README

CRISP - External TAXII 2.1 Threat Intelligence Feed Consumption

Overview

This Django application focuses on consuming external TAXII 2.1 compliant threat intelligence feeds for the CRISP (Cyber Risk Information Sharing Platform) system. The component enables organizations to automatically retrieve, process, and store standardized threat intelligence from external sources such as AlienVault OTX.

The feed consumption component addresses a critical need in cybersecurity: staying updated with the latest threat intelligence. By automating the consumption of standardized TAXII feeds, organizations can maintain awareness of emerging threats without manual intervention.

What is TAXII?

TAXII (Trusted Automated Exchange of Intelligence Information) is a protocol designed specifically for the communication of cyber threat intelligence (CTI) between systems. TAXII 2.1, which our application consumes, is an HTTPS RESTful API that uses JSON for data formatting and allows systems to exchange STIX content.

Key characteristics of TAXII 2.1:

- RESTful API using JSON payloads
- Supports HTTPS for transport security
- Provides discovery, collection, and API root services
- Enables polling for new intelligence
- Supports filtering of data during exchange
- Uses basic authentication (username/password)

What are STIX Objects?

STIX (Structured Threat Information Expression) is a standardized language for representing cyber threat intelligence. Our feed consumption component processes these structured objects from external sources.

The STIX objects we primarily handle include:

Indicators

Technical patterns that can be used to detect suspicious or malicious activity, such as:

- IP addresses (IPv4, IPv6)
- Domain names
- URLs
- File hashes (MD5, SHA-1, SHA-256)
- Email addresses
- Registry keys

Attack Patterns (TTPs)

Tactics, Techniques, and Procedures used by threat actors, including:

- MITRE ATT&CK framework techniques
- Custom attack methodologies
- Malware behaviors

How Feed Consumption Works

The feed consumption process follows these key steps:

1. **Connection Establishment**: The application connects to an external TAXII server using authentication credentials
2. **Collection Discovery**: The application identifies available collections on the TAXII server
3. **Polling**: The application polls a specified collection for content blocks
4. **STIX Processing**: The received content blocks are parsed and converted to STIX objects
5. **Entity Conversion**: STIX objects are converted to CRISP domain entities
6. **Deduplication**: The system checks for existing entities to avoid duplicates
7. **Storage**: New entities are stored in the database
8. **Notification**: Observers are notified of new threat intelligence

Architecture Components for Feed Consumption

Service Layer

StixTaxiiService

Base service that provides common functionality for TAXII operations:

- Establishing TAXII connections

- Discovering collections
- Parsing STIX content

OTXTaxiiService

Specialized service for AlienVault OTX TAXII feeds:

- Handles OTX-specific authentication
- Processes OTX STIX format
- Manages batched consumption for memory efficiency

Repository Layer

IndicatorRepository

Handles storage and retrieval of threat indicators:

- Creating new indicators
- Updating existing indicators
- Checking for duplicates

TTPRepository

Handles storage and retrieval of TTPs:

- Creating new TTP records
- Updating existing TTPs
- Checking for duplicates

Parser Component

STIX1Parser

Parses STIX 1.x XML content:

- Extracts indicator information
- Extracts TTP information
- Handles parsing errors gracefully

Design Patterns Used

Factory Pattern

The `StixObjectCreator` hierarchy standardizes the creation of STIX objects:

- `StixIndicatorCreator` : Creates STIX indicators from raw data
- `StixTTPCreator` : Creates STIX attack patterns from raw data

Observer Pattern

The `ThreatFeed` as a subject notifies observers when new intelligence is consumed:

- `InstitutionObserver` : Notifies institutions about feed updates
- `AlertSystemObserver` : Triggers alerts based on feed updates

Prerequisites

- Python 3.9+
- PostgreSQL 13+
- Redis 6+
- RabbitMQ (for Celery)

Environment Setup

1. **Clone the repository**

```
git clone <repository-url>

cd CRISP
```

2. **Create and activate a virtual environment**

```
python -m venv venv

source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install dependencies**

```
pip install -r requirements.txt
```

4. **Create a .env file**

Create a `.env` file in the project root with the following variables:

```
# Database settings

DB_NAME=crisp

DB_USER=myuser  # Replace with your PostgreSQL user

DB_PASSWORD=your_password  # Replace with your PostgreSQL password

DB_HOST=localhost

DB_PORT=5432



# Django settings

DJANGO_SECRET_KEY=your_django_secret_key  # Generate with: python -c "from django.core.management.utils import get_random_secret_key; print(get_random_secret_key())"

DEBUG=True



# OTX API Key (for AlienVault OTX integration)

OTX_API_KEY=your_otx_api_key  # Get from otx.alienvault.com



# Redis settings

REDIS_URL=redis://localhost:6379/0
```

5. **Set up the PostgreSQL database**

```
sudo -u postgres psql

CREATE USER myuser WITH PASSWORD 'your_password';

