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.

Initial Information Model Diagram

Figure: VXDF Information Model. The conceptual model shows how a **Flow** entity relates to its key parts: **Source**, **Sink**, and **Evidence**. Each Flow represents one validated vulnerability data path, with a Source (where untrusted data enters), and a Sink (where that data leads to a security impact). A Flow may include an ordered sequence of intermediate steps (propagation points) from Source to Sink, depicting how data moves through the code. Each Flow is accompanied by Evidence that validates the exploitability of that path (such as a proof-of-concept or test result). This evidence "anchors" the flow, indicating that the path is not just theoretical but has been confirmed to cause a security breach.

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: The primary goals for VXDF v1.0 include:

- *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**, **SHOULD**, **SHOULD NOT**, and **MAY** in this document are to be interpreted as requirement levels, in line with RFC 2119 conventions. These indicate mandatory or optional aspects of the specification:

- **MUST** / **MUST NOT**: A requirement that is absolutely mandatory (or prohibited) for compliance. All conforming VXDF documents and implementations are expected to follow these strictly.

- **SHOULD** / **SHOULD NOT**: A strong recommendation. There may exist valid reasons in particular circumstances to deviate from these, but the full implications must be understood and carefully weighed before doing so.

- **MAY**: Truly optional or discretionary. The item is permitted but not required, and implementers have flexibility.

Other terminology in this spec:

- **VXDF Document (or VXDF File)**: A JSON document that conforms to the VXDF schema and rules, containing one or more vulnerability data flows with validation evidence. This is the primary interchange artifact defined by this specification.

- **Flow**: A single *validated exploitable data flow*. This represents one vulnerability instance. It typically consists of a Source, a Sink, and possibly intermediate steps, describing how data travels from the Source to the Sink. A Flow in VXDF also includes metadata like severity and category, and it **MUST** include at least one piece of Evidence proving its exploitability.

- **Source**: The starting point of a data flow, where untrusted or external input enters the system. In code terms, this could be a function that reads user input, a parameter from an HTTP request, etc. The Source is characterized by a code location (file, line, etc.) and context. In VXDF, *Source* refers specifically to the location and nature of the input that can be controlled by an attacker.

- **Sink**: The end point of a data flow where the untrusted data is used in a potentially dangerous way, triggering the vulnerability. This is usually a sensitive operation or API call (e.g., executing a SQL query, writing to a system command, rendering output to a web page) that, when fed with malicious input, results in an exploit. In VXDF, *Sink* is also described by a code location and indicates the point in code that needs fixing (often where validation or sanitization is missing).

- **Step**: An intermediate step in the data flow from Source to Sink. Steps describe the progression of data through the code, for example passing through functions, stored in variables, or modified along the way. VXDF can include a sequence of Steps to provide a trace. Each Step has a code location and optionally a note explaining what happens at that point. The first step could be the Source and the last the Sink, or those can be explicitly identified separately; in any case, the ordered Steps **SHOULD** form a logical path from the Source to the Sink.

- **Evidence**: Information that validates the exploitability of the flow. Evidence may include a proof-of-concept input, an execution trace, a screenshot, or a simple textual explanation of how the exploit was confirmed. In VXDF, Evidence is typically a structured object (or several) that provides details like what method was used to validate (e.g., manual test, automated fuzz), and a description of the outcome. At least one piece of evidence is required for each flow. This is central to VXDF's purpose: the presence of Evidence distinguishes a VXDF flow from an unverified static finding.

- **Category**: A high-level classification of the vulnerability type (e.g., "SQL Injection", "Cross-Site Scripting"). This gives a quick idea of what kind of issue the flow represents. VXDF provides a field for Category to aid human understanding and filtering of flows.

- **CWE**: Stands for Common Weakness Enumeration, a standardized identifier for types of software weaknesses (e.g., CWE-89 for SQL Injection). VXDF flows can optionally reference a CWE ID to unambiguously identify the general class of vulnerability. This can help when aggregating or reporting metrics, as CWE is widely recognized.

- **Severity**: The criticality or risk level of the vulnerability. VXDF uses qualitative severity levels (Critical, High, Medium, Low, Informational) to indicate how serious the issue is. This helps prioritize flows. "Critical" typically means the vulnerability could be easily exploited with severe impact (e.g., remote code execution), whereas "Low" might mean a minor information leak or a difficult-to-exploit edge case. The spec defines these categories, and producers **SHOULD** choose the one that best matches the impact, possibly informed by CVSS or internal rating.

- **Metadata**: Ancillary information about a VXDF document, such as who generated it, when, and what it pertains to. Metadata in VXDF is not about a particular flow, but about the whole document (for example, the tool name that produced it, a title for the report, or the target system name).

- **Extension Field**: Any JSON field within a VXDF document that is not part of this specification's core schema, but added via the extension mechanism (usually by prefixing the field name with `x-`). Extension fields **MAY** be used by vendors or specific communities to include additional information (for instance, `x-riskScore`). These fields are not standardized in VXDF and might not be understood by generic tools, but conforming processors **MUST** ignore (tolerate) unknown `x-` fields so long as the document is otherwise valid.

