Cryptography land mines

Symmetric algorithm keys

The rule of thumb in cryptography says never to invent the wheel. In practical terms that means two things. First, one should never invent their own cryptoalgorithm. Anyway, it will be (almost certainly) worse than existing ones, and even worse, you will not even know that. Second, one should never implement already known algorithms but use crypto libraries that are widely available and proven to be secure. So, we should use existing wheels.

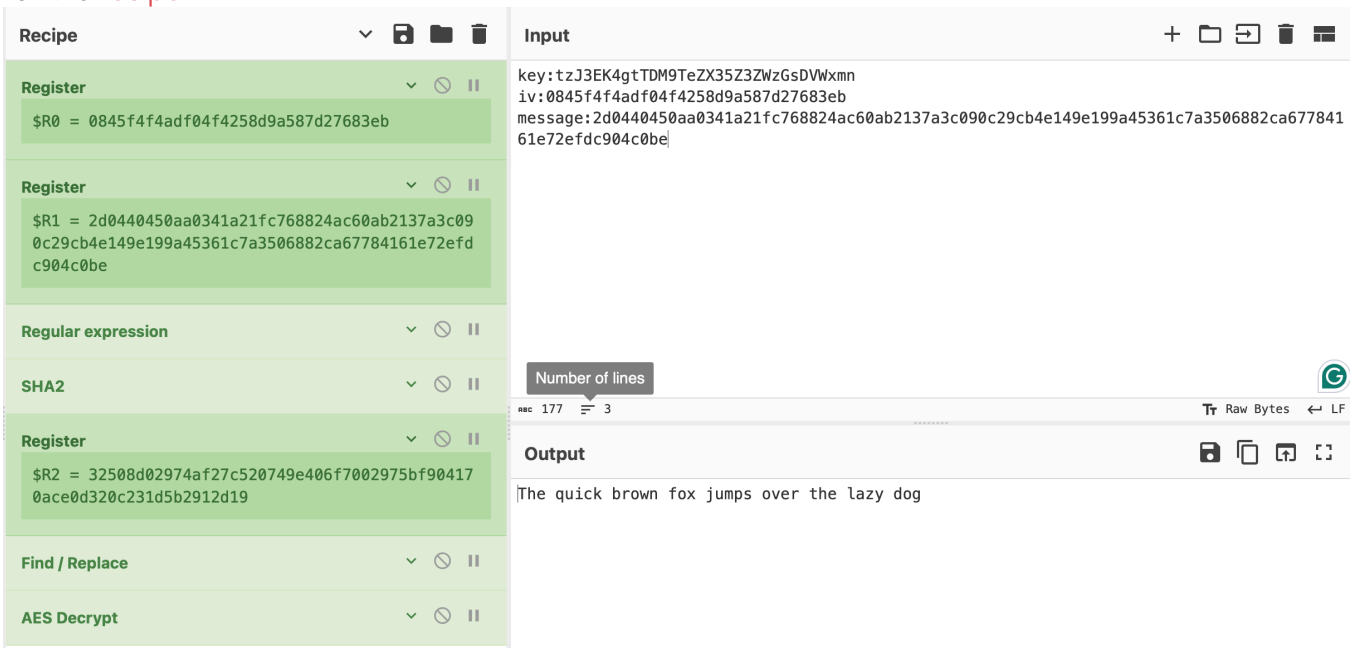
But can we use them properly?

```
import { randomBytes, createHash, createCipheriv } from 'crypto';

const secret = Buffer.from("tzJ3EK4gtTDM9TeZX35Z3ZWzGsDVWxmn");

function encrypt(text, secret) {
    const iv = randomBytes(16);
    const key = createHash('sha256').update(secret).digest();

    const cipher = createCipheriv('aes-256-cbc', key, iv);

    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');

    return {
        iv: iv.toString('hex'),
        content: encrypted
    };
}

console.log(encrypt("The quick brown fox jumps over the lazy dog", secret));
```