CREATE DATABASE crisp;

GRANT ALL PRIVILEGES ON DATABASE crisp TO myuser;

\q
```

6. **Apply migrations**

```
python manage.py makemigrations core

python manage.py migrate
```

7. **Create a superuser**

```
python manage.py createsuperuser
```

Running the Application

Start the Django Development Server

```
python manage.py runserver
```

The server will be available at http://127.0.0.1:8000/

Start the Celery Worker (for asynchronous feed consumption)

```
# Make sure Redis is running
```

```
redis-cli ping  # Should return PONG



# Start Celery worker

celery -A crisp worker -l info
```

Consuming Threat Intelligence Feeds

Setting Up a Feed through the Admin Interface

1. **Create an Institution**

 - Navigate to http://127.0.0.1:8000/admin/

 - Go to "Institutions" and click "Add Institution"

 - Fill in the details and save

2. **Create a Threat Feed**

 - Go to "Threat feeds" and click "Add Threat feed"

 - Fill in the details:

   - Name: "AlienVault OTX Feed"

   - Description: "External threat feed from AlienVault OTX"

   - Owner: Select your institution

   - Is External: Check this box

   - TAXII Server URL: "https://otx.alienvault.com/taxii"

   - TAXII API Root: "taxii"

   - TAXII Collection ID: "user_AlienVault"

   - TAXII Username: Your OTX API key

   - Is Public: Check if you want to share this feed

 - Save the threat feed

Consuming the Feed via API

Start Consumption

Use the API endpoint to start the consumption process:

```
# Process in batches of 100 with data from the last 7 days

curl -X POST "http://127.0.0.1:8000/api/threat-feeds/1/consume/?
force_days=7&batch_size=100"
```

Advanced Options

```
# Process in batches of 100 with data from the last 30 days

curl -X POST "http://127.0.0.1:8000/api/threat-feeds/1/consume/?
force_days=30&batch_size=100"



# Process in the background (async mode)

curl -X POST "http://127.0.0.1:8000/api/threat-feeds/1/consume/?
force_days=7&batch_size=100&async=true"



# Limit the number of blocks to process (for testing)

curl -X POST "http://127.0.0.1:8000/api/threat-feeds/1/consume/?limit=10"
```

Check Feed Status

```
curl -X GET "http://127.0.0.1:8000/api/threat-feeds/1/status/"
```

This will show the number of indicators and TTPs, along with the last sync time.

Data Flow: Threat Intelligence Consumption

1. The system connects to external TAXII feeds (such as AlienVault OTX)
2. It retrieves STIX-formatted threat intelligence data (indicators and TTPs)
3. The data is processed in batches to manage memory usage
4. Each item is parsed, converted to CRISP entities, and stored in the database
5. The system avoids duplicates by checking STIX IDs
6. Processing statistics are tracked and reported

Testing the Feed Consumption

The test suite for the TAXII feed consumption covers multiple aspects:

1. **Mock Data** - Simulated STIX/TAXII data for testing
2. **Unit Tests** - Individual component testing
3. **Integration Tests** - Testing components working together
4. **End-to-End Tests** - Testing the complete consumption process

Test Files Structure

Mock Data

- `test_stix_mock_data.py` : Contains mock STIX 1.x (XML), STIX 2.0, and STIX 2.1 (JSON) data for testing

Component Tests

- `test_stix1_parser.py` : Tests for the STIX 1.x parser that processes XML content from TAXII feeds
- `test_taxii_service.py` : Tests for the TAXII service implementations (OTXTaxiiService and StixTaxiiService)
- `test_stix_factory.py` : Tests for the Factory pattern implementation (StixIndicatorCreator and StixTTPCreator)

Integration and End-to-End Tests

- `test_taxii_integration.py` : Integration tests that verify the correct interaction between different components
- `test_end_to_end.py` : End-to-end tests that simulate the complete TAXII feed consumption process

Running Tests

Basic Testing Commands

```
# Run all tests

python manage.py test core --settings=crisp.test_settings



# Run tests with verbose output

python manage.py test core --settings=crisp.test_settings -v 2
```

Running Specific Test Files

```
# Test STIX parser

python manage.py test core.tests.test_stix1_parser --
settings=crisp.test_settings



# Test TAXII service

python manage.py test core.tests.test_taxii_service --
settings=crisp.test_settings



# Test integration

python manage.py test core.tests.test_taxii_integration --
settings=crisp.test_settings