VXDF Document Structure

A VXDF document is a JSON object composed of two main sections: a **header** (with metadata and version information) and a **list of flows**. In broad terms, it looks like this:

- **VXDF Version**: An explicit version indicator (e.g., `"vxdfVersion": "1.0"`). This allows the format to evolve. Parsers use it to ensure compatibility. This field is required at the top level. If an implementation encounters a vxdfVersion it doesn't support, it **SHOULD** reject or handle the document cautiously.

- **Metadata** *(optional)*: A JSON object containing metadata about the document. This can include fields like:

    - `generator`: The name/version of the tool or person that generated the VXDF file (e.g., `"MySecurityScanner 3.2"` or `"Jane Researcher v1"`). This helps track the origin of the report.

    - `timestamp`: When the document was generated, in ISO 8601 date-time format (UTC). Useful for knowing recency of the info.

    - `target`: An identifier for the application, system, or component that was analyzed. For example, this might be a software name and version (`"Acme WebApp 2.1"`), a repository URL, or a unique identifier like a package name. This ties the findings to a specific scope.

    - Additional metadata fields can be added as needed (via extensions or future versions) – e.g., an overall risk score, environment details, etc.

- **Flows**: An array of flow objects, each representing one validated vulnerability data flow. This is the core of the document. Each flow in the array includes:

    - **id**: A unique identifier for the flow (unique within the document, and ideally stable across merges or reruns if the same issue appears). This could be a simple number or a string. It allows referencing a specific flow (for example, in discussions or when mapping to external systems like bug IDs).

    - **title**: A short human-readable title for the vulnerability. E.g., `"SQL Injection in login form"`. The title summarizes the issue, helping humans quickly scan a list of flows.

    - **description**: A longer explanation of the vulnerability and its implications. This may describe what the flow means in context, the potential impact, or any relevant technical detail. (This field is optional but recommended; if omitted, the evidence and the nature of the flow should speak for themselves.)

    - **severity**: The severity level of this issue, as one of a fixed set of strings (Critical, High, Medium, Low, Informational). This helps prioritize issues. Tools might assign this based on CVSS scoring or company policy, but the result is encoded

in these standard labels for consistency.

- **category**: A brief classification of the vulnerability type, e.g., `"SQL Injection"`, `"Cross-Site Scripting"`, `"Path Traversal"`. This gives immediate context about the nature of the bug. While not strictly required (some might rely purely on CWE or description), providing a category is useful for grouping and filtering issues.

- **cwe**: An optional field to specify a CWE identifier (e.g., `"CWE-79"` for XSS). When provided, it removes ambiguity about what kind of weakness is being exploited. It's especially helpful for reporting and cross-tool correlation (since CWE IDs are common reference points in security).

- **source**: An object describing the Source of the flow – typically the location in code where untrusted input enters. This object includes at minimum a `file` path (or other identifier of the code artifact) and a `line` number. It can also include a `function` name (if applicable) and a snippet of code or relevant text at that location. For instance, the source might be `{ "file": "src/Login.java", "line": 42, "function": "doPost", "snippet": "username = request.getParameter(\"user\");" }`. The snippet is optional but provides quick context (the spec does not require it because not all producers may have access to source code text).

- **sink**: An object describing the Sink of the flow – the code location where the vulnerability actually manifests. It has the same structure as source (file, line, etc.). For example, a sink might be `{ "file": "src/Database.java", "line": 87, "function": "executeQuery", "snippet": "stmt.execute(\"SELECT * FROM users WHERE user='\" + username + \"'\");" }`. The sink is often where a fix needs to be applied (e.g., adding input sanitization or using a safer API).

- **steps**: *Optional.* An array of step objects outlining the data flow path from the source to the sink. Each step is an object similar to source/sink (with file, line, function, snippet) and may include a `note` to clarify what's happening at that step. The steps are ordered in the exact sequence that data passes through. For example, steps might show that the data flows from the controller to a utility function to the database layer. If provided, the first step could repeat the source location and the last step the sink, or the steps could strictly be intermediate points with the understanding that source and sink are the endpoints. In any case, the presence of a steps array helps someone trace the vulnerability through multiple layers of the code. If a flow is direct or the path is obvious, steps can be omitted or kept minimal. (In the JSON schema below, steps are optional

and can be empty, but typically 0 means a direct source-to-sink with no notable stops.)

- **evidence**: An array of evidence objects supporting this flow's exploitability. **At least one evidence entry is required** for every flow. Each evidence object can have:
  - `description`: A human-readable description of the evidence. This might describe what input was used and what outcome was observed. E.g., `"Using the payload ' OR '1'='1 in the username field, we were able to log in as an existing user without a password."`
  - `method`: (Optional) A short tag indicating how the evidence was obtained. For example, `"manual test"`, `"automated fuzzing"`, `"dynamic analysis"`, `"unit test"`. This gives context on whether the exploit was confirmed manually or by a tool, etc.
  - `timestamp`: (Optional) When the evidence was gathered or the test performed, in ISO 8601 format. This can be useful in case evidence may depend on time (for instance, if a proof was captured on a certain date, or to indicate freshness).
  - (Additional fields could be added via extension, e.g., a reference to an external proof-of-concept file, but typically a descriptive text is enough.)

