CRISP Coding Standards

Overview

This document outlines the coding standards and conventions used in the CRISP (Cybersecurity Resilience through Information Sharing Platform) project. CRISP is a Django-based threat intelligence platform with a React frontend, implementing STIX/TAXII standards for threat data sharing.

Project Structure

Backend (Django)

- **Framework**: Django 4.2.10 with Django REST Framework
- **Database**: PostgreSQL with psycopg2-binary
- **Architecture**: Service-oriented with Repository pattern
- **Authentication**: JWT-based with django-rest-framework-simplejwt
- **Task Queue**: Celery with Redis

Frontend (React)

- **Framework**: React 19.1.0 with Vite 6.3.5
- **Routing**: React Router DOM 6.28.0
- **Styling**: CSS modules and FontAwesome icons
- **Build Tool**: Vite with ESLint configuration

Directory Structure Standards

Backend Structure

```
core/
├── api/                    # API endpoints and views
├── config/                 # Configuration files
├── management/commands/    # Django management commands
├── middleware/             # Custom middleware components
├── models/                 # Database models
├── parsers/               # Data parsers (STIX, etc.)
├── patterns/              # Design patterns implementation
│   ├── decorator/         # Decorator pattern
│   ├── factory/           # Factory pattern
│   ├── observer/          # Observer pattern
│   └── strategy/          # Strategy pattern
├── repositories/          # Repository pattern implementations
├── serializers/           # DRF serializers
├── services/              # Business logic services
├── tasks/                 # Celery tasks
├── tests/                 # Test files
└── validators/            # Custom validators
```

Frontend Structure

```
src/
├── components/            # React components
│   ├── enhanced/          # Enhanced/advanced components
│   ├── dashboard/         # Dashboard components
│   ├── threat/           # Threat-related components
│   └── user/             # User management components
├── assets/               # Static assets
├── data/                 # Static data files
└── api.js               # API client functions
```

Naming Conventions

Python (Backend)

- **Files**: snake_case (e.g., `auth_service.py`, `threat_feed_views.py`)
- **Classes**: PascalCase (e.g., `AuthenticationService`, `ThreatFeedRepository`)
- **Functions/Methods**: snake_case (e.g., `authenticate_user`, `get_by_stix_id`)
- **Variables**: snake_case (e.g., `user_session`, `auth_result`)
- **Constants**: SCREAMING_SNAKE_CASE (e.g., `MAX_TRUST_LEVEL`)

- **Private methods**: Prefix with underscore (e.g., `_get_client_info`)

JavaScript/React (Frontend)

- **Files**: PascalCase for components (e.g., `UserManagement.jsx`)
- **Components**: PascalCase (e.g., `LoadingSpinner`, `ConfirmationModal`)
- **Functions**: camelCase (e.g., `getUsersList`, `createUser`)
- **Variables**: camelCase (e.g., `currentUser`, `showModal`)
- **Constants**: SCREAMING_SNAKE_CASE (e.g., `API_BASE_URL`)

Database

- **Tables**: Plural, snake_case (e.g., `custom_users`, `threat_feeds`)
- **Columns**: snake_case (e.g., `created_at`, `trust_level`)
- **Foreign Keys**: `{model}_id` (e.g., `organization_id`)

Code Style Guidelines

Python Style

- **Line Length**: Maximum 120 characters
- **Imports**: Group in order: standard library, third-party, local imports
- **Docstrings**: Use triple quotes for all functions and classes
- **Type Hints**: Use where applicable, especially in service classes
- **Error Handling**: Use specific exception types with meaningful messages

Example Python Code Style:

```
"""
Authentication Service - JWT-based authentication with trust integration
Handles user authentication, session management, and security features
"""

from typing import Dict, Optional, Tuple
from django.contrib.auth import authenticate
from rest_framework_simplejwt.tokens import RefreshToken

class AuthenticationService:
    """Enhanced authentication service with trust-aware access control"""

    def __init__(self):
        self.trust_service = TrustService()

    def authenticate_user(self, username: str, password: str,
                          request=None, remember_device: bool = False) -> Dict:
        """Authenticate user with comprehensive security checks"""
        auth_result = {
            'success': False,
            'user': None,
            'message': ''
        }

        try:
            # Implementation here
            pass
        except Exception as e:
            logger.error(f"Authentication failed: {e}")
            return auth_result
```

JavaScript/React Style

- **Line Length**: Maximum 120 characters
- **Semicolons**: Always use semicolons
- **Quotes**: Use single quotes for strings, double quotes for JSX attributes
- **Arrow Functions**: Prefer arrow functions for functional components
- **Destructuring**: Use object destructuring where appropriate
- **Hooks**: Follow React hooks rules (use at top level)

Example React Code Style:

```
import React, { useState, useEffect } from 'react';
import { getUsersList, createUser } from '../../api.js';
import LoadingSpinner from './LoadingSpinner.jsx';

const UserManagement = ({ active = true, initialSection = null }) => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchUsers = async () => {
      try {
        const response = await getUsersList();
        setUsers(response.data);
      } catch (err) {
        setError('Failed to fetch users');
      } finally {
        setLoading(false);
      }
    };

    fetchUsers();
  }, []);

  return (
    <div className="user-management">
      {loading ? <LoadingSpinner /> : /* component content */}
    </div>
  );
};

export default UserManagement;
```

