed to solve it during the competition[1].

1.1 Official description

> La société MegaSecure mets à disposition un serveur sécurisé permettant de lancer des opérations tout en maîtrisant la consommation énergétique de ce dernier.
>
> Le serveur permet d'exécuter des commandes de manière sécurisée. En effet, il repose sur un composant sécurisé afin de vérifier la signature de toute commande reçue avant de l'exécuter.
>
> Le code Python équivalent à la signature est le suivant :
>
> ```
> def sign(self, m):
>     return pow(int(sha256(m), 16), self.d, self.N)
> ```
>
> et la vérification est faite de la manière suivante :
>
> ```
> def verif(self, m, s):
>     return int(sha256(m), 16) == pow(int(s),self.e,self.N)
> ```
>
> Néanmoins, étant en phase de test seules deux commandes (`ls -la flag.txt` et `cat flag.txt`) sont disponibles. Par ailleurs, il a été remarqué que le serveur présente des comportements étranges dans certaines configurations.
>
> `nc challenges.france-cybersecurity-challenge.fr 2253`

1.2 Exploration

Let's connect to the remote server using Netcat, we are greeted with the following prompt:

```
Welcome on SecureGreenServer, you can execute your commands in a
   secure environment while taking care of the environment by
   tuning the power consumption of the server !
Commands are:
|-> sV <V>: Set voltage to V (in V)
    For instance: sV 2.3

|-> sF <F>: Set frequency to F (in Hz)
    For instance: sF 100e6

```

[1]These statistics are subject to change as these write-ups were sent before the end of the competition.

```
|-> x <c> <s>: Verify the signature s of the command c then execute
    it
    For instance: x 'ls -la flag.txt'
       11965733308966373769348387876071810551555082319314O4<snip>

|-> i: Show server information
|-> u: Show this usage information
|-> q: Quit
>>>
```

Let's show the server information:

```
>>> i
{'voltage': 1.3, 'frequency': 50000000.0, 'max_voltage': 5, '
   min_voltage': 1.3, 'max_frequency': 2000000000.0, 'min_frequency
   ': 50000000.0}
```

Let's run the example command:

```
>>> x 'ls -la flag.txt' 119657333089663737693483878760718l0<snip>
-r--------    1 guest    users          174 Apr 30 13:27 flag.txt
0
```

Let's rerun the command with a wrong signature:

```
>>> x 'ls -la flag.txt' 4242
Signature verification failed with following parameters:
e = 65537
N = 1630100484731635563196761980366998334240032960964706111O1<snip>
```

1.3 Proposed solution

The goal is to sign and run `cat flag.txt` on the remote server.

We notice that at each connection, the signature of the `ls -la flag.txt` command changes. This means that we have to guess the secret key and sign the command in the same session.

In the following sections we are going to explore how to crash the server, then how to find parameters in which the computation is faulted without crashing. Finally we factorize the faulted RSA N modulo to recover the secret and sign the `cat flag.txt` command.

1.3.1 Crashing the server

We have control over server frequency f and voltage V such as:

- $1.3 \leq V \leq 5$ (in volt),

- $50 \leq f \leq 2000$ (in MHz).

Higher processors frequencies will cause more transistors switching per second, which means more current consumption due to switching losses. It also reduces the time window in which processor signals propagate. **If the voltage is too low, then the processor may become unstable.**

Let's put the maximum frequency and the minimum voltage:

```
>>> sF 2000000000
>>> sV 1.3
>>> x 'ls -la flag.txt' 1
Command not done, the system had to reboot for some reason
```

Oops, the system detected the fault and rebooted. This raises the following question: *Are there sets of parameters causing faults without rebooting the server?*

1.3.2 Faulting the server

Let's run the `ls -la flag.txt` given in example many times at $f = 2$ GHz and with V varying from 1.3 V to 5 V. We write the following Python script:

```python
import socket
import numpy as np

def recv_until_prompt(s: socket.socket) -> bytes:
    """Receive data until prompt"""
    rx = s.recv(1024)
    while not rx.endswith(b">>> "):
        rx += s.recv(1024)
    return rx

# Connect to remote server
s = socket.socket(socket.AF_INET)
s.connect(("challenges.france-cybersecurity-challenge.fr", 2253))

# Get session example
rx = recv_until_prompt(s)
example_command = rx.split(b"\n")[9][18:]
print("example command:", example_command)

# Fix the frequency
s.send(f"sF 2000000000\n".encode())
rx = recv_until_prompt(s)

Vs = np.linspace(1.3, 5, 64)  # Scan from 1.3V to 5V
n_reboot, n_good = 0, 0
```

```
for i, voltage in enumerate(Vs):
    s.send(f"sV {voltage}\n".encode())
    rx = recv_until_prompt(s)

    s.send(example_command + b"\n")
    rx = recv_until_prompt(s)

    if b"Command not done, the system had to reboot for some reason
       " in rx:
        n_reboot += 1
    elif b"flag.txt" in rx:
        n_good += 1
    else:
        print("fault!", rx, voltage)

print(f"{n_reboot=}, {n_good=}")
```