- The evidence essentially answers: *"How do we know this flow is exploitable?"* It might include an input string, expected vs. actual behavior, or references to logs. The presence of evidence is what "validates" the data flow.

- **Extensions**: Any field that is not recognized by this specification (i.e., not one of the above) **MUST** use a prefix of `x-` in its name. For instance, an organization might add `x-riskScore: 9` or `x-internalId: "VULN-1234"`. Such fields are considered non-normative and are for tool/organization-specific use. Conforming VXDF processors **SHOULD** ignore these fields if they don't know them (but may preserve them when passing data through). This allows flexibility without breaking interoperability. The use of `x-` prefix avoids naming collisions with future official fields.

The VXDF JSON Schema (next section) formalizes this structure. In summary, a valid VXDF document must include the version, at least one flow (with required subfields including evidence), and follow the specified JSON structure. The design ensures that everything needed

to understand and reproduce the vulnerability is present or referenced, linking every field back to a user pain point:

- By reading source, sink, and steps, a developer knows *where* to look in code.
- By reading severity and category/CWE, they know *how important* it is and what kind of flaw it is.
- By reading evidence, they know *the exploit scenario* and can even try it themselves to confirm or after a fix to ensure it's resolved.
- The format being consistent means less guesswork in parsing the information, whether by humans or tools.

JSON Schema (Normative)

The following is the normative JSON Schema definition of VXDF v1.0. 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.)

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema#",
  "$id": "https://example.com/vxdf-1.0.schema.json",
  "title": "Validated Exploitable Data Flow (VXDF) Schema",
  "description": "JSON Schema for VXDF format version 1.0",
  "type": "object",
  "properties": {
    "vxdfVersion": {
      "type": "string",
      "const": "1.0",
      "description": "The VXDF format version. Must be '1.0' for this specification version."
    },
    "metadata": {
      "type": "object",
      "description": "Document metadata such as generator and context.",
      "properties": {
        "generator": {
          "type": "string",
          "description": "Name and version of the tool or author that generated the VXDF file."
        },
        "timestamp": {
          "type": "string",
```

```
        "format": "date-time",
        "description": "Time when the VXDF document was generated, in ISO 8601 format."
       },
       "target": {
        "type": "string",
        "description": "Optional identifier for the target application or component that was
analyzed."
       }
     },
     "required": [],
     "additionalProperties": false,
     "patternProperties": {
      "^x-": {}
     }
    },
    "flows": {
     "type": "array",
     "minItems": 1,
     "description": "List of validated exploitable data flows (vulnerabilities) identified.",
     "items": {
      "$ref": "#/$defs/flow"
     }
    }
  },
  "required": [
   "vxdfVersion",
   "flows"
  ],
  "additionalProperties": false,
  "patternProperties": {
   "^x-": {}
  },
  "$defs": {
   "flow": {
    "type": "object",
    "properties": {
     "id": {
      "type": "string",
      "description": "Unique identifier for this flow within the document."
     },
     "title": {
      "type": "string",
      "description": "Short summary of the vulnerability or data flow."
     },
```

```
        "description": {
          "type": "string",
          "description": "Detailed description of the flow and its security impact."
        },
        "severity": {
          "type": "string",
          "description": "Severity rating of the issue.",
          "enum": [
            "Critical",
            "High",
            "Medium",
            "Low",
            "Informational"
          ]
        },
        "category": {
          "type": "string",
          "description": "High-level category of the vulnerability (e.g., \"SQL Injection\")."
        },
        "cwe": {
          "type": "string",
          "description": "Common Weakness Enumeration ID (e.g., \"CWE-89\") relevant to this
vulnerability."
        },
        "source": {
          "$ref": "#/$defs/codeLocation",
          "description": "Location and context of the source (entry point of untrusted data)."
        },
        "sink": {
          "$ref": "#/$defs/codeLocation",
          "description": "Location and context of the sink (dangerous usage of the data)."
        },
        "steps": {
          "type": "array",
          "description": "Ordered steps in the data flow from source to sink, including intermediate
propagation points.",
          "items": {
            "type": "object",
            "properties": {
              "file": {
                "type": "string"
              },
              "line": {
                "type": "integer"
```

```
      },
      "function": {
        "type": "string"
      },
      "snippet": {
        "type": "string"
      },
      "note": {
        "type": "string"
      }
    },
    "required": [
      "file",
      "line"
    ],
    "additionalProperties": false,
    "patternProperties": {
      "^x-": {}
    }
  }
},
"evidence": {
  "type": "array",
  "minItems": 1,
  "description": "Evidence supporting the exploitability of this flow.",
  "items": {
    "type": "object",
    "properties": {
      "description": {
        "type": "string"
      },
      "method": {
        "type": "string"
      },
      "timestamp": {
        "type": "string",
        "format": "date-time"
      }
    },
    "required": [
      "description"
    ],
    "additionalProperties": false,
    "patternProperties": {
```

