JavaScript Programming Techniques for Security Programming

Characters, Codes, and Numbers

Every character you see on a screen is really just a **number** stored in memory.

Character	Decimal (ASCII)	Binary	Hexadecimal
`H`	72	`01001000`	`48`
`e`	101	`01100101`	`65`
`l`	108	`01101100`	`6c`
`o`	111	`01101111`	`6f`

ASCII is an older character set that uses 7 bits (0–127) to represent English letters, digits, and symbols.

Example:

- `A` → 65
- `a` → 97
- **UTF-8** is a **Unicode encoding** that includes **all ASCII characters** as its first 128 codes, and can also represent millions of characters from other languages (like 한, 日, 😊) using more bytes (up to 4).

In other words, **UTF-8 = ASCII + everything else**.

Hexadecimal (HEX) is a **base-16** way to represent binary data: Each byte (8 bits) = two HEX digits.

Examples:

- `A` → 65 (decimal) → `01000001` (binary) → `0x41` (HEX)
- `a` → 97 (decimal) → `01100001` (binary) → `0x61` (HEX)

0x prefix means “this number is in HEX.”

Example:

- **0x41** → 65 → `'A'`, and **0x61** → 97 → `'a'`
  (when encoded in **ASCII/UTF-8**).

Buffer (a chunk of binary memory)

A **Buffer** is like a **box full of raw bytes** — a chunk of memory that stores data in its **binary form** (0s and 1s).

- In JavaScript (Node.js), a `Buffer` lets you handle data **before** it's turned into text, images, or files.

Why Do We Need Buffers?

Computers don’t understand text like "Hello";
they only understand **bytes** — numbers between 0 and 255.

- So, when we send or receive data (like files, messages, or images), Node.js stores it first in a **Buffer** — raw binary form — before turning it into readable text.

Buffer Example

We use Buffer.from function to create a buffer in a memory.

```
const buf = Buffer.from("Hello", "utf8");
console.log(buf);         // <Buffer 48 65 6c 6c 6f>
console.log(buf.length); // 5 bytes
```

Inside memory (Buffer):

48	65	6c	6c	6f	← HEX
H	e	l	l	o	← ASCII / UTF-8

Each letter = 1 byte → 8 bits → 8 tiny switches (on/off).

Buffer to HEX string

The numbers (ASCII/UTF-8 characters) in a buffer can be shown as HEX string.

```
const buf = Buffer.from("Hello", "utf8");
const hex = buf.toString('hex');
console.log(hex); // "48656c6c6f"
```

So `48 | 65 | 6c | 6c | 6f` in buffer → "48656c6c6f"

From HEX string to UTF-8

```
const buf = Buffer.from("48656c6c6f", "hex")
const back = buf.toString("utf8");
console.log(back); // "Hello"
```

48	65	6c	6c	6f	← HEX
H	e	l	l	o	← ASCII / UTF-8

- The HEX string is decoded into the (same) bytes [72,101,108,108,111].
- UTF-8 interprets those bytes as characters again.

So `48 | 65 | 6c | 6c | 6f` in buffer → "Hello"

Object (Buffer) Manipulation in JavaScript

The `module.js`:

```
// Create a Buffer object with some data
var x = Buffer.from('Hello World', 'utf-8');

// Export as an object property
module.exports = { x };
```

Converting Between Buffer and String

To store any JavaScript objects, including buffer objects, we need to transform them into a string (buffer.js).

```
const { x } = require('./module'); // x is an Object (Buffer)

// Convert to hexadecimal string (for series of numbers)
var y = x.toString('hex');

// Convert to UTF-8 string
var y = x.toString('utf-8');

// Convert back to buffer object
var z = Buffer.from(y); // z is an object
```

Storing the Buffer in a File

This example shows how to store an object in a file and load an object from a file (files.js).

```
const fs = require('fs');
// Buffer with a content "Hello World"
const { x } = require('./module'); // x is an object
```

We need to change the buffer content into a string to store it in a file.

```
try {
    // 48656c6c6f20576f726c64
    fs.writeFileSync(filename, x.toString('hex'));
} catch (err) {
    console.log(err);
}
```

We read file content and convert it into an Object (Buffer).

```
try {
    // content = Buffer<34 38 36 35 36 63...>
    //   ← ASCII codes of '4','8','6','5','6','c'...
    var content = fs.readFileSync(filename);
    // content.toString() = "48656c6c6f20576f726c64"
    var c = Buffer.from(content.toString(), 'hex');
    // c = Buffer<48 65 6c 6c 6f...>
    // c.toString = "Hello World"
    console.log(c.toString('UTF-8'))
} catch (err) {
    console.log(err);
}
```

Detailed Explanation:

1. What we see in files are all ASCII/UTF-8 code: when we see `1`, it is not number 1, but number 49 (HEX 0x31).
2. What should be in Buffer are all numbers: when we want to store `1`, we should store 49.

```
var content = fs.readFileSync(filename);
```

The values in the content (buffer) are a series of characters: '4','8','6','5', ... (34, 38, 36, 35, ...), not the value 48, 65, so the content should be converted:

1. Using the content.toString() to interpret the 34('4'), 38('8') ... as a string '48'.
2. Then, `Buffer.from(, 'hex')` function to store (48, 65, ...)

Storing the JSON Object in a File

We can use `JSON.stringify()` function to store JSON objects.

```
const fs = require('fs');

try {
    // JSON object -> JSON string
    fs.writeFileSync(filename, JSON.stringify(users));
} catch (err) {
    console.log(err);
}
```

To retrieve JSON string into JSON object, we use `JSON.parse()` function.

```
try {
    var data = fs.readFileSync(filename);
    // JSON string -> JSON object
    return JSON.parse(data);
} catch (err) {
    console.log(err);
}
```

Warning: Use Sync Functions

When we use JavaScript file I/O for manipulating objects, we use sync functions (`readFileSync` or `writeFileSync`), not async functions with callback functions.

- This is to ensure that we get the information or store it before taking the next step.
- As a general rule, try to use sync functions when dealing with encryption or decryption, as data integrity is more important than data usability.

Encoding Multiple Values

When we write multiple pieces of information, for example strings "a", "b", and "c", we concatenate them with a boundary character such as ":".

For example, we store the information as `"a:b:c"`, and we load the information and restore it using the `split()` method.

```
var x = "x";
var y = "y";
var z = "z";

// Encoding
var str = `${x}:${y}:${z}`;

// Decoding
const [x2, y2, z2] = str.split(':');
```

Node.js Crypto Module

We use the `crypto` module to use JavaScript decrypt and encrypt functions.

- Each time a message is encrypted, it is randomized to produce a different output.
- So, we should also use the random function.

```
// ./encrypt_demo.js
const {
  createCipheriv,
  randomBytes,
  createDecipheriv
} = require('crypto');

// The message to encrypt
const message = 'I like ASE 285 students!';

// Generate a random 32-byte key (for AES-256)
const key = randomBytes(32);

// Generate a random 16-byte IV (Initialization Vector)
const iv = randomBytes(16);

// Create the cipher (AES-256 in CBC mode)
const cipher = createCipheriv('aes-256-cbc', key, iv);

// Encrypt the message
let encrypted = cipher.update(message, 'utf8', 'hex');
encrypted += cipher.final('hex');

console.log('Encrypted:', encrypted);
console.log('Key:', key.toString('hex'));
console.log('IV:', iv.toString('hex'));

// Decrypt to check
const decipher = createDecipheriv('aes-256-cbc', key, iv);
let decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');

console.log('Decrypted:', decrypted);
```