Above is the piece of source code (stripped and somewhat changed) that I stumbled upon not so long ago. Let's have a closer look at how it is structured and what it does. The first thing to mention is that this code was meant to be a part of the automation procedure, and a secret was going to be pulled from a secrets manager (here the secret is hardcoded for the sake of simplicity, otherwise a hardcoded secret is a good way to lose your bonus). An encryption function accepts the secret and a text. It generates an input vector, hashes the secret to produce an encryption key, does the encryption using the standard crypto library and splits out the result.

```
/app # node crypt.mjs
{
  iv: '0845f4f4adf04f4258d9a587d27683eb',
  message: '2d0440450aa0341a21fc768824ac60ab2137a3c090c29cb4e149e199a45361c7a3506882ca67784161e72efdc904c0be'
}
```

All neat. If you run the code, it beautifully returns the correct values. Let's check. To verify that the result is correct (and consequently, the code is correct), I will use the CyberChef tool. Here is the link for the recipe



But... there would not be this article if something wasn't wrong. First, let's revise some theory.

> The Advanced Encryption Standard (AES), ... is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. For AES, NIST selected three members of the Rijndael family, each with a block size of 128 bits, but three different key lengths: 128, 192 and 256 bits. [Advanced Encryption Standard - Wikipedia]

So, in simple words, AES-256 is an encryption algorithm, that can process blocks of 128 bits using a 256-bit key. And another piece of theory.

> Kerckhoffs's principle holds that a cryptosystem should be secure, even if everything about the system, except the key, is public knowledge. Combining these two pieces, we can state that for our cryptosystem to be secure, we must use strong keys (secrets). [Kerckhoffs's principle - Wikipedia]

Everyone knows that, right?s

Now let's have a look at the secret that we use. We can see that it is random, but unfortunately not random enough. In our case, the secret is a set of 32 alphanumeric symbols, making it exactly 256 bit length. Nevertheless, given that those are alphanumeric symbols, they occupy only a small chunk of

the ASCII table. Using this strategy, we restrict the possible value of each byte from 256 options (byte can have 2^8 values) to only 62 (26 uppercase letters + 26 lowercase letters + 10 numbers) cutting the strength of each byte four times, and the entire password - in 10^22 times. Experienced readers will notice, that the secret is hashed, effectively randomizing the key. So what is the problem? The problem is that the key remains a derivative of the initial secret not changing the strength of the secret material, since the potential attacker may know that the hashing is there (remember Kerckhoffs's principle - we must not rely on the system design, but only on the secret strength).

What to do? The solution is simple. The secret should be a random byte array, that is stored in hexadecimal or base64 form (since the majority of secrets management solutions expect strings as secret values) in a safe place (remember about bonuses).

To generate the secret you can use the following terminal commands:
on Linux

```
echo -n $(dd if=/dev/urandom bs=32 count=1 status=none) | sha256sum
```

on Mac

```
echo -n $(dd if=/dev/urandom bs=32 count=1 status=none) | shasum -a 256
```

or if you are on Windows (believe me, PowerShell generator is a bit wild), or if you are not a CLI type of person (which you need to be IMO) you can use this nodeJS snippet instead

```
import { createHash, randomBytes } from 'crypto'

console.log(createHash('sha256').update(randomBytes(32)).digest("hex"));
```

Here is the example using the nodeJS snippet

```
/app # node gengerate.mjs
6999e160ca53a3ccd18f0e3e7f4290dc44cdb2aeb9b0e30c07df90189d598a31
```

Eventually, the code will look like.