```
          "^x-": {}
        }
       }
     }
    },
    "required": [
     "id",
     "title",
     "severity",
     "source",
     "sink",
     "evidence"
    ],
    "additionalProperties": false,
    "patternProperties": {
     "^x-": {}
    }
   },
   "codeLocation": {
    "type": "object",
    "properties": {
     "file": {
      "type": "string"
     },
     "line": {
      "type": "integer"
     },
     "column": {
      "type": "integer"
     },
     "function": {
      "type": "string"
     },
     "snippet": {
      "type": "string"
     }
    },
    "required": [
     "file",
     "line"
    ],
    "additionalProperties": false,
    "patternProperties": {
     "^x-": {}
```

```
    }
   }
  }
 }
}
```

Notes on the schema: This schema enforces all the rules described in the prose. Notably, it requires `vxdfVersion` to be `"1.0"`, at least one flow in `flows`, and each flow to have the key fields (id, title, severity, source, sink, evidence). It also restricts severity to the allowed set of values, and it disallows any properties not defined (except those beginning with `x-` which are permitted by `patternProperties` as extensions). All string fields are generally free-form text (with some recommendations or formats like `timestamp` using date-time). The `codeLocation` sub-schema applies to source and sink (and by reference to each step item), ensuring each location has at least a file and line number. This means producers must provide file and line for source/sink – which is typically available in code analysis contexts. If a vulnerability is not tied to a specific file (for example, an environment misconfiguration), the producer might use a placeholder or general description in the snippet or file field; however, such uses are expected to be rare for this format which is oriented around code flows.

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.

Conformance & Validation

This section describes what it means to conform to the VXDF specification, both for documents and for tools that produce/consume them.

VXDF Document Conformance: A conformant VXDF document **MUST** meet all requirements of the JSON schema above and the rules stated in this specification:

- It **MUST** be a valid JSON file encoded in UTF-8 (the standard JSON encoding).
- It **MUST** contain a top-level object with at least `vxdfVersion` and `flows` fields. The `vxdfVersion` must be a supported version string (for this spec, exactly `"1.0"`).
- It **MUST** include at least one flow in the `flows` array. An empty `flows` list is not considered a valid VXDF output (if there were no findings, typically no VXDF file is produced, or a special indication outside this spec would be used).
- Each flow in the `flows` array **MUST** have all required sub-fields as per schema: `id`, `title`, `severity`, `source`, `sink`, and `evidence`. If any of these is missing or null, the document is invalid. In particular, flows without evidence are not allowed – this is by design to enforce that every flow is verified.

- All string fields like `file`, `function`, `description`, etc., **MUST NOT** contain control characters or content that would break JSON encoding (this is generally ensured by JSON encoding itself). They should be plain text. If a field needs to include a special character (e.g., a newline in a snippet), it must be properly escaped as per JSON standards.

- Fields such as `timestamp` **MUST** follow the ISO 8601 date-time format (e.g., `"2025-04-01T10:00:00Z"` for UTC). If time zone is not specified, consumers should assume UTC or treat it as local time as per JSON date-time semantics.

- The `id` field for each flow **SHOULD** be unique within that VXDF document. It is **RECOMMENDED** (though not strictly required by schema) that if multiple VXDF files from the same source are merged, IDs remain unique to avoid confusion. For instance, a tool might generate IDs like `"F1", "F2", ...` each run; if combining outputs, one might prefix them with a run ID or use GUIDs. Uniqueness of IDs allows reliable reference (say, if a particular flow is discussed in a report or linked to an external ticket).

- The flows array **SHOULD** be considered an unordered set of findings by semantics (no inherent priority by order), though producers might order flows by severity or significance for convenience. Consumers **MUST NOT** assume a specific order (except that JSON will preserve whatever order was in the file).

- If a `steps` array is provided in a flow, it **SHOULD** logically connect the source to the sink. This means the first step should generally correspond to the source location (or at least be in the vicinity of it), and the last step to the sink. The spec does not enforce this automatically, but it is implied by the concept of steps. If steps are out-of-order or don't include source/sink, the flow might still technically validate against schema, but it would be considered a misuse of the format. Producers **SHOULD** either include the source and sink as part of the steps sequence (particularly if they want to provide a full trace), or use steps only for intermediate nodes. In any case, producers **SHOULD** document how they populate steps so that consumers know what to expect. A consumer **MAY** attempt to auto-correct or highlight if, say, the source is not included as a step – but generally the data should be consistent.

- The content of `source`, `sink`, and `steps` fields (file paths, function names, code snippets) should be relevant and appropriately detailed. For example, the `file` path should be something meaningful (absolute path, or project-relative path, or module identifier). The spec doesn't mandate a particular path format (different systems have different conventions), but **MUST** consistently use a string. It's good practice to, for instance, use UNIX-style forward slashes for paths or a URI scheme if appropriate (like a VS Code link or similar), as long as it's documented by the producing tool.

- **No Unknown Fields:** Apart from extension fields (discussed below), a VXDF document **MUST NOT** contain additional JSON properties that are not defined in the schema. If such fields are present and do not start with `x-`, they violate conformance. For example, a field `evidences` (note the plural) instead of `evidence` would be a schema violation, likely a typo. The schema's `additionalProperties: false` enforces this for each object, making the document invalid. This rule ensures a strict schema adherence and catches mistakes (e.g., a producer accidentally outputting `"severtiy": "High"` – spelled wrong – which would not be recognized).