# Test end-to-end

python manage.py test core.tests.test_end_to_end --
```

```
settings=crisp.test_settings
```

Test Coverage

```
# Install coverage package

pip install coverage



# Run tests with coverage

coverage run --source='.' manage.py test core --settings=crisp.test_settings



# Display basic coverage report

coverage report



# Generate detailed HTML coverage report

coverage html
```

Current Test Coverage

The current test coverage for the TAXII feed consumption is approximately 91%, with all 78 tests passing successfully. This indicates a robust and well-tested implementation.

Key components with high coverage:

- Repository layer: 100%
- Factory pattern implementations: >84%
- Observer pattern implementation: 97%

Industry-Standard Implementation

This TAXII feed consumption implementation follows established industry standards and practices used by major threat intelligence platforms. The following elements align with industry best practices:

Structured Processing Pipeline

The implementation follows the standard industry flow for threat intelligence consumption:

1. **Connection Establishment**: Securely connect to TAXII server with appropriate authentication
2. **Collection Discovery**: Identify and select appropriate intelligence collections
3. **Content Polling**: Retrieve content blocks using standardized TAXII polling
4. **STIX Processing**: Parse and validate standardized STIX data
5. **Entity Conversion**: Transform external STIX objects to internal domain entities
6. **Deduplication**: Check for and handle duplicate intelligence items
7. **Storage**: Persist threat intelligence in structured database
8. **Notification**: Alert relevant systems about new intelligence

Other Industry-Standard Elements

1. **STIX/TAXII Compatibility**: Full support for the STIX format and TAXII 2.1 protocol, aligning with platforms like MISP, OpenCTI, and ThreatConnect
2. **Automated Collection**: Scheduled and on-demand polling of external feeds, similar to commercial threat intelligence platforms
3. **Batch Processing**: Efficient handling of large volumes of threat data through batched processing, a common approach in production systems
4. **Standardized Entity Types**: Proper processing of industry-standard entity types including indicators (IPs, domains, hashes) and TTPs (attack patterns)
5. **Authentication Management**: Secure handling of API keys and credentials for accessing external feeds
6. **Error Handling**: Robust error handling for network issues, malformed data, and service disruptions
7. **API-First Design**: RESTful API endpoints for programmatic consumption control, allowing integration with other security tools

Project Structure

```
crisp/

├── manage.py                                # Django management script
├── requirements.txt                # Project dependencies
├── .env                            # Environment variables
├── crisp/
│   ├── __init__.py
│   ├── settings.py                 # Django settings
│   ├── urls.py                     # URL configuration
│   ├── celery.py                   # Celery configuration
│   └── wsgi.py
├── core/
│   ├── __init__.py
│   ├── admin.py                    # Admin panel configuration
│   ├── apps.py
│   ├── models/                     # Domain models
│   │   ├── __init__.py
│   │   ├── indicator.py            # Indicator model
│   │   ├── institution.py          # Institution model
│   │   └── ttp_data.py             # TTP data model
│   ├── patterns/                   # Design pattern implementations
│   │   ├── observer/               # Observer pattern
│   │   │   ├── __init__.py
│   │   │   └── threat_feed.py        # ThreatFeed as Subject
│   │   ├── factory/                  # Factory pattern
│   │   │   ├── __init__.py
│   │   │   ├── stix_object_creator.py
│   │   │   ├── stix_indicator_creator.py
│   │   │   └── stix_ttp_creator.py
│   │   └── decorator/                # Decorator pattern
│   │       ├── __init__.py
│   │       ├── stix_object_component.py
│   │       └── stix_decorator.py
│   ├── repositories/                 # Repositories layer
│   │   ├── __init__.py
│   │   ├── threat_feed_repository.py
│   │   ├── indicator_repository.py
│   │   └── ttp_repository.py
│   ├── services/                     # Services layer
│   │   ├── __init__.py
│   │   ├── otx_taxii_service.py      # OTX TAXII service
│   │   ├── stix_taxii_service.py     # Generic STIX/TAXII service
│   │   └── threat_feed_service.py
│   ├── parsers/                      # Data parsers
│   │   ├── __init__.py
│   │   └── stix1_parser.py           # STIX 1.x parser
│   ├── tasks/                        # Celery tasks
│   │   ├── __init__.py
│   │   └── taxii_tasks.py            # TAXII-related tasks
│   └── views/                        # Django views
│       ├── __init__.py
│       ├── home.py                   # Homepage view
│       └── api/                      # API views
│           ├── __init__.py
│           └── threat_feed_views.py   # ThreatFeed API
```

Conclusion

The TAXII feed consumption component provides a robust solution for automatically retrieving, processing, and storing standardized threat intelligence. By leveraging industry-standard protocols (STIX/TAXII) and following established processing workflows, the implementation ensures compatibility with a wide range of threat intelligence sources and aligns with industry-leading platforms.

The batched processing approach allows for efficient handling of large volumes of threat data, while the comprehensive test suite ensures reliability and correctness. With a current test coverage of 91%, the implementation demonstrates a strong commitment to quality and robustness.