
SBA
Research

Illegal States Are My Favorite Security Vulnerabilities

(to remove at compile time)

BOB2026 – 2026-03-13

Federal Ministry Innovation, Mobility and Infrastructure Republic of Austria
Federal Ministry Economy, Energy and Tourism Republic of Austria
FFG Promoting Innovation.
vienna business agency
For the City of Vienna
European Commission
FWF Austrian Science Fund
netidee FÖRDERUNGEN

$ whoami

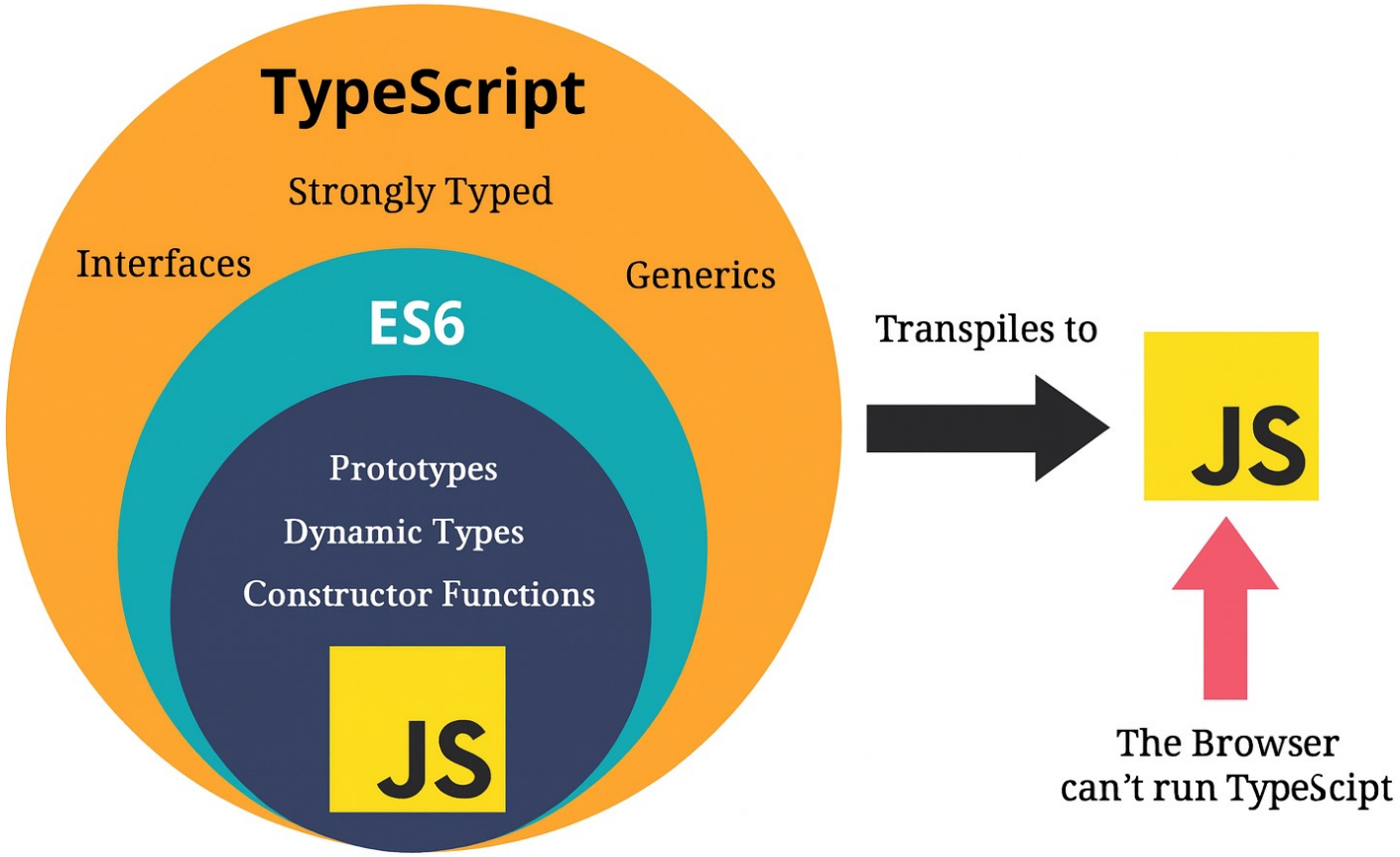
Michael Koppmann, Security Consultant

- Penetration tests
- Source code audits
- Architecture reviews
- Security training
- Software development