- **Extension Fields:** Any extra fields must be prefixed with `x-`. These are exempt from the unknown field rule if they follow the naming convention. For instance, `x-customNotes` or `x-internalPriority` would be allowed. However, even extension fields must still be valid JSON values (no special types outside JSON). Producers adding extension fields **SHOULD** ensure the names are descriptive and avoid conflicts (e.g., choosing a prefix that might collide with another vendor's extension is unlikely due to the `x-` rule, but if a domain-specific community agrees on some extension, that's fine).

- If the specification is extended in the future, new official fields will be introduced in newer `vxdfVersion` values. A v1.0 consumer encountering a higher version **MUST** handle it gracefully, typically by either rejecting the file as unsupported or ignoring fields beyond its knowledge. The safe route is to treat unknown non-`x-` fields in a higher version file as errors unless the consumer is updated to that version's schema. (This basically follows normal versioning expectations; v1.0 tools are built to the v1.0 schema.)

Producer Conformance: A tool or process that outputs VXDF **MUST** produce files that meet the above document conformance. Additionally:

- It **MUST** populate all required fields. For example, if a static analysis tool cannot automatically generate an exploit payload, it must at least put some reasoned evidence (like a manual analysis note) in the evidence field rather than leaving it empty.

- It **SHOULD** use the fields as intended (e.g., do not misuse severity or category fields for something else; if an internal risk score needs to be conveyed, use an extension rather than overloading severity).

- It **SHOULD NOT** include confidential or sensitive data in the VXDF output that isn't necessary for understanding the vulnerability. (For instance, avoid dumping large chunks of proprietary code in the snippet beyond what's needed, or including personal data in evidence if using real user input for testing, see Security & Privacy Considerations.)

- It **SHOULD** ensure the JSON is well-formed and preferably provide a `$schema` reference (the `$id` and `$schema` fields as in the spec) in the output so that generic

JSON tools can automatically validate against the correct schema. The schema reference is recommended (pointing to an official URL hosting the schema once available).

- If multiple flows share some common aspect (e.g., same evidence or same source affecting two sinks), the producer currently has to duplicate that info per flow in VXDF v1.0 (there is no cross-reference mechanism in this version). The producer **SHOULD** handle this duplication if needed for clarity, or consider grouping logically. (This spec doesn't forbid multiple flows having identical evidence text, for example, if one test case validates two similar sinks.)

Consumer Conformance: Software that reads or ingests VXDF documents:

- **MUST** validate that the document conforms to the schema (or at least not blindly trust data if it doesn't conform). If a VXDF file is not valid, the consumer should reject it or handle the error (e.g., log a warning, attempt to normalize if minor).

- **MUST** interpret the fields according to spec. For example, a consumer showing results in a UI should label things appropriately: show "Source" and "Sink" clearly, use the severity string as given (maybe mapping to a color or priority internally), etc. If a field like `cwe` is present, it can be used to link to CWE details (like opening a reference web page) – this is not required, but a reasonable use.

- **MUST** ignore any `x-` extension fields it doesn't understand. "Ignore" in this context means it should not treat their presence as an error; it can drop them or pass them through if re-serializing. For instance, if a consumer is a aggregator that merges multiple VXDFs, it should carry forward unknown `x-` fields on flows, untouched.

- **SHOULD NOT** ignore known fields. If a consumer chooses not to utilize some field (say it doesn't display `category`), it should still correctly parse and store it rather than dropping it. This ensures that if the data is later exported again or audited, nothing is lost.

- **SHOULD** verify evidence when possible. While not a strict requirement of the format, a consumer tool might have the capability to rerun a proof-of-concept or check that the described evidence indeed triggers the issue (especially if the consumer is an interactive triage tool). Doing so can be useful, but the decision is up to the implementation. VXDF makes such checks easier because all needed info is present.

- If a consumer maps VXDF into another system (like creating bug tracker entries or converting to SARIF for some reason), it **SHOULD** maintain the key information. E.g., ensure the evidence text is included in the ticket description, etc. Essentially, data loss or

misinterpretation should be avoided.

Schema Validation: The normative JSON Schema provided can be used directly to validate documents. Using a JSON Schema validator that supports draft 2020-12 (or later) is the straightforward way. However, even without a validator, the rules can be checked with simple custom code or manual review:

- Verify required fields presence.
- Verify no disallowed extra fields.
- Check types (e.g., ensure line numbers are numbers, severity is one of the allowed strings, etc.).
- If any check fails, the document is non-conformant.

The authors of this spec will also provide a test suite (see Appendix F) with example valid and invalid VXDF files to help implementers ensure their validation logic is correct.

In summary, a conformant VXDF implementation treats the spec as a contract: producers promise to only output according to the schema, and consumers can rely on that guarantee to simplify parsing. When all rules are followed, VXDF documents become a reliably structured source of truth about confirmed vulnerabilities in code.

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: Sample VXDF File

To illustrate the VXDF format, here is an example VXDF document containing two vulnerability flows. This example is non-normative (just for understanding), but it follows the v1.0 schema. The scenario is a web application with a SQL injection and a cross-site scripting issue.


