CRISP Testing Policy Document

Document Information

- **Document Title**: Testing Policy for CRISP (Cyber Risk Information Sharing Platform)
- **Version**: 1.0
- **Prepared by**: CRISP Development Team
- **Approved by**: Project Lead

Table of Contents

1. Introduction

1.1 Purpose

This document establishes the testing policy and procedures for the CRISP (Cyber Risk Information Sharing Platform) project. It defines the testing strategy, tools, processes, and standards that must be followed to ensure the delivery of high-quality, secure, and reliable software.

1.2 Scope

This policy applies to all components of the CRISP system, including:

- Backend Django application
- Frontend React application
- Database operations
- API integrations
- Security components
- Third-party integrations

1.3 Objectives

- Ensure software quality through comprehensive testing
- Minimize defects in production

- Maintain security standards
- Enable continuous integration and deployment
- Provide automated testing feedback
- Ensure compliance with cybersecurity requirements

2. Testing Strategy

2.1 Testing Pyramid

The CRISP project follows the testing pyramid approach:

```
    /\
   /  \     E2E Tests (10%)
  /____\    Integration Tests (20%)
 /______\   Unit Tests (70%)
```

2.1.1 Unit Tests (70%)

- **Scope**: Individual functions, methods, and classes
- **Tools**: pytest (Python), Vitest (JavaScript)
- **Coverage Target**: 85% minimum
- **Execution**: Every commit, local development

2.1.2 Integration Tests (20%)

- **Scope**: Component interactions, API endpoints, database operations
- **Tools**: pytest-django, Django TestCase, React Testing Library
- **Coverage Target**: 75% of critical paths
- **Execution**: Every pull request

2.1.3 End-to-End Tests (10%)

- **Scope**: Complete user workflows
- **Tools**: Playwright
- **Coverage Target**: 90% of user journeys
- **Execution**: Pre-deployment, release candidates

2.2 Testing Types

2.2.1 Functional Testing

- Unit testing
- Integration testing
- API testing
- User interface testing
- End-to-end testing

2.2.2 Non-Functional Testing

- Performance testing (Locust)
- Security testing (Bandit, Safety)
- Load testing
- Compatibility testing

2.2.3 Security Testing

- Vulnerability scanning
- Code security analysis
- Penetration testing
- Dependency security checks

3. Testing Tools and Technologies

3.1 Primary Testing Tools

3.1.1 Backend Testing

Tool	Purpose	Justification
pytest	Python unit testing	Industry standard, excellent Django integration, rich plugin ecosystem
pytest-django	Django-specific testing	Seamless Django ORM and settings integration
pytest-cov	Code coverage	Comprehensive coverage reporting with pytest integration
coverage	Coverage reporting	Detailed HTML reports, industry-standard metrics

3.1.2 Frontend Testing

Tool	Purpose	Justification
Vitest	Unit/Integration testing	Fast execution, TypeScript support, modern testing framework
React Testing Library	Component testing	Best practices for React testing, user-centric approach
Playwright	E2E testing	Cross-browser support, modern web app testing, reliable selectors
@testing-library/jest-dom	Custom matchers	Enhanced assertions for DOM testing

3.1.3 Quality Assurance Tools

Tool	Purpose	Justification

Tool	Purpose	Justification
Bandit	Security scanning	Python-specific security vulnerability detection
Safety	Dependency security	Known vulnerability database checking
Flake8	Code quality	PEP 8 compliance, code style enforcement
ESLint	JavaScript linting	Industry standard for JavaScript code quality

3.1.4 Performance Testing

Tool	Purpose	Justification
Locust	Load testing	Python-based, scalable, realistic user simulation

3.2 Tool Selection Justification

3.2.1 pytest vs unittest

Selected: pytest

- More readable test syntax
- Better fixture system
- Rich plugin ecosystem
- Superior error reporting
- Django integration through pytest-django

3.2.2 Playwright vs Selenium

Selected: Playwright

- Modern architecture with better stability
- Built-in waiting strategies
- Cross-browser testing capabilities
- Better debugging tools
- Active development and Microsoft backing

3.2.3 Vitest vs Jest

Selected: Vitest

- Native ES modules support
- Fast execution with Vite integration
- TypeScript support out of the box
- Modern testing features
- Better React 18+ compatibility

4. Testing Procedures

4.1 Test Development Process

4.1.1 Test-Driven Development (TDD)

1. Write failing test first
2. Implement minimum code to pass
3. Refactor code while maintaining tests
4. Repeat for each feature