SBA Research

- Austrian Research Center for Information Security

JavaScript with Types



TypeScript and the Runtime

```
1. function processUserInput(userInput) {
2.     console.log(userInput.toUpperCase());
3. }
```

TypeScript and the Runtime

```
1. function processUserInput(userInput) {
2.     console.log(userInput.toUpperCase());
3. }

4. const maliciousInput = {
5.     toUpperCase: () => { console.log("Pwned!"); }
6. };

7. processUserInput(maliciousInput);
```

TypeScript and the Runtime

```
1. function processUserInput(userInput: string): void {
2.     console.log(userInput.toUpperCase());
3. }

4. const maliciousInput = {
5.     toUpperCase: () => { console.log("Pwned!"); }
6. };

7. processUserInput(maliciousInput);
8. // Argument of type '{ toUpperCase: () => void; }'
   is not assignable to parameter of type 'string'.
```

TypeScript and the Runtime

```
1. function processUserInput(userInput) {
2.     console.log(userInput.toUpperCase());
3. }
```

TypeScript and the Runtime

```
1. function processUserInput(userInput: unknown): void {
2.     if (typeof userInput !== 'string') {
3.         throw new TypeError('Expected a string.');
4.     }
5.
6.     console.log(userInput.toUpperCase());
7. }
```

```
1. import { z } from 'zod';

2. const UserInput = z.string();
3. function processUserInput(rawInput: unknown): void {
4.   const userInput = UserInput.parse(rawInput);
5.   console.log(userInput.toUpperCase());
6. }
```

The Power of Compiler Flags

Make your life easier, one flag at a time.

The Power of Compiler Flags

Intro to the TSConfig Reference

A TSConfig file in a directory indicates that the directory is the root of a TypeScript or JavaScript project...

Compiler Options

Top Level	files, extends, include, exclude and references
"compilerOptions"	
Type Checking	allowUnreachableCode, allowUnusedLabels, alwaysStrict, exactOptionalPropertyTypes, noFallthroughCasesInSwitch, noImplicitAny, noImplicitOverride, noImplicitReturns, noImplicitThis, noPropertyAccessFromIndexSignature, noUncheckedIndexedAccess, noUnusedLocals, noUnusedParameters, strict, strictBindCallApply, strictBuiltinIteratorReturn, strictFunctionTypes, strictNullChecks, strictPropertyInitialization and useUnknownInCatchVariables

Source: https://www.typescriptlang.org/tsconfig/

`strict` Mode: The One Flag to Rule Them All

- alwaysStrict
- strictNullChecks
- strictBindCallApply
- strictBuiltinIteratorReturn
- strictFunctionTypes
- strictPropertyInitialization
- noImplicitAny
- noImplicitThis
- useUnknownInCatchVariable

```
1. {
2.   …
3.   "compilerOptions": {
4.     "strict": true,
5.   },
6.   …
7. }
```

`noImplicitAny`: Remove the Hidden Loopholes

noImplicitAny: Remove the Hidden Loopholes

```
1. function fn(s) {
2.   console.log(s.startWith("1"));
3. }

4. fn(42);
5. // TypeError: s.startWith is not a function
```

```
"noImplicitAny": false
```

noImplicitAny: Remove the Hidden Loopholes

```
1. function fn(s) {
2.   console.log(s.startWith("1"));
3. }

4. fn(42);
5. // Parameter 's' implicitly has an 'any' type.
```

```
"noImplicitAny": true
```

noImplicitAny: Remove the Hidden Loopholes

```
1. function fn(s: string): void {
2.   console.log(s.startWith("1"));
3. }

4. fn(42);
5. // Argument of type 'number' is not assignable to
   parameter of type 'string'.
```

```
"noImplicitAny": true
```