```
{
  "vxdfVersion": "1.0",
  "metadata": {
    "generator": "AcmeSecurityScanner 1.4",
    "timestamp": "2025-04-01T15:30:00Z",
    "target": "Acme WebApp v2.3"
  },
  "flows": [
    {
      "id": "F-1",
      "title": "SQL Injection in login form",
      "description": "Unsanitized user input from the login form is concatenated into a SQL query, allowing an attacker to bypass authentication.",
      "severity": "High",
      "category": "SQL Injection",
      "cwe": "CWE-89",
      "source": {
        "file": "src/com/acme/auth/LoginController.java",
        "line": 123,
        "function": "processLogin",
        "snippet": "String userInput = request.getParameter(\"username\");"
      },
      "sink": {
        "file": "src/com/acme/auth/UserDAO.java",
        "line": 45,
        "function": "findUserByName",
        "snippet": "ResultSet rs = stmt.executeQuery(\"SELECT * FROM Users WHERE name='\" + userInput + \"'\");"
      },
      "steps": [
        {
          "file": "src/com/acme/auth/LoginController.java",
          "line": 122,
          "function": "processLogin",
          "snippet": "// user submits login form with username and password",
          "note": "Start of data flow: login request received"
```

```
      },
      {
        "file": "src/com/acme/auth/LoginController.java",
        "line": 123,
        "function": "processLogin",
        "snippet": "String userInput = request.getParameter(\"username\");",
        "note": "Source: read username parameter from request"
      },
      {
        "file": "src/com/acme/auth/LoginController.java",
        "line": 130,
        "function": "processLogin",
        "snippet": "User user = userDao.findUserByName(userInput);",
        "note": "Passing tainted data into data access layer"
      },
      {
        "file": "src/com/acme/auth/UserDAO.java",
        "line": 40,
        "function": "findUserByName",
        "snippet": "String query = \"SELECT * FROM Users WHERE name='\" + name + \"'\";",
        "note": "Query is constructed with unsanitized input"
      },
      {
        "file": "src/com/acme/auth/UserDAO.java",
        "line": 45,
        "function": "findUserByName",
        "snippet": "ResultSet rs = stmt.executeQuery(query);",
        "note": "Sink: query executed with malicious input"
      }
    ],
    "evidence": [
      {
        "description": "By entering `admin' OR '1'='1` as the username (and any password), the attacker gains access without valid credentials. The SQL query becomes `SELECT * FROM Users WHERE name='admin' OR '1'='1'`, which returns all users, bypassing login.",
        "method": "manual test",
        "timestamp": "2025-04-01T14:55:00Z"
      }
    ]
  },
  {
    "id": "F-2",
    "title": "Reflected XSS via search query parameter",
```

```
    "description": "The web application reflects user input in the search results page without HTML-encoding, allowing JavaScript injection in users' browsers.",
    "severity": "Medium",
    "category": "Cross-Site Scripting",
    "cwe": "CWE-79",
    "source": {
      "file": "src/com/acme/search/SearchServlet.java",
      "line": 88,
      "function": "doGet",
      "snippet": "String query = request.getParameter(\"q\");"
    },
    "sink": {
      "file": "webapp/WEB-INF/jsp/results.jsp",
      "line": 22,
      "snippet": "<%= request.getParameter(\"q\") %>"
    },
    "steps": [
      {
        "file": "src/com/acme/search/SearchServlet.java",
        "line": 88,
        "function": "doGet",
        "snippet": "String query = request.getParameter(\"q\");",
        "note": "Source: read 'q' parameter from HTTP request"
      },
      {
        "file": "src/com/acme/search/SearchServlet.java",
        "line": 95,
        "function": "doGet",
        "snippet": "request.setAttribute(\"query\", query);",
        "note": "Store tainted query in request attribute"
      },
      {
        "file": "webapp/WEB-INF/jsp/results.jsp",
        "line": 22,
        "snippet": "<%= request.getAttribute(\"query\") %>",
        "note": "Sink: reflecting query in HTML output without encoding"
      }
    ],
    "evidence": [
      {
        "description": "Visiting the URL `/search?q=<script>alert(1)</script>` causes a JavaScript alert to pop up in the victim's browser when the results page loads. This shows that an attacker could inject arbitrary script.",
        "method": "dynamic test",
```