4.1.2 Test Organization

```
Capstone-Unified/
├── core/
│   ├── tests/
│   │   ├── test_models.py
│   │   ├── test_services.py
│   │   ├── test_api.py
│   │   └── test_integration.py
│   └── trust_management/
│       └── tests/
│           └── test_models.py
├── frontend/crisp-react/
│   ├── src/
│   │   ├── components/
│   │   │   └── Component.test.jsx
│   │   └── test/
│   │       ├── integration/
│   │       └── e2e/
│   └── __tests__/
└── scripts/testing/
    ├── run-all-tests.sh
    └── run-quick-tests.sh
```

4.1.3 Test Naming Conventions

- **Unit Tests**: `test_<function_name>_<scenario>`
- **Integration Tests**: `test_<component>_integration_<scenario>`
- **E2E Tests**: `test_<user_story>_flow`

4.2 Test Execution Strategy

4.2.1 Local Development

```
# Backend unit tests
cd Capstone-Unified
python -m pytest core/tests/ -v

# Frontend unit tests
cd frontend/crisp-react
npm run test
```

```
# Integration tests
npm run test:integration

# Full test suite
npm run test:all
```

4.2.2 Pre-commit Testing

- Unit tests for changed modules
- Linting and code quality checks
- Security scans on modified files

4.2.3 Pull Request Testing

- Full unit test suite
- Integration tests
- Code coverage analysis
- Security vulnerability scanning

4.2.4 Pre-deployment Testing

- Complete test suite
- E2E testing
- Performance testing
- Security testing

5. Continuous Integration and Deployment

5.1 CI/CD Pipeline Tool: GitHub Actions

Tool Selection Justification:

- **Native GitHub Integration**: Seamless integration with our GitHub repository
- **No Additional Setup**: Built into GitHub, no external CI/CD service required
- **Cost Effective**: Free for public repositories, generous limits for private repos
- **Scalability**: Auto-scaling runners, parallel job execution
- **Ecosystem**: Rich marketplace of pre-built actions
- **Security**: Secure secrets management, isolated environments

5.2 Pipeline Architecture

5.2.1 Pipeline Stages

```
Trigger (Push/PR)
    ↓
Environment Setup
    ↓
```

```
Dependency Installation
    ↓
Code Quality Checks
    ↓
Unit Tests
    ↓
Integration Tests
    ↓
Security Scans
    ↓
Coverage Report
    ↓
Build Artifacts
    ↓
Deployment (if main branch)
```

5.2.2 Pipeline Configuration

File: `.github/workflows/ci-cd.yml`

Key features:

- **Multi-environment support**: Test with PostgreSQL service
- **Parallel execution**: Multiple test suites run concurrently
- **Artifact management**: Test reports and coverage uploaded
- **Security integration**: Bandit and Safety scans
- **Docker integration**: Automated image building and pushing

5.3 Pipeline Execution

5.3.1 Trigger Conditions

- **Push to main branch**: Full pipeline with deployment
- **Push to Dev branch**: Full pipeline without deployment
- **Pull requests**: Full pipeline for quality assurance
- **Manual trigger**: On-demand pipeline execution

5.3.2 Environment Configuration

```
Environment Variables:
- DJANGO_SETTINGS_MODULE: crisp_unified.settings
- SECRET_KEY: test-secret-key-for-ci-cd-pipeline
- DB_NAME: crisp_test
- DB_USER: postgres
- DB_PASSWORD: postgres
- DB_HOST: localhost
- DEBUG: False
- EMAIL_BACKEND: django.core.mail.backends.locmem.EmailBackend
```

5.3.3 Test Services

- **PostgreSQL 13**: Database service for integration tests
- **Python 3.9**: Consistent runtime environment
- **Node.js**: Frontend testing environment

5.4 Alternative CI/CD Tools Considered

5.4.1 Travis CI

Pros: Mature platform, good documentation **Cons**: Pricing changes, less GitHub integration, slower builds **Decision**: Not selected due to cost and performance concerns

5.4.2 Jenkins

Pros: High customization, self-hosted **Cons**: Maintenance overhead, setup complexity, infrastructure costs **Decision**: Not selected due to maintenance requirements

5.4.3 GitLab CI/CD

Pros: Comprehensive DevOps platform **Cons**: Would require repository migration, learning curve **Decision**: Not selected to maintain GitHub ecosystem

6. Test Coverage Requirements

6.1 Coverage Targets

6.1.1 Backend (Python/Django)

- **Minimum Overall Coverage**: 85%
- **Critical Components**: 95%
  - Authentication and authorization
  - Trust management
  - Data anonymization
  - Security services
- **Models**: 90%
- **API Endpoints**: 90%
- **Business Logic**: 95%

6.1.2 Frontend (React)

- **Minimum Overall Coverage**: 80%
- **Critical Components**: 90%
  - Authentication flows
  - Trust management UI
  - Data visualization
- **Utility Functions**: 95%
- **Components**: 85%