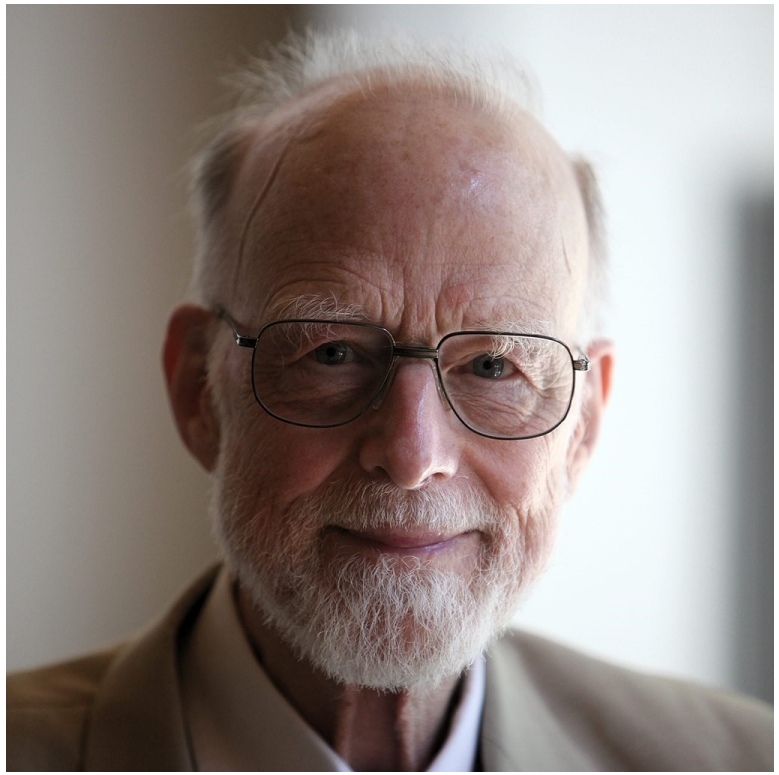
noImplicitAny: Remove the Hidden Loopholes

```
1. function fn(s: string): void {
2.   console.log(s.startWith("1"));
3. }

4. fn("42");
5. // Property 'startWith' does not exist on type
   'string'. Did you mean 'startsWith'?
```

```
"noImplicitAny": true
```

`strictNullChecks`: Refuse the Gift



Source: https://commons.wikimedia.org/wiki/File:Sir_Tony_Hoare_IMG_5125.jpg

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965."
Tony Hoare

`strictNullChecks`: Refuse the Gift

```
1. fn divide(numerator: f64, denominator: f64) -> Option<f64>
   {
2.     if denominator == 0.0 {
3.         None
4.     } else {
5.         Some(numerator / denominator)
6.     }
7. }

8. let result = divide(2.0, 3.0);
9. match result {
10.     Some(x) => println!("Result: {x}"),
11.     None    => println!("Cannot divide by 0"),
12.}
```

`strictNullChecks`: Refuse the Gift

```
1. function printName(obj: { first: string; last?: string }) {
2.   // …
3. }

4. printName({ first: "Bob" });
5. printName({ first: "Alice", last: "Alisson" });
```

`strictNullChecks`: Refuse the Gift

```
1. const loggedInUser =
        users.find((u) => u.name === loggedInUsername);
2. console.log(loggedInUser.age);
```

```
"strictNullChecks": false
```

`strictNullChecks`: Refuse the Gift

```
1. const loggedInUser =
        users.find((u) => u.name === loggedInUsername);
2. console.log(loggedInUser.age);


3. // 'loggedInUser' is possibly 'undefined'
```

```
"strictNullChecks": true
```

`strictNullChecks`: Refuse the Gift

```
1. const loggedInUser: User =
2.     users.find((u) => u.name === loggedInUsername);
3. console.log(loggedInUser.age);

4. // Type 'User | undefined' is not assignable to type 'User'.
5. // Type 'undefined' is not assignable to type 'User'.
```

```
"strictNullChecks": true
```

`exactOptionalPropertyTypes`: More Precise Object Definitions

```
1. interface UserDefaults {
2.   // The absence of a value represents 'system'
3.   colorThemeOverride?: "dark" | "light";
4. }
```

```
"exactOptionalPropertyTypes": false
```

`exactOptionalPropertyTypes`: More Precise Object Definitions

```
1. interface UserDefaults {
2.   // The absence of a value represents 'system'
3.   colorThemeOverride?: "dark" | "light";
4. }

5. const def: UserDefaults = { colorThemeOverride: undefined };
6. // Type '{ colorThemeOverride: undefined; }' is not
   assignable to type 'UserDefaults' with
   'exactOptionalPropertyTypes: true'. Consider adding
   'undefined' to the types of the target's properties.
```

```
"exactOptionalPropertyTypes": true
```

Gonna Flag Them All!

```
{
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "strictBindCallApply": true,
    "strictPropertyInitialization": true,
    "strictBuiltinIteratorReturn": true,
    "noImplicitThis": true,
    "useUnknownInCatchVariables": true,
    "alwaysStrict": true,
    "noUnusedLocals": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "exactOptionalPropertyTypes": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "noUncheckedIndexedAccess": true,
    "noImplicitOverride": true,
    "noPropertyAccessFromIndexSignature": true,
    "allowUnusedLabels": false,
    "allowUnreachableCode": false
}
```

Source: https://www.typescriptlang.org/tsconfig/

Type System Patterns

Model your domain, the functional way

Sum Types: Supercharged Enums

Also known as:

- Tagged Unions
- Discriminated Unions
- Variant Types

Allow presenting closed relationships of types.