Architecture Patterns

Design Patterns Used

1. **Repository Pattern**: Data access abstraction (`IndicatorRepository`, `ThreatFeedRepository`)
2. **Service Pattern**: Business logic encapsulation (`AuthService`, `TrustService`)
3. **Factory Pattern**: Object creation (`StixFactory`, `TaxiiServiceFactory`)
4. **Observer Pattern**: Event handling (`FeedObservers`, `AlertSystemObserver`)
5. **Decorator Pattern**: Feature enhancement (`StixDecorator`)

6. **Strategy Pattern**: Algorithm selection (`AnonymizationStrategies`)

Service Layer Guidelines

- Services handle business logic and orchestration
- Repositories handle data access only
- API views are thin and delegate to services
- Use dependency injection where possible

Testing Standards

Backend Testing

- **Framework**: Django's built-in testing framework
- **Coverage**: Aim for 80%+ test coverage
- **Test Types**: Unit tests, integration tests, end-to-end tests
- **File Naming**: `test_*.py` (e.g., `test_auth_service.py`)
- **Test Methods**: `test_*` prefix (e.g., `test_authenticate_user_success`)

Frontend Testing

- **Framework**: ESLint for linting, no specific testing framework configured
- **Component Testing**: Test user interactions and state changes
- **API Testing**: Mock API calls in component tests

Security Guidelines

Backend Security

- **Authentication**: JWT tokens with refresh mechanism
- **Authorization**: Role-based access control (RBAC)
- **Input Validation**: Validate all user inputs
- **SQL Injection**: Use Django ORM, avoid raw SQL
- **CSRF Protection**: Django CSRF middleware enabled
- **Logging**: Log security events (authentication, authorization failures)

Frontend Security

- **XSS Prevention**: Sanitize user inputs
- **API Keys**: Never expose API keys in frontend code
- **HTTPS**: Always use HTTPS in production
- **Token Storage**: Store JWT tokens securely

Database Standards

Model Guidelines

- Use Django's built-in fields where possible
- Add `created_at` and `updated_at` timestamps to all models
- Use UUIDs for primary keys where appropriate
- Implement soft deletes for important data

Migration Guidelines

- Always create migrations for model changes
- Test migrations in development before production
- Use descriptive migration names
- Backup database before running migrations in production

API Standards

REST API Guidelines

- Use standard HTTP methods (GET, POST, PUT, DELETE)
- Use appropriate HTTP status codes
- Implement pagination for list endpoints
- Use consistent URL patterns (`/api/v1/resource/`)
- Include API versioning in URLs

Response Format

```
{
  "success": true,
  "data": {},
  "message": "Operation successful",
  "pagination": {
    "page": 1,
    "total_pages": 10,
    "total_items": 100
  }
}
```

Documentation Standards

Code Documentation

- **Python**: Use docstrings for all classes and methods
- **JavaScript**: Use JSDoc comments for complex functions
- **README**: Maintain up-to-date README files
- **API Docs**: Use drf-yasg for automatic API documentation

Commit Messages

- Use conventional commit format
- Include issue numbers when applicable
- Write clear, descriptive commit messages

Example:

```
feat: add user authentication with 2FA support

- Implement JWT-based authentication
- Add two-factor authentication using TOTP
- Update user model with security fields

Fixes #123
```

Performance Guidelines

Backend Performance

- Use database indexing for frequently queried fields
- Implement caching with Redis for expensive operations
- Use select_related() and prefetch_related() for Django ORM queries
- Implement pagination for large datasets

Frontend Performance

- Use React.memo() for expensive components
- Implement lazy loading for large components
- Optimize bundle size with Vite
- Use async/await for API calls

Environment and Configuration

Environment Variables

- Use `.env` files for configuration
- Never commit sensitive data to version control
- Use different configurations for development/production
- Document all required environment variables

Dependencies

- Pin dependency versions in requirements.txt and package.json
- Regularly update dependencies for security patches
- Use virtual environments for Python development
- Run security audits on dependencies

Tools and Linting

Backend Tools

- **Black**: Code formatting (if configured)
- **flake8**: Linting (if configured)
- **isort**: Import sorting (if configured)
- **Django Debug Toolbar**: Development debugging

Frontend Tools

- **ESLint**: Configured with React plugins
- **Prettier**: Code formatting (if configured)
- **Vite**: Build tool and development server

IDE Configuration

- Use consistent IDE settings across team
- Configure auto-formatting on save
- Enable linting in IDE
- Use consistent indentation (4 spaces for Python, 2 for JavaScript)

Conclusion

These coding standards ensure consistency, maintainability, and security across the CRISP project. All team members should follow these guidelines and update them as the project evolves. Regular code reviews and automated tools help enforce these standards.

For questions or clarifications about these standards, please refer to the project documentation or ask any one of our team members.