6.2 Coverage Reporting

6.2.1 Backend Coverage

```
# Generate coverage report
coverage run --source='.' manage.py test
coverage report --show-missing --skip-covered
coverage html

# Coverage configuration in .coveragerc
[run]
source = .
omit =
    */migrations/*
    */venv/*
    */tests/*
    manage.py
    */settings/*
```

6.2.2 Frontend Coverage

```
// vitest.config.js coverage configuration
export default defineConfig({
  test: {
    coverage: {
      provider: 'v8',
      reporter: ['text', 'json', 'html'],
      exclude: [
        'node_modules/',
        'src/test/',
        '**/*.{test,spec}.{js,jsx}',
        '**/test-utils.js'
      ],
      threshold: {
        global: {
          branches: 80,
          functions: 80,
          lines: 80,
          statements: 80
        }
      }
    }
  }
})
```

6.3 Coverage Quality Gates

- **PR Merge**: Minimum coverage maintained

- **Coverage Decline**: No more than 2% reduction allowed
- **New Code**: 90% coverage requirement for new features

7. Quality Gates

7.1 Code Quality Gates

7.1.1 Static Analysis

- **Flake8**: PEP 8 compliance, complexity analysis
- **ESLint**: JavaScript/React code quality
- **Bandit**: Security vulnerability detection
- **Safety**: Dependency vulnerability checking

7.1.2 Test Quality Gates

- All tests must pass
- Minimum coverage thresholds met
- No critical security vulnerabilities
- Performance benchmarks maintained

7.1.3 Pre-commit Hooks

```
# Performance check
npm run perf:check

# Linting
npm run lint

# Unit tests for modified files
pytest --co -q | grep modified_files
```

7.2 Deployment Gates

7.2.1 Main Branch Deployment

- All quality gates passed
- Security scans clear
- Integration tests successful
- Performance tests within limits

7.2.2 Release Deployment

- Full E2E test suite passed
- Load testing completed
- Security audit completed
- Documentation updated

8. Test Environment Management

8.1 Environment Types

8.1.1 Local Development

- **Purpose**: Developer testing and debugging
- **Data**: Minimal test datasets
- **Configuration**: Development settings
- **Isolation**: Per-developer instances

8.1.2 CI/CD Environment

- **Purpose**: Automated testing
- **Data**: Controlled test fixtures
- **Configuration**: Test-specific settings
- **Isolation**: Fresh environment per build

8.1.3 Staging Environment

- **Purpose**: Integration and E2E testing
- **Data**: Production-like datasets (anonymized)
- **Configuration**: Production-like settings
- **Isolation**: Shared, controlled access

8.2 Environment Configuration

8.2.1 Database Management

```
# Test settings
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'crisp_test',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

8.2.2 External Service Mocking

- **Email Services**: Django locmem backend
- **Third-party APIs**: Mock responses
- **File Storage**: In-memory or temporary storage

8.3 Environment Maintenance

8.3.1 Data Refresh

- **Frequency**: Weekly for staging
- **Process**: Automated data anonymization
- **Validation**: Data integrity checks

8.3.2 Configuration Management

- **Version Control**: All configurations in Git
- **Secrets Management**: GitHub Secrets for sensitive data
- **Environment Parity**: Maximum similarity between environments

9. Test Data Management

9.1 Test Data Strategy

9.1.1 Data Types

- **Fixtures**: Predefined test datasets
- **Factories**: Programmatically generated test data
- **Mock Data**: Simulated external service responses
- **Anonymized Production Data**: Sanitized real-world data

9.1.2 Data Generation

```
# Django fixtures example
from django.test import TestCase
from core.models import Organization, TrustRelationship

class TrustRelationshipTestCase(TestCase):
    fixtures = ['organizations.json', 'trust_levels.json']

    def setUp(self):
        self.org1 = Organization.objects.get(name='Test Org 1')
        self.org2 = Organization.objects.get(name='Test Org 2')
```

9.2 Data Management Practices

9.2.1 Data Isolation

- **Test Independence**: Each test creates/cleans its own data
- **Transaction Rollback**: Automatic cleanup after each test
- **Unique Identifiers**: UUIDs to prevent conflicts

9.2.2 Data Privacy

- **No Production Data**: Never use real user data in tests
- **Anonymization**: All test data must be synthetic or anonymized

- **Compliance**: GDPR and cybersecurity regulations adherence

9.3 Test Data Lifecycle

9.3.1 Creation