People who like sum types also like: *Pattern Matching*

Sum Types: Supercharged Enums

```
1. type Shape =
2.   | { kind: "circle"; radius: number }
3.   | { kind: "rectangle"; width: number; height: number }
4.   | { kind: "triangle"; base: number; height: number };

5. function getArea(shape: Shape): number {
6.   switch (shape.kind) {
7.     case "circle": return Math.PI * shape.radius ** 2;
8.     case "rectangle": return shape.width * shape.height;
9.     case "triangle": return (shape.base * shape.height) / 2;
10.    default:
11.      const _exhaustiveCheck: never = shape;
12.      return _exhaustiveCheck;
13.  }
14.}
```

```ts
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "rectangle"; width: number; height: number }
  | { kind: "triangle"; base: number; height: number }
  | { kind: "square"; side: number };

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle": return Math.PI * shape.radius ** 2;
    case "rectangle": return shape.width * shape.height;
    case "triangle": return (shape.base * shape.height) / 2;
    default:
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}
// Type '{ kind: "square"; side: number; }' is not assignable to type 'never'
```

Smart Constructors: Validate Untrusted Inputs

External data is untrusted and needs to be:

- Canonicalized
- Validated
- Sanitized

Do this once in an immutable value, then enjoy the happy path!

Smart Constructors: Validate Untrusted Inputs

```
1. class UserAge {
2.   readonly value: number;

3.   constructor(value: number) {
4.     if (value < 0 || value > 150 || !Number.isInteger(value)) {
5.       throw new Error("Invalid user age");
6.     }

7.     this.value = value;
8.   }
9. }
```

```ts
class UserAge {
  readonly value: number;

  private constructor(value: number) {
    if (value < 0 || value > 150 || !Number.isInteger(value)) {
      throw new Error("Invalid user age");
    }

    this.value = value;
  }

  static create(value: number): UserAge | undefined {
    try {
      return new UserAge(value);
    } catch (err: unknown) {
      return undefined;
    }
  }
}
```

Smart Constructors: Validate Untrusted Inputs

```
1. declare const __brand: unique symbol
2. type Brand<B> = { [__brand]: B }
3. export type Branded<T, B> = T & Brand<B>
```

Smart Constructors: Validate Untrusted Inputs

```
1. type Age = Branded<number, "Age">;

2. function createAge(input: number): Age | undefined {
3.   if (input < 0 || input > 150 || !Number.isInteger(input)) {
4.     return undefined;
5.   }

6.   return input as Age;
7. }

8. function getBirthYear(age: Age) {
9.     return new Date().getFullYear() - age;
10.}
```

Smart Constructors: Validate Untrusted Inputs

Drawbacks of Branded types:

- You cannot export a type but hide its constructor
- Any function could just cast data to a branded type
- Needs to be checked during code review

Use validated types or classes to differentiate between:

- Currencies
- Raw and sanitized HTML
- Personal and anonymized data
- Different domain concepts

Result Types: Replace Imperative Exception Handling

```
1. export type Result<T, E> =
       { success: true; value: T } | { success: false; error: E };

2. export function makeSuccess<T>(value: T): Result<T, never> {
3.   return { success: true, value };
4. }

5. export function makeFailure<E>(error: E): Result<never, E> {
6.   return { success: false, error };
7. }
```

Result Types: Replace Imperative Exception Handling

```
1. function safeParseJSON(input: string): Result<unknown, Error> {
2.   try {
3.     return makeSuccess(JSON.parse(input));
4.   } catch (error: unknown) {
5.     return makeFailure(error as Error);
6.   }
7. }
8.
   const result = safeParseJSON('{ "foo": "bar" }');

9. if (result.success) {
10.  console.log("Parsed object:", result.value);
11.} else {
12.  console.error("Invalid JSON:", result.error.message);
13.}
```

TypeScript in Action

Yes, you can actually use these patterns!

