Burger

Introduction

Category: Cryptography
Solves: 25
Points: 397

This was a really fun challenge! As I am unfamiliar with cryptography, this challenge took me some time. However, it was worth it as I learned a lot from this challenge :)

The goal of this task is to obtain the flag from a hashing machine, given the source code with the hashing algorithm. The machine accepts any valid hexadecimal input and appends it to the flag, before hashing it.

After the challenge, I've learnt that this challenge is attempting to recreate the AES ECB vulnerability.

Solving the challenge

At first glance

The challenge prompt gives us a Netcat command which runs a server, along with a link to download its source code.

The server does as the source code says: Appends the input, converted from hexadecimal to bytes, to the flag and returns the hash of it. The server is essential to us as it contains the flag which we are trying to obtain.

By analysing the source code further, we find out a few more things:

- The flag is a 56-character long string
- The `hash()` function takes our input, converted to bytes, as its only argument. It returns the hash of (input + flag), where our input is prepended to the flag.
- The hashing algorithm in `hash()` uses the SHA512 hashing algorithm and works in blocks of 16 bytes. In other words, it hashes every 16 characters of the string and chains it together to form the final hash.
    - Only the first 32 characters of the hexadecimal representation of the calculated hashes are kept per block.
    - This means that every 32 characters/16 hexadecimal numbers in the hash returned from the server corresponds to 16 characters in the input+flag string.
    - The hash blocks are also in order, meaning that the first 16 hexadecimal numbers in the hash corresponds to the first 16 characters in the input+flag string.

Given all these data, we can now attempt to solve for the flag.

Approach

First of all, we must understand that the hash we obtain is non-reversible, unlike text secured through encryption. This means that there is no function that can systematically decode hashes that we input.

In other words, we can only crack the hash by comparing hashes of known inputs passed through the same hashing algorithm with our target hash. In this case, we can pass known inputs through the SHA512 algorithm and compare it with the hash(es) from the server.

Since the server accepts a blank input to give the unmodified flag hash, we can attempt to solve for the flag by brute-forcing the hash. Since we know that the flag is 56 characters long, and that it is a byte string, we can calculate the maximum number of attempts we have to make.

A byte string can take values from 0-255. Thus, there are 256 permutations for one byte. Narrowing down the possibilities to only the basic ASCII characters, we are left with about 95 possibilities per byte. Since we know the flag starts with `greyhats{` and ends with `}`, we only have to brute-force 46 characters. That is 95^{46} permutations, which is only about 9446824413773783721646626909963562851135776704278099454808312884779297746717929840087890625 permutations. Seems doable, right? ...right?

Just kidding! The above number was about $6.01 * 10^{91}$, which is larger than the estimated number of particles in the entire universe ($\approx 3.28 * x10^{80}$). I was stuck with the thought process above for quite a while, thinking of ways to reduce the number of characters I needed to guess, and maybe get the flag in time to submit it before the competition ends.

Anyway, we can be certain that brute-forcing is not the intended solution to solve this challenge. I looked at the code for a very long time before I noticed that it was weird for the algorithm to process characters in blocks instead of just hashing it in one go.

The SHA512 algorithm is able to accept inputs up to 2^{128} bits long, so it isn't an issue of maximum input size. Python3 itself is also able to accept arbitrarily large numbers, constrained by the program's allocated memory. Thus, this processing in blocks of 16 must be related to the solution.

Then, I realised: splitting the text into chunks of 16 means that I only have to crack a few SHA512 hashes with 16-character long inputs. How can I further reduce the number of unknown characters?

Here is where our input to the server comes in handy: it is prepended to the string, thus we can generate a hash where we know the first few characters of the input. By sending 6 known characters to the server, we can get a hash where we know 15/16 of the characters for the first chunk. This leaves us with just one character to guess!

We can extend this to any other chunk of the hash, as long as we know 15/16 characters of the chunk. Now, we can even try out all 256 characters for all 56 characters of the flag, which will only take 14336 tries. Let us proceed with the actual implementation of this method!