```
# Factory Boy example
import factory
from core.models import Organization

class OrganizationFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Organization

    name = factory.Sequence(lambda n: f"Test Organization {n}")
    organization_type = 'PRIVATE'
    contact_email = factory.LazyAttribute(lambda obj:
f'{obj.name.lower().replace(" ", "")}@example.com')
```

9.3.2 Maintenance

- **Regular Updates**: Keep test data current with schema changes
- **Performance**: Monitor test data size and execution impact
- **Cleanup**: Automated removal of obsolete test data

10. Reporting and Metrics

10.1 Test Reporting

10.1.1 Automated Reports

- **Test Results**: Pass/fail status with detailed logs
- **Coverage Reports**: HTML and JSON formats
- **Performance Metrics**: Execution time trends
- **Security Scan Results**: Vulnerability reports

10.1.2 Report Storage

- **GitHub Actions Artifacts**: Test reports and coverage
- **GitHub Pages**: Published coverage reports
- **Local Storage**: Developer machines for debugging

10.2 Key Metrics

10.2.1 Quality Metrics

- **Test Coverage**: Percentage of code covered by tests
- **Test Pass Rate**: Percentage of tests passing

- **Defect Density**: Bugs per lines of code
- **Mean Time to Detection**: Average time to find bugs

10.2.2 Performance Metrics

- **Test Execution Time**: Time to run full test suite
- **Build Time**: CI/CD pipeline execution time
- **Code Quality Score**: Static analysis results
- **Security Score**: Vulnerability assessment results

10.3 Reporting Schedule

10.3.1 Continuous

- **CI/CD Pipeline Results**: Every build
- **Code Coverage**: Every commit
- **Security Scans**: Every build

10.3.2 Weekly

- **Test Trend Analysis**: Coverage and performance trends
- **Quality Metrics Review**: Team quality review
- **Performance Benchmarks**: System performance assessment

11. Responsibilities

11.1 Development Team

11.1.1 Individual Developers

- Write unit tests for all new code
- Maintain existing tests when modifying code
- Run tests locally before committing
- Fix failing tests immediately
- Maintain minimum coverage requirements

11.1.2 Senior Developers/Tech Leads

- Review test coverage in code reviews
- Ensure test quality and best practices
- Mentor junior developers on testing
- Make architectural testing decisions

11.2 Quality Assurance

11.2.1 QA Engineers

- Design and implement integration tests
- Create and maintain E2E test suites

- Perform manual testing for complex scenarios
- Validate test environment configurations

11.2.2 Security Team

- Review security test implementations
- Define security testing requirements
- Validate security scan configurations
- Assess vulnerability reports

11.3 DevOps/Infrastructure

11.3.1 DevOps Engineers

- Maintain CI/CD pipeline
- Manage test environments
- Monitor pipeline performance
- Ensure infrastructure reliability

11.4 Project Management

11.4.1 Project Managers

- Ensure testing activities are scheduled
- Monitor testing progress and metrics
- Allocate resources for testing activities
- Report on quality metrics to stakeholders

12. References

12.1 Documentation Links

12.1.1 Repository Links

- **Main Repository**: https://github.com/COS301-SE-2025/CRISP
- **Test Cases Directory**: `/Capstone-Unified/core/tests/`
- **Frontend Tests**: `/frontend/crisp-react/src/test/`
- **CI/CD Configuration**: `/.github/workflows/ci-cd.yml`
- **Testing Scripts**: `/scripts/testing/`

12.1.2 Test Reports Location

- **Coverage Reports**: Generated in CI/CD artifacts
- **Security Reports**: `bandit-report.json` in pipeline artifacts
- **Performance Reports**: Locust HTML reports
- **E2E Test Reports**: Playwright HTML reports

12.2 External References

12.2.1 Testing Frameworks

- **pytest Documentation**: https://docs.pytest.org/
- **Vitest Documentation**: https://vitest.dev/
- **Playwright Documentation**: https://playwright.dev/
- **React Testing Library**: https://testing-library.com/docs/react-testing-library/intro/

12.2.2 Best Practices

- **Martin Fowler's Testing**: https://martinfowler.com/testing/
- **Google Testing Blog**: https://testing.googleblog.com/
- **Django Testing Documentation**: https://docs.djangoproject.com/en/stable/topics/testing/

12.3 Standards and Compliance

12.3.1 Industry Standards

- **ISO/IEC 29119**: Software Testing Standards
- **NIST Cybersecurity Framework**: Security testing guidelines
- **OWASP Testing Guide**: Web application security testing

12.3.2 Internal Standards

- **Code Review Guidelines**: Internal development standards
- **Security Policies**: Organizational security requirements
- **Quality Assurance Procedures**: Company QA standards

12.4 Tool Documentation

12.4.1 CI/CD Tools

- **GitHub Actions**: https://docs.github.com/en/actions
- **Docker**: https://docs.docker.com/
- **PostgreSQL**: https://www.postgresql.org/docs/

12.4.2 Testing Tools

- **Bandit Security**: https://bandit.readthedocs.io/
- **Coverage.py**: https://coverage.readthedocs.io/
- **Locust**: https://locust.io/
- **ESLint**: https://eslint.org/docs/