```
        "timestamp": "2025-04-01T15:00:00Z"
      }
    ]
  }
 ]
}
```

In this example:

- The first flow (F-1) is a **SQL Injection**. The source is where the application reads the username from the request (line 123 in `LoginController.java`). The sink is where that input is used in a SQL query in `UserDAO.java`. The steps array details each stage: receiving the request, reading the parameter, calling into the DAO, building the query, and executing it. The evidence shows an example of an attack payload (`admin' OR '1'='1`) and explains how it leads to a login bypass. Severity is High, category is SQL Injection, and CWE-89 is given.

- The second flow (F-2) is a **Reflected XSS**. The source is reading the “q” query parameter in a search servlet. The sink is the results JSP where the query is reflected directly. The steps illustrate how the data goes from the servlet to the JSP (via a request attribute). The evidence describes using a `<script>alert(1)</script>` payload and seeing an alert, confirming the XSS. Severity is Medium, category XSS, CWE-79.

- Both flows include metadata at top indicating the generator tool, a timestamp, and the target application name.

This sample demonstrates how VXDF concisely captures each vulnerability’s story: what the issue is, where in code it occurs, how the exploit works, and how severe it is. The JSON structure can be parsed by tools or read by humans (with the help of formatting) to act upon these findings.

Appendix B: Mapping to SARIF and SPDX

VXDF is designed to complement existing standards. This appendix outlines how one might translate or relate VXDF data to two relevant formats: SARIF (Static Analysis Results Interchange Format) for code analysis results, and SPDX (Software Package Data Exchange) for software inventory and basic vulnerability metadata. Understanding these mappings can help integrate VXDF into broader ecosystems.

Mapping VXDF to SARIF (Static Analysis Results)

SARIF is a standard format for static analysis tool output (Unlocking the Power of SARIF: The Backbone of Modern Static Analysis - DEV Community). While SARIF is very feature-rich and can represent many kinds of results (including those with or without code flows), VXDF covers a narrower, specialized case (validated flows). Nonetheless, there's conceptual overlap, and a VXDF flow can be represented in SARIF terms as a result with a code flow and attachments. Here's how key VXDF elements correspond:

- **SARIF `run` and `results`:** A SARIF file typically has a `runs` array (each for a tool execution) containing a list of `results`. In a mapping, each VXDF flow could become a SARIF `result`. If the VXDF file is produced by a single tool execution, one could map that to one SARIF run containing multiple results. The SARIF `tool` object would capture the VXDF generator (from metadata) as the tool name, etc.

- **Rule or Reporting Descriptor:** SARIF encourages defining a rule for each kind of issue (like "SQL Injection" rule). In mapping, the VXDF `category` or `cwe` can map to a SARIF rule. For example, a rule id might be "CWE-89" with a friendly name "SQL Injection". The SARIF result would reference that ruleId. This allows grouping results by type in SARIF viewers.

- **Locations (Source & Sink):** SARIF results often have a primary location (`result.locations[0]`) which is where the issue is considered to occur. For something like an injection, often the sink is considered the place to fix (e.g., the vulnerable query). So we could map the VXDF `sink` to the SARIF result's location. The `source` and intermediate steps would then be represented as a SARIF **codeFlow**. SARIF codeFlow consists of `threadFlowLocations` (essentially the steps in a trace). We would create a threadFlowLocation for each step in VXDF (and likely one for the source and sink as well, to have a complete path). Each threadFlowLocation has a `location` property (file, region, etc.) which maps nicely to the VXDF file/line. SARIF even allows an optional `message` on each step, which we could use to carry the VXDF step's `note`. For example, SARIF might look like: first threadFlowLocation is the source line with message "Source: read username parameter", intermediate ones follow, last is the sink with message "Sink: query executed…".

- **Evidence:** SARIF doesn't have an explicit first-class "evidence" concept for dynamic verification. However, SARIF results do have a `message` (typically a description of the issue) and they allow `attachments` or `relatedLocations`. We could map the VXDF evidence into the SARIF result's message or put it as an `artifact` attachment. For instance, one could add a `result.message.text` that includes something like "Proof: [the evidence description]". Some SARIF consumers might not show attachments clearly, so including key evidence details in the main description ensures it's seen. Alternatively, SARIF's `properties` (a free-form key-value) could include an `evidence` entry with the same content. This way, the data isn't lost, even if SARIF doesn't define a

specific slot for it.

- **Severity:** SARIF has a concept of result `level` (e.g., "error", "warning", "note") and also allows specifying `rank` or `properties.severity`. We can map VXDF severity to SARIF's `level`: for example, VXDF "High" and "Critical" might become SARIF level "error" (meaning urgent), "Medium" as "warning", "Low" as "warning" or "note" depending on policy, and "Informational" as "note". Additionally, we could include the exact VXDF severity string in a `properties` field if needed to avoid any ambiguity. SARIF doesn't predefine the "Critical" vs "High", it just knows error/warning/note, so that mapping loses some granularity unless we carry it in an extension property.

- **CWE and references:** SARIF has a place to put references to taxonomies like CWE. If the SARIF report defines a rule for this issue, that rule can include a reference (e.g., `"taxonomies": [{"name": "CWE", ... "id": "CWE-89"}]`). So mapping VXDF's cwe is straightforward into SARIF's rule metadata. Alternatively, SARIF result can have `result.taxonomies` reference too. This is a detail, but essentially, yes, we can propagate the CWE.

- **Run-level metadata:** If the VXDF `metadata.target` is something like a product name, in SARIF one could include that in the run's `columnKind` or an invocation property. SARIF doesn't have an explicit target field, but one could put it in `run.invocations[0].executionSuccessful` or more simply in `run.tool.driver.name` (though that's meant for tool name) or just in the run's properties. The `generator` (tool name) from VXDF would map to SARIF's `run.tool.driver.name` and `run.tool.driver.version`.

- **Multiple flows per result:** SARIF can technically have multiple codeFlows for one result, if an issue has multiple paths. In VXDF, we'd normally list them as separate flows if they are distinct vulnerabilities. However, if a VXDF producer used one flow to represent something like "multiple sources to one sink", that doesn't map 1-to-1 easily. In such a case, splitting them is advisable. SARIF would handle multiple sources as separate results or as one result with parallel flows. VXDF leans toward separate flows for clarity.

Example Mapping: Taking the first flow (SQL injection) from Appendix A, a SARIF result could be:

- `ruleId: "CWE-89"` (with a rule description "SQL Injection" in the SARIF tool's rules list).

- `level: "error"` (since it's High severity).
- `message: "Unsanitized user input from login form flows into SQL query (potential SQL Injection). Evidence: By entering 'admin' OR '1'='1', login bypass was achieved."` (combining description and evidence succinctly).
- `locations[0]`: points to `UserDAO.java#45` (sink).
- `codeFlows[0].threadFlowLocations`: an array with:
    - location for `LoginController.java#123` (source), with message "Source: username parameter read",
    - location for `UserDAO.java#40` (intermediate, building query),
    - location for `UserDAO.java#45` (sink, executeQuery).
- Additionally, the rule metadata in SARIF includes the CWE-89 reference and perhaps a link to the CWE database.
- The run tool is "AcmeSecurityScanner 1.4".

A SARIF viewer would then let you see the issue "SQL Injection in login form" with the trace from source to sink and a message describing evidence.

Limitations: Some richness of VXDF might not cleanly translate to SARIF. For example, VXDF evidence might be very detailed or multiple; SARIF primarily has one message per result (though one could enumerate evidence points in it). Also, SARIF doesn't differentiate a "validated" result vs not – it's up to the producing tool's context. In our mapping, all SARIF results coming from a VXDF would inherently be validated, but SARIF doesn't label them as such explicitly. We could add a property like `"validated": true` in SARIF's custom properties to preserve that notion.

In summary, **VXDF to SARIF mapping is feasible**: think of VXDF as a subset of SARIF (with added emphasis on evidence). One could write a converter that takes a VXDF JSON and emits SARIF JSON fairly straightforwardly, using the guidelines above.

Mapping VXDF to SPDX (and SBOM context)

SPDX is an SBOM format that lists software components, their metadata (licenses, versions), and can include references to vulnerabilities or external documents (SPDX: It's Already in Use

for Global Software Bill of Materials (SBOM) and Supply Chain Security - Linux Foundation). While VXDF is not an SBOM, it can be linked to one. The idea is that an SBOM tells you *what* is in your software, and a VXDF can tell you *where the vulnerabilities are* in that software with detailed evidence.

There are a couple of ways to map or integrate VXDF with SPDX:

- **Via Package/Component Identifiers:** If a VXDF document has a `metadata.target` that identifies a software package, this can correspond to an SPDX **Package** name or identifier. For instance, if `target` is "Acme WebApp v2.3", the SBOM for Acme WebApp v2.3 would have a Package with that name and version. To link them, one could use the SPDX `ExternalReference` or `Annotations`. SPDX allows external references of type "security" or "vulnerability". For example, an SPDX document might have an `ExternalRef` for the package:

    - `ExternalRef: SECURITY cpe23Type <cpe> ...` (that's one way, not directly linking to VXDF though). More directly, if the VXDF is published somewhere (say as a file or URL), SPDX could reference it. There isn't an official "VXDF" reference type yet, but one could repurpose the `SECURITY` reference type or use a generic one. For instance:

