WRITEUP

CSCG 2024

CHALLANGE: CHOOSE YOUR OWN ADVENTURE

From: werter (Discord name)

Date: 31.03.2024

CONTENT

THE CHALLENGE

Name: Chose Your Own Adventure

Authors: THOMAS

Categories: CRYPTO

Difficulty: Easy

Description: I just learned how to use the shell and wrote my first choose-your-own-adventure game. Can you help our hero find the flag?

Given: natcat ssl connection and a zip directory

Summarize: The challenge is a text-based adventure game. In the game we can found a decrypted flag. The used random algorithm for the key is insecure so it is possible to calculate the key and decrypt the massage.

1. THE SERVICE

The challenge gives access over ncat ssl to a server. A script is running on the server. It contains the game. In the game, several coin flips must be done. After the way splits and another coin flip is done, depending on which way it was taken, the first part of the flag decrypted or the second part in cleartext are given.

2. THE CODE

The zip directory contains two files.

The Tree view:

```
.
├── Dockerfile
└── run.sh
```

Dockerfile

The Dockerfile starts the server. It sets all configuration and installs important services. It can be used for local testing.

run.sh

This shell script is made in zsh. It contains the challenge. It contains a fake flag. The first part goes decrypted. Most important part from the script is how the flag gets decrypted.

```
for ((i=1; i<=32; i++))
        do
            PASSKEY=$(echo $PASSKEY$RANDOM | md5sum)
        done
        export PASSKEY=$PASSKEY
        echo "CSCG{FLAG_PART_1_" | openssl enc -aes-256-cbc -e -pass
env:PASSKEY 2>/dev/null | hexdump -C
```

On start the PASSKEY contains the result of the six coin flips. It gets 32 times hashed together with a random number between 0 and 32767. The finished PASSKEY is used as a key for an openssl decryption.

3. SOLUTION

To solve the challenge, it is important to found both parts of the flag. The second part is going easy. Just answer: “P” on the question “?Choose P for the Portal of Fate or O for the Oracle's Den: “

After that is only a coin flip to win.

Now the second part of the flag is resulted.

The first part of the flag is much harder. The used -aes-256-cbc algorithms are very secure and not easy to crack. So, I need to decrypt the openssl encryption on the normal way.

```
echo "CSCG{FLAG_PART_1_" | openssl enc -aes-256-cbc -e -pass env:PASSKEY
2>/dev/null | hexdump -C
```

I found out how to decrypt a massage with openssl fast. The only problem was the hexdump -C. It took me a lot of time to get the 100% original massage back without some extra stuff.

But I found a way to extract only the hexdata and to reverse this.

```
echo "$massage" | awk '{print $2$3$4$5$6$7$8$9$10$11$12$13$14$15$16$17}' |
tr -d '\n' | xxd -r -p | openssl enc -aes-256-cbc -d -pass env:PASSKEY
2>/dev/null
```

The only way to decrypt is with the same key the encryption uses. Fortunately, the challenge gives the code as the key is generated.

```
        for ((i=1; i<=32; i++))
        do
            PASSKEY=$(echo $PASSKEY$RANDOM | md5sum)
        done
```

The started PASSKEY can be easy replicated. It is the result of the six coin flips and this can look like:

HHTHTH

The new passkey is the oldpasskey plus a random number together md5 hashed.

The $RANDOM function generates a pseudo random number between 0 and 32767. Pseudo means it is not secure. The only thing left is understanding how the random generator works.

After I longer searched for bash, I noticed that bash is not used. The script use zsh. After a short search over zsh I found out that zsh implementation for the random algorithm is not good.

ZSH DOKU:

> A pseudo-random integer from 0 to 32767, newly generated each time this parameter is referenced. The random number generator can be seeded by assigning a numeric value to RANDOM.
>
> The values of RANDOM form an intentionally-repeatable pseudo-random sequence; subshells that reference RANDOM will result in identical pseudo-random values unless the value of RANDOM is referenced or seeded in the parent shell in between subshell invocations.

I tested the identical pseudo random values in subshells.

```
#!/usr/bin/env zsh

PASSKEY=TTTHHH
for ((i=1; i<=32; i++))
do
    random=$(echo $RANDOM)
    PASSKEY=$(echo $PASSKEY$RANDOM | md5sum)
    echo "$random"
    echo "$PASSKEY"
done
```

21027

46431c0b441f8bd011ae71fdd6bbaf7d -

21027

1fed0bfbb0c41711fcd00dc4e9fdb857 -

21027

1b1fd6f0d18a1ba0649813cd7d10de3d -

21027

[...]

So, this means that only 32768 possible keys exist. This can be easily brute forced.

I was shure there is a way to calculate the random number that is used. To found out how $RANDOM works I cloned the github repo from zsh.

https://github.com/zsh-users/zsh

After short time I found the interesting code passages.

src/init.c

```
srand((unsigned int)(shtimer.tv_sec + shtimer.tv_usec)); /* seed $RANDOM */
```

src/parms.c

```
randomgetfn(UNUSED(Param pm))
{
   return rand() & 0x7fff;
}
```

These lines of code mean the C rand() function is used. It gets seeded with the time in sec and millsec.

