AST - Deobfuscation

1) Goal

Given an **AST JSON** of obfuscated JavaScript (`ch38.json`), recover the hidden strings by:

- Finding decoding **sinks** (calls to `String.fromCharCode`).
- Identifying the **transform** applied to numeric arrays (e.g., `>> n` or `^ key`).
- Resolving the **array source** feeding `.map(...)` .
- Computing the **key** if the RHS is an identifier (e.g., `sens`).
- Rebuilding the plaintext.

2) Terminology (used precisely)

- **AST**: Abstract Syntax Tree (structural representation of JS code).
- **Sink**: Code that emits the final string; here: `String.fromCharCode(...)` or `String.fromCodePoint(...)` .
- **LHS / RHS**: Left-hand side and right-hand side of a binary expression.
  - Example: `c ^ sens` → **LHS** is `c` , **RHS** is `sens`
  - Example: `c >> 4` → **LHS** is `c` , **RHS** is `4` .
- **Transform**: The operator used to transform numbers before turning them into characters (e.g., `>>` , `^`).
- **IIFE**: Immediately Invoked Function Expression, e.g. `(function(){ return [ ... ]; })()` .

3) Reading the AST: what to target

Search the AST for `String.fromCharCode` . Two relevant map-sites exist:

1. `array.map(c => String.fromCharCode(c >> 4)) .join('')`
   - Transform = `>>` (right shift).
   - RHS = `4` (literal).
   - Array is a **direct literal**: `[1856,1824,1776,1728,1776,1728,1776]` → Decodes to `trololo` .
2. `array.map(c => String.fromCharCode(c ^ sens)) .join('')`
   - Transform = `^` (XOR).
   - RHS = `sens` (identifier → compute its value).

- Array is produced by an **IIFE** returning an array of big integers:
  `[65353704,65353663,65353663,65353707, ...]`
  → After computing `sens`, XOR-decoding yields `g00d_j0b_easy_deobfuscation`.

4) Computing the key `sens` (exactly like the code)

AST shows:

```
let sens = [10] + [45] + [65] + [78] + [47];
sens >>= 4;
```

JS coercion logic:

- `[10] + [45] + ...` → arrays become strings via `.toString()` → `"10" + "45" + "65" + "78" + "47"` → `"1045657847"`.
- `Number("1045657847")` → `1045657847`.
- `sens >>= 4` → right shift by 4 bits (≈ integer division by 16) → `1045657847 >> 4 = 65353615`.

Command to verify:

```
node -e "const parts=[10,45,65,78,47]; const s=parts.map(String).join('');
const before=Number(s); const after=before>>4;
console.log({concat:s,before,after})"
```

Expected:

```
{ concat: '1045657847', before: 1045657847, after: 65353615 }
```

So `sens = 65353615`.

5) How the decoding works (math)

5.1 Right shift (“trololo”)

Decoding rule: `char = fromCharCode(c >> 4)`.
Example:

- `c = 1856`
- `1856 >> 4 = 116`

- `String.fromCharCode(116) = 't'`
  Repeat → `trololo`.
  Quick check:

```
node -e "const d=[1856,1824,1776,1728,1776,1728,1776];
console.log(d.map(c=>String.fromCharCode(c>>4)).join(''))"
```

5.2 XOR

Encoding logic (obfuscator): `stored = original_code ^ sens`.
Decoding logic (ours): `stored ^ sens = original_code`.
Property: `(A ^ B) ^ B = A`.

First element demo (complete)

- `sens = 65353615`
- `c = 65353704`

Binary (aligned):

```
c     = 11111001010011011111101000
sens  = 11111001010011011110001111
XOR   = 00000000000000000001100111  (only low bits differ)
```

`0b1100111 = 103` → `String.fromCharCode(103) = 'g'`.

Node check:

```
node -e "const sens=65353615; const c=65353704; const v=c^sens;
console.log('c=',c,'c^sens=',v,'char=',String.fromCharCode(v))"
```

Output:

```
c= 65353704 c^sens= 103 char= g
```

More elements:

```
node -e "const sens=65353615; [65353663,65353663,65353707].forEach(c=>{const
v=c^sens; console.log(c,'->',v,'->',String.fromCharCode(v))})"
```

Full decode:

```
node -e "const sens=65353615; const data=
[65353704,65353663,65353663,65353707,65353680,65353701,65353663,65353709,653
53680,65353706,65353710,65353724,65353718,65353680,65353707,65353706,6535369
6,65353709,65353705,65353722,65353724,65353708,65353710,65353723,65353702,65
353696,65353697];
console.log(data.map(c=>String.fromCharCode(c^sens)).join(''))"
```

Result: flag

8) Generalizing to other CTFs (expert method)

1. **Find sinks**: `fromCharCode` / `fromCodePoint`.
2. **Mark operators**: for each sink, note `transform` (`>>`, `^`, etc.) and **RHS** (literal vs identifier).
3. **Resolve the array**: literal / identifier init / IIFE / chain.
4. **If RHS literal**: auto-decode immediately.
5. **If RHS identifier**: locate its **definition**, **simulate** its operations (concat, shifts, XOR assigns), compute the final **numeric key**, decode.
6. **Verify** on the first element: print `c`, `c op key`, char.
7. **Decode full string**.

This pipeline scales to most JS obfuscation puzzles that hide strings via numeric arrays + simple bitwise transforms.