Eliminate Boolean Flags

Eliminate Boolean Flags

```
1. interface AuthState {
2.     isAuthenticated: boolean;
3.     isBanned: boolean;
4. }
```

Eliminate Boolean Flags

```
type AuthState =
  | { status: "unauthenticated" }
  | { status: "authenticated"; token: string; userId: number }
  | { status: "banned"; reason: string };
```

```
function performSecureAction(auth: AuthState): void {
  switch (auth.status) {
    case "unauthenticated":
      console.warn("You must log in first.");
      break;
    case "authenticated":
      console.log(`Performing secure action for user ${auth.userId}`);
      break;
    case "banned":
      console.warn(`Access denied: ${auth.reason}`);
      break;
    default:
      const _exhaustiveCheck: never = auth;
      return _exhaustiveCheck
  }
}
```

Type-Safe IDs: Prevent ID Mix-Ups

```
function canAccessOrder(userId: string, orderId: string): Boolean { … }
```

Type-Safe IDs: Prevent ID Mix-Ups

```
1. type UserId = Branded<string, "UserId">;
2. type OrderId = Branded<string, "OrderId">;

3. function canAccessOrder(userId: UserId, orderId: OrderId): Boolean { … }

4. if (canAccessOrder(orderId, userId)) { … }

5. // Argument of type 'OrderId' is not assignable to parameter of type
   'UserId'.
```

Enforce Business Rules: A Type-Safe Email Verification Flow

```
1. interface User {
2.     email: string;
3.     isVerified: boolean;
4. }
```

Enforce Business Rules: A Type-Safe Email Verification Flow

```
1. import { Branded } from "./Branded";
2. import { makeFailure, makeSuccess, Result } from "./Result";


3. export type VerificationCode = Branded<string,
   "VerificationCode">;
4. export type UnverifiedEmail = Branded<string,
   "UnverifiedEmail">;
```

Enforce Business Rules: A Type-Safe Email Verification Flow

```
export class VerifiedEmail {
  private constructor(readonly value: string) {}

  static verify(
    email: UnverifiedEmail,
    code: VerificationCode
  ): Result<VerifiedEmail, Error> {
    // Check if given code matches code in database
    return makeSuccess(new VerifiedEmail(email));
  }
}
```

Enforce Business Rules: A Type-Safe Email Verification Flow

```
export type EmailContactInfo =
  | { kind: "Unverified"; value: UnverifiedEmail }
  | { kind: "Verified"; value: VerifiedEmail };

export function newEmailContactInfo(address: string): EmailContactInfo { … }

export function sendAndStoreVerificationCode(
  contact: EmailContactInfo, code: VerificationCode
): void { … }

export function verifyEmailContactInfo(
  contact: EmailContactInfo, code: VerificationCode
): Result<EmailContactInfo, Error> { … }
```

Enforce Business Rules: A Type-Safe Email Verification Flow

```
1. export function newEmailContactInfo(address: string): EmailContactInfo {
2.   return { kind: "Unverified", value: address as UnverifiedEmail };
3. }
```

Enforce Business Rules: A Type-Safe Email Verification Flow

```
1. export function sendAndStoreVerificationCode(
2.   contact: EmailContactInfo,
3.   code: VerificationCode
4. ): void {
5.   switch (contact.kind) {
6.     case "Verified": {
7.       return;
8.     }
9.     case "Unverified": {
10.      // Store verification code in database for address and send mail
11.    }
12.    default: { // Exhaustiveness check }
13.  }
14.}
```

```
export function verifyEmailContactInfo(
  contact: EmailContactInfo,
  code: VerificationCode
): Result<EmailContactInfo, Error> {
  switch (contact.kind) {
    case "Verified": {
      return makeSuccess(contact);
    }
    case "Unverified": {
      const result = VerifiedEmail.verify(contact.value, code);

      return result.success
        ? makeSuccess({ kind: "Verified", value: result.value })
        : makeFailure(result.error);
    }
    default: { // Exhaustiveness check }
  }
}
```

Enforce Business Rules: A Type-Safe Email Verification Flow

```
1. interface User {
2.   email: EmailContactInfo;
3. }

4. function sendPasswordResetEmail(email: VerifiedEmail) { … }

5. function changeEmail(address: string): EmailContactInfo { … }
```

Key Takeaways

- **Make illegal state unrepresentable**
- Encode business rules in your types
- Validate once and stay on the happy path
- Use the compiler to your advantage
- Eliminate vulnerabilities by design

Make Security a First-Class Citizen

Security isn’t just about fixing vulnerabilities – it’s about designing software in a way that prevents them in the first place.

Michael Koppmann

SBA Research

Floragasse 7, 1040 Vienna, Austria

E-Mail: mkoppmann@sba-research.org

Matrix: @mkoppmann:sba-research.org