🛡️ ZodSheriff

Safe execution and validation of LLM-generated Zod schemas

🎯 Key Features

- 🔒 **LLM Safety First**: Safely execute and validate schemas generated by AI models, preventing potential security vulnerabilities and runtime errors
- 🧹 **Automatic Schema Cleanup**: Automatically exports valid schemas as constants, removes unsafe code paths, and preserves helpful comments
- ⚡ **Performance Optimized**: Smart caching and efficient validation with configurable resource limits to prevent DoS attacks
- 🎛️ **Flexible Configuration**: Choose from preset configurations (extremely safe, medium, relaxed) or customize validation rules
- 🚨 **Comprehensive Validation**: Deep validation of method chains, arguments, and object structures with detailed error reporting
- 📝 **TypeScript Ready**: Full TypeScript support with comprehensive type definitions

🚀 Quick Start

```
# Install globally
npm install -g zodsheriff

# Validate a schema file
zodsheriff schema.ts

# Use with specific safety configuration
zodsheriff --config extremelySafe schema.ts

# Read from clipboard
zodsheriff --clipboard

# Output only the cleaned schema
zodsheriff --clean-only schema.ts
```

📦 Installation

```
# As a global CLI tool
npm install -g zodsheriff

# As a project dependency
npm install zodsheriff
```

💻 Usage

Command Line Interface

```
zodsheriff [options] [file]

Options:
  --stdin           Read schema from standard input
  --clipboard       Read schema from system clipboard
  --config          Set validation config: extremelySafe | medium | relaxed
  --clean-only      Output only the cleaned schema
  --json           Output result in JSON format
```

As a Module

```
import { validateZodSchema, extremelySafeConfig } from 'zodsheriff';

const schemaCode = `
  import { z } from 'zod';
  export const userSchema = z.object({
    name: z.string(),
    age: z.number()
  });
`;

// Using default config (relaxed)
const result = await validateZodSchema(schemaCode);

// Using specific config
const safeResult = await validateZodSchema(schemaCode, extremelySafeConfig);

console.log(result.isValid);        // true/false
console.log(result.cleanedCode);    // Cleaned and formatted schema
console.log(result.issues);         // Array of validation issues
```

🏗️ Design Philosophy

ZodSheriff was built with a clear focus on safely handling LLM-generated schemas while maintaining developer convenience. Here are the key design decisions that shape the project:

Safety First Approach

1. **Schema Structure**
    - Only allows const declarations for schemas
    - Automatically exports all valid schemas
    - Preserves TypeScript types and JSDoc comments
    - Removes any unsafe or unnecessary code paths

2. **Method Chain Validation**
    - Validates each method in the chain against allowed Zod methods
    - Prevents arbitrary code execution through method arguments
    - Enforces safe argument patterns for methods like refine() and transform()
3. **Resource Protection**
    - Configurable limits on:
        - Chain depth
        - Object nesting
        - Property counts
        - String lengths
        - Total node count
    - Timeout protection against infinite loops
    - Prevention of regex DoS attacks
4. **Property Safety**
    - Protects against prototype pollution
    - Configurable allowed/denied property names
    - Safe handling of computed properties
    - Prevention of unsafe object methods

Convenience Features

1. **Schema Preservation**
    - Keeps original formatting where safe
    - Maintains developer comments and documentation
    - Preserves type annotations
    - Retains whitespace and code style
2. **Smart Validation**
    - Caches validation results for performance
    - Provides specific error messages with line numbers
    - Suggests fixes for common issues
    - Allows different severity levels (ERROR, WARNING, INFO)
3. **Progressive Enhancement**
    - Three preset configurations (extremely safe, medium, relaxed)
    - Customizable validation rules
    - Optional runtime protection
    - Configurable performance settings

🔍 Technical Deep Dive

Validation Process

1. **Parsing and AST Analysis**

```
// 1. Parse code to AST
const ast = parse(schemaCode);

// 2. Validate imports
validateZodImport(ast);

// 3. Process declarations
validateVariableDeclaration(node);

// 4. Validate method chains
validateChainNode(node, depth);

// 5. Clean and export
generateCleanCode(ast);
```

2. **Chain Validation**
   - Validates each method call in the chain
   - Ensures proper method order
   - Checks argument safety
   - Prevents unsafe combinations
3. **Resource Management**

```
class ResourceManager {
  // Track and limit resource usage
  private nodeCount = 0;
  private readonly depthMap = new Map<number, number>();

  // Enforce limits
  public incrementNodeCount(): void {
    if (++this.nodeCount > this.config.maxNodeCount) {
      throw new ValidationError("Node count exceeded");
    }
  }
}
```

Safety Mechanisms

1. **Method Safety**

```
// Only allowed Zod methods
const allowedZodMethods = new Set([
  "string", "number", "boolean",
  "object", "array", "union",
  // ...more methods
]);
```

2. **Argument Validation**

```
// Example: Safe refine validation
validateRefineArgument(node: CallExpression): boolean {
  // Must be a function
  if (!isFunction(node.arguments[0])) return false;

  // No async or generator functions
  if (node.arguments[0].async || node.arguments[0].generator) {
    return false;
  }

  // Validate function body
  return validateFunctionBody(node.arguments[0].body);
}
```

3. **Property Protection**

```
const deniedProperties = new Set([
  "__proto__",
  "constructor",
  "prototype"
]);
```

🔮 Future Work

1. **Enhanced Safety Features**
   - Runtime schema execution sandboxing
   - More granular method argument validation
   - Custom validation rule definitions
   - Dynamic safety level adjustment
2. **Developer Experience**
   - VSCode extension for real-time validation
   - Interactive schema fixing suggestions
   - Schema visualization tools
   - Integration with popular AI coding assistants
3. **Performance Optimizations**

    - Parallel validation for large schemas
    - Smarter caching strategies
    - Incremental validation
    - WebAssembly acceleration
4. **Extended Functionality**
    - Support for more Zod features
    - Custom method definitions
    - Schema transformation tools
    - Integration with other validation libraries
5. **Ecosystem Integration**
    - Better TypeScript integration
    - Framework-specific plugins
    - CI/CD tools
    - Schema registry support