..		
README.md	add writeups by dhanesh	2 years ago
hwaes.py	31c3 challenges placeholders	2 years ago
solve.py	31C3 CTF 2014: make folder names lowercase for consistency	2 years ago

README.md

31C3 CTF 2014: hwaes

Category: crypto Points: 10 Solves: 53 Description:

> We implemented AES in hardware and saved a lot of memory. Feel free to use our online AES encryption service to secure your data.
>
> ```
> nc 188.40.18.66 2786
> ```

Write-up

Getting started

If you examine the script you see that we have several functions:

- setkey
- getkey
- encrypt
- flag

So we can give a key and encrypt a string with that key. But if we encrypt the flag, the code uses a random key to encrypt it but the program doesn't seem to change the key after it encrypted the flag. So maybe we can use `getkey` and decrypt the flag? But that doesn't work. After some testing with `setkey` , `encrypt` and `getkey` we notice that after an encryption our key changed. However it was still encrypted with the key we set and subsequent encryption are also encrypted with the key we set. So what key do we get if we use `getkey` ? The answer can be found by examining the encryption cipher very closely. The cipher makes use of several encryption rounds and each round uses a different key. Those keys are derived from the cipherkey using the Rijndael key schedule. So the key we get using `getkey` is probably the expanded key of the cipher key. This also explains the 'hint' in the challenge description. The normal method for decrypting is making all the expanded keys from the cipherkey and storing them in the RAM. But if you just store the (first) cipherkey and the last expanded key you can decrypt and encrypt on the fly thus saving you precious RAM. So now we just need to either get the original cipherkey by inverse expanding the key we get or we can also decrypt on the fly with the expanded key. I opted for the first choice since it requires a bit less work.

Inverse key expansion

This step requires some logic thinking but is mostly straightforward. You can find lots of information on the expanded key algorithm and you can write the code yourself or just find the code online but you don't really need this. You only need to inverse the expansion algorithm. However for completeness I added the expansion algorithm in python below.

```
# 4-byte temporary variable for key expansion
  word = exkey[-4:]
  # Each expansion cycle uses 'i' once for Rcon table lookup
  for i in xrange(1, 11):

    #### key schedule core:
    # left-rotate by 1 byte
    word = word[1:4] + word[0:1]
    # apply S-box to all bytes
    for j in xrange(4):
      word[j] = aes_sbox[word[j]]
    # apply the Rcon table to the leftmost byte
    word[0] = word[0] ^ aes_Rcon[i]
    #### end key schedule core

    for z in xrange(4):
      for j in xrange(4):
        # mix in bytes from the last subkey
        word[j] ^= exkey[-self.key_size + j]
      exkey.extend(word)
```

To inverse this I started with the last part and unmixed bytes from the previous subkey. This results in the following code:

```
for z in xrange(3):
  for j in xrange(4):
    # unmix the bytes from the last subkey
    word[j] ^= invexkey[j - (z+2)*4]
  temp[:0] = word
  word = invexkey[-(z+2)*4:-(z+1)*4]
```

Note that I only unmix `3` bytes and that I store everythin in a temporary array. The reason for this is because I start with the last byte and work my way to the first so I prepend every byte to the array `temp` and if I have the whole key I append it to the array `invexkey`. To get the first byte we need to unmix it with the last byte from the previous key which we just partially created. However we need to `scramble` that byte before we use it to unmix it with the first byte.

```
word = temp[-4:]
#### key schedule core:
# left-rotate by 1 byte
word = word[1:4] + word[0:1]
# apply S-box to all bytes
for j in xrange(4):
  word[j] = aes_sbox[word[j]]
# apply the Rcon table to the leftmost byte
word[0] = word[0] ^ aes_Rcon[i]
#### end key schedule core

for j in xrange(4):
  # unmix the bytes from the last subkey
  word[j] ^= invexkey[-self.key_size + j]
```

As you can see, we didn't need to inverse the key schedule core. Now we need to append our created subkey to the array `invexkey` and repeat the process until we have created all subkeys and the original cipherkey.

Getting the flag

Combining the code we created in the previous step with some code to interact with the service results in this included Python file. Running that script gives us the flag we wanted!

The flag is `31C3\_0748a7b8be603056aa9c391e`.

Other write-ups and resources

- https://github.com/VMXh/write-ups/blob/master/31C3%20CTF/hwaes/README.md
- http://vnsecurity.net/ctf%20-%20clgt%20crew/2014/12/30/31C3CTF-crypto-hwaes-writeup.html
- http://tasteless.eu/2014/12/31c3-ctf-hwaes/

- http://hxp.io/blog/8/31C3%20CTF:%20crypto10%20%22hwaes%22/
- http://www.kizhakkinan.com/?p=187