Implementation

We can start by sending known characters to the server to obtain the hashes we need. I used the character `a` as my known character, which has a hexadecimal value of 61 (represented as 0x61).

This can be obtained by running `hex(ord('a'))` in Python, which essentially converts `a` into its corresponding character code before converting it into its hexadecimal form.

We can then create a function to help us guess the last character of a block, given the first 15 known characters of the block before hashing, and the 32-character hash. The function I used in the challenge is given below.

```
import hashlib
def getChar(knownChars: str, hash: str) -> char:
 for i in range(256):
     testChar = chr(i)
     testHash = hashlib.sha512((knownChars + testChar).encode()).hexdigest()
[:32]
     if testHash == hash:
         return testChar
```

I iterated from 0-255 when guessing the character code as I was lazy to narrow down the range, but you can do so if you wish.

This function takes `knownChars`, the 15 known characters, as its first argument and `hash`, the 32-character hash as its second argument. It will return a character if it finds a match, else it returns nothing.

Since the hashes are constant for the same input, we can create a list of hashes that contain n*`a` prepended before the flag, where n ranges from 0 to 15. The reason why we stop at 15 is because each block is of size 16, thus we only need a maximum of (16 - 1) known characters before we can work our way backwards to guess the entire block.

The following code block is a python function to get a list named `hashes`, where `hashes[i]` is the hash of `('a' * i) + flag`.

```
import socket
def getHashes(maxChars: int) -> list:
    host = "challs1.nusgreyhats.org"
    port = 5210
    hashes = []
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    s.connect((host, port))
    s.recv(1024)    # discard welcome message
    for i in range(maxChars):
        s.send(('61' * i + '\n').encode())
        hashes.append(s.recv(1024).decode().split('\n')[1])
    s.close()
    return hashes
```

I was unable to create this function during the competition as I was unable to get the hash after sending an input. I took about an hour after the competition to try creating the function, and realised that it was because my input did not end with a `'\n'`, which signifies the return key being pressed. Can't believe the solution was so simple :(guess I learnt a very valuable lesson which will definitely help me the next time I'm trying to do something like that again!

Finally, we can start on solving the hash character-by-character. Once again, a python function can be used to help us. The following function makes use of the two functions shown above. It takes the length of the flag as its only input, and outputs the flag!

```python
def getFlag(flagLen: int) -> str:
    hashes = getHashes(16)
    flag = ''
    known15 = 'a'*15
    for i in range(flagLen):
        listIndex = 15 - (i % 16)
        hashStart = (i // 16) * 32
        hash32 = hashes[listIndex][hashStart:hashStart + 32]
        guessChar = getChar(known15, hash32)
        flag += guessChar
        known15 = known15[1:] + guessChar
    return flag
```

- `known15` stores the first 15 characters of the block that we will be looking at.
- `listIndex` is used to choose the element we need from the `hashes` list.
    - It represents the number of '`a`'s we have placed before the flag to substitute actual known characters.
    - Thus, it should be 15 when we are guessing the very first character of the block and 0 for the last.
- `hashStart` is the starting index from which we extract the 32 characters from the chosen hash.
    - First, we do integer division of `i` by 16 to get the n^{th} block we are working with.
        - When `i` is between 0 and 16, it is under the first block of index 0, thus `i` returns 0
    - Thereafter, we multiply the number by 32 as each block is 32 characters long.
- `hash32` is the 32-character long hash we will be using to guess in the current iteration.

I was only able to write a fraction of the above function during the competition as I was using Python in the command line, which made it very annoying to modify functions as I would have to retype the whole thing. Another lesson learnt for me: try to use IDEs while solving challenges to make life easier!

After running the `getFlag()` function, we obtain the flag:
greyhats{B3lanJa_m3_Burg3R_1f_y0u_3njoyed_7he_Ch@ll3n93}

(the script can be found under burger.py)