It might be possible to brute force the time and get the same seed back.

With this seed it is possible to calculate all future random numbers.

Because this would be a little bit harder, I decided to brute force all possible keys direct.

So, I wrote a script and run it.

```
#!/bin/bash

Key=TTTTTH
massage='00000000  53 61 6c 74 65 64 5f 5f  7f b4 94 5d c8 01 20 5e
|Salted__...].. ^|
00000010  8c 85 5f 6c 0c 56 78 59  6d de 79 8b ac 66 31 7e
|.._l.VxYm.y..f1~|
00000020  a0 db 49 93 24 0c b0 5f  57 ac ce 4e ce c7 31 77
|..I.$.._W..N..1w|
00000030  b8 d6 e5 22 06 0f 91 f8  9a 84 98 e6 9e 6e 7e 87
|..."........n~.|
00000040'

crack () {
    export PASSKEY=$1

    output=$(echo "$massage" | awk '{print
$2$3$4$5$6$7$8$9$10$11$12$13$14$15$16$17}' | tr -d '\n' | xxd -r -p  |
openssl enc -aes-256-cbc -d -pass env:PASSKEY 2>/dev/null)
    if [[ $output == *"CSCG{"* ]]; then
        echo "$output"
    fi
}

for round in {0..32767}
do
    METAKEY=$Key
    for ((i=1; i<=32; i++))
    do
        METAKEY=$(echo "$METAKEY$round" | md5sum)
    done
    crack "$METAKEY"
done
```

./cracl.sh 2>/dev/null

This gave me a short brake in which I ate my lunch. After the brake I simply got the first part of the flag.

Because I was interested in this i decided to try the other way and create a python script to calc the seed.

```python
import datetime
import ctypes

random = ctypes.CDLL("libc.so.6")

firstrandomnumber=8109
# coin H=1, T=0
c1=1
c2=1
c3=1
c4=0
c5=0
c6=1

starttime = datetime.datetime(2024, 3, 29).timestamp()

seed = int(starttime)
while True:
    #print(seed)
    random.srand(seed)
    if random.rand() & 0x7FFF == firstrandomnumber:
        if random.rand() & 1 == c1:
            if random.rand() & 1 == c2:
                if random.rand() & 1 == c3:
                    if random.rand() & 1 == c4:
                        if random.rand() & 1 == c5:
                            if random.rand() & 1 == c6:
                                randomnuber = random.rand()& 0x7FFF
                                print(randomnuber)
                                exit()
    seed+=1
```

The response getting used in this bash script is based on my first script.

```
#!/bin/bash

RANDOMNUMBER=11697
Key=HHHTTH
massage='00000000  53 61 6c 74 65 64 5f 5f  d0 7e d0 c0 89 b6 00 28
|Salted__.~.....(|
00000010  d2 e7 92 a8 ba fb 99 09  c2 8a 5c c6 84 af 4e 2a
|..........\...N*|
00000020  0f 1a 68 53 36 ba 9d e9  16 1e 20 93 28 02 54 a8  |..hS6..... 
.(.T.|
00000030  81 62 db 26 21 72 2f 3a  b3 95 cf 99 ed c1 56 e4
|.b.&!r/:......V.|
00000040'

crack () {
    export PASSKEY=$1
    output=$(echo "$massage" | awk '{print
$2$3$4$5$6$7$8$9$10$11$12$13$14$15$16$17}' | tr -d '\n' | xxd -r -p  |
openssl enc -aes-256-cbc -d -pass env:PASSKEY 2>/dev/null)
    if [[ $output == *"CSCG{"* ]]; then
        echo "$output"
    fi
}

METAKEY=$Key
for ((i=1; i<=32; i++))
do
    METAKEY=$(echo "$METAKEY$RANDOMNUMBER" | md5sum)
done
crack "$METAKEY"
```

This solution gives the flag much faster as my first solution.

4. FIX VULNERABILITY.

In this challenge are two vulnerabilities.

1. **The used random algorithm.**
   $RANDIM is based on the C rand() function and it is a pseudo random generator. This means that the randomnubers can be calculated. If you know the seed, you know every possible random number.
   As far as coin flips are concerned it is not so important to use a secure random algorithm. In other cases, like the key generation a secure random algorithm is necessary.
   On linux on way is to use the build in /dev/random or /dev/urandom. This gives cryptographic secure random numbers.
   Another way is to use real random numbers. To get this is hard. It can be get by a API or create self. This random numbers need a source which can be for example a lava lamp or a Quante computer.

2. **The key for the decryption**
   The used openssl decryption is secure. The Achillis heel is the key. Not only the insecure random algorithm is used. With the behavior of zsh subshells there are only 32767 possible keys. For the hashing, the insecure hash md5 is used. Md5 is for multiple reasons not a recommendation anymore. The use of a secure algorithm like SHA 256 makes it more secure but it is still not good.
   It is recommended to use a standard for secure key generation. There are multiple Bibliotheca for this.

   Another way is to use other decrypt standard. Like RSA with a private and public key. Or the one-time pad.