Running this Python script, we get:

```
fault! b'Signature verification failed with following parameters:
e = 65537
N = 171664987635353322221796683199868132096341862323194726615<snip>
2.650793650793651
n_reboot=23, n_good=40
```

We run this script another time with V fixed at 2.65 V. **All responses are now giving faulted N values.**

1.3.3 RSA moduli factorization

Let's go back to the given `sign` and `verif` functions:

```
def sign(self, m):
    return pow(int(sha256(m), 16), self.d, self.N)

def verif(self, m, s):
    return int(sha256(m), 16) == pow(int(s), self.e, self.N)
```

We recognize the RSA signature and verification scheme using SHA256 hash function. m is signed using the secret d. The moduli N, and e are public.

d and N are derived from two large prime numbers p and q. RSA security relies on the fact that factorizing $N = p \times q$ is hard. **When the moduli N is faulted, it might no longer be hard to factorize.**

"Using Fault Injection to weaken RSA public key verification" by Ivo van der Elzen explains in section 4 how to attack faulted N moduli. We are going to implement this attack.

Be careful when writing the key generation. If $N = \prod_i p_i^{k_i}$ *has more than 2 prime factors, then* $\phi = \prod_i p_i^{k_i-1}(p_i - 1)$*, with* k_i *the power of the prime factor. It took me quite some time to realize this.*

```python
import socket
import numpy as np
from hashlib import sha256
from sympy.ntheory import factorint

def recv_until_prompt(s: socket.socket) -> bytes:
    """Receive data until prompt"""
    rx = s.recv(1024)
    while not rx.endswith(b">>> "):
        rx += s.recv(1024)
    return rx

def sign(m, d, N):
    return pow(int(sha256(m).hexdigest(), 16), d, N)

# Connect to remote server
s = socket.socket(socket.AF_INET)
s.connect(("challenges.france-cybersecurity-challenge.fr", 2253))

# Fix voltage and frequency
s.send(f"sF 2000000000\n".encode())
rx = recv_until_prompt(s)
s.send(f"sV 2.65\n".encode())
rx = recv_until_prompt(s)

while True:
    # Request faulted N
    s.send(b"x 'ls -la flag.txt' 1234\n")
    rx = recv_until_prompt(s)
    if b"N =" not in rx: continue

    # Find divisors
    # Could be wrapped inside a timeout
    try:
        e = int(rx.split(b"\n")[1][4:])
        N_faulted = int(rx.split(b"\n")[2][4:])
        print(f"{N_faulted=}")
        print("Factorization... please CTRL-C after 1s")
        ps = factorint(N_faulted)
    except KeyboardInterrupt:
        continue

    # RSA generation
    phi = 1
    for p, count in ps.items():
        phi *= (p - 1) * p**(count-1)
    d = pow(e, -1, phi)

    # Send signed message and get flag!
    signed = sign(b'cat flag.txt', d, N_faulted)
    break

# Hope same fault happens again
while True:
    s.send(f"x 'cat flag.txt' {signed}\n".encode())
    rx = recv_until_prompt(s)
    print(rx)
    if b"FCSC" in rx: break
```

After some CTRL-C and a bit of wait, we get the flag and a reference to the CLKSCREW paper:

```
Congrats! You did this: https://www.usenix.org/system/files/
   conference/usenixsecurity17/sec17-tang.pdf
FCSC{e0e31e5f46a1488159b16e2f6b58fc1135c0a77ef0a423c5fa0d1ee74836b5
ef}
0
```

During this challenge, I learned:

- How to exploit RSA using moduli factorization, and discovering that ϕ has a more generic formula than just $(p-1)(q-1)$,
- How to compute inverse modulo using Python `pow`,
- To remember to look for repetitions (this is also a good advice for `X-Factor 2/2`).

Figure 1: Johnny from Airplane! doing power management