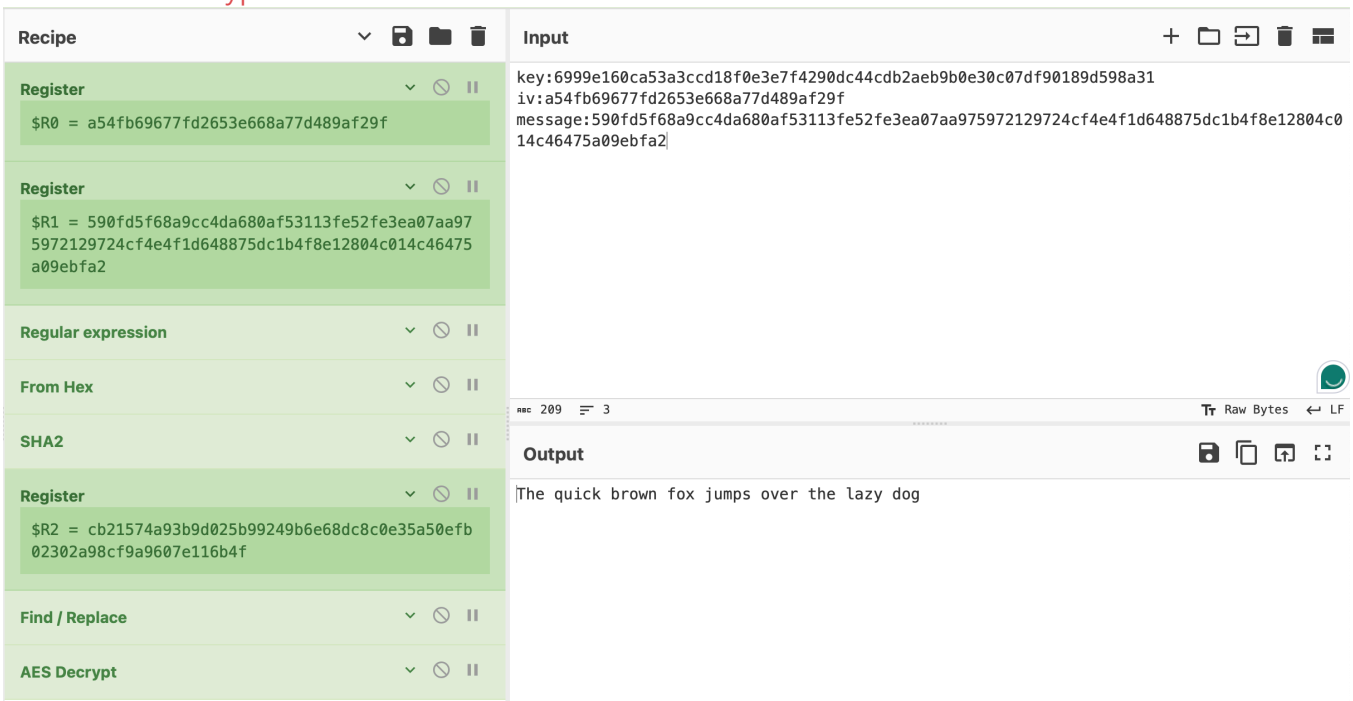
```
import { randomBytes, createHash, createCipheriv } from 'crypto';

const secret =
Buffer.from("6999e160ca53a3ccd18f0e3e7f4290dc44cdb2aeb9b0e30c07df90189d598a31",
"hex")

function encrypt(text, secret) {
    const iv = randomBytes(16);
    const key = createHash('sha256').update(secret).digest()

    const cipher = createCipheriv('aes-256-cbc', key, iv);
```

```
    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');

    return {
        iv: iv.toString('hex'),
        message: encrypted
    };
}

console.log(encrypt("The quick brown fox jumps over the lazy dog", secret))
```

But stop, why hashing is still in there? Practically it is redundant, but in theory, hashing improves the probability distribution rendering the final key more (better?) random. Effectively, from an attacker's standpoint of view, it does not matter what brute force the secret or the key value, since both are random. But hashing the secret makes your appsec engineer's heart feel better. However, you can remove the hashing if you have already hashed the secret before, for example in a terminal while generating one.

```
/app # node crypt-correct.mjs
{
  iv: 'a54fb69677fd2653e668a77d489af29f',
  message: '590fd5f68a9cc4da680af53113fe52fe3ea07aa975972129724cf4e4f1d648875dc1b4f8e12804c014c46475a09ebfa2'
}
```

Let's do the decryption.



Everything worked out as expected.

As for the conclusion, use the wheels correctly, and never be afraid to ask for help. Especially, when it comes to security matters.