```
PackageName: Acme WebApp
PackageVersion: 2.3
...
ExternalReference:
  RefType: security-advisory
  Locator: https://example.com/vxdf/acme-webapp-2.3-findings.json
  Comment: "Validated flows vulnerability report (VXDF format) for this package"
```

- This way, someone looking at the SBOM finds that reference and can retrieve the VXDF for details.

- **Using Vulnerability Extension (if any):** Newer SBOM efforts sometimes incorporate vulnerability info (for example, CycloneDX SBOM has a Vulnerabilities section; SPDX 3.0 might address this too). If SPDX or an extension lists vulnerabilities (like CVEs or others) affecting a component, VXDF could provide the technical details. For instance, suppose SPDX notes that Acme WebApp v2.3 is affected by a vulnerability "ACME-2025-001 – SQL Injection in login". The VXDF could serve as the detailed report for that vulnerability. An SPDX `DocumentComment` or `Annotation` could include a pointer or summary: "See VXDF report ACME-2025-001.json for exploit path and proof."

- **CWE and other references:** SPDX can include references to CWE or CVE under `Annotations` or `Relationship`. However, SPDX is more about cataloging than

detailing. If an SPDX document enumerated vulnerabilities, it might list a CVE and possibly CWE. VXDF's role would be to show the internals. So the mapping here is more about cross-referencing by identifiers. For example, if a VXDF flow has `cwe: CWE-79`, an SPDX might have an annotation like "Vulnerability contains CWE-79". But it's not a direct one-to-one field mapping, rather a conceptual linkage.

Including VXDF data in SBOM: One could theoretically embed some of the VXDF info into an SBOM, but it's probably not ideal to put full flow details inside an SBOM JSON, as it serves a different purpose. Instead, linking is cleaner. The SBOM could carry high-level vulnerability status (like "vulnerable" or "fixed" flags, akin to VEX), while the VXDF carries the detailed proof for those marked as "vulnerable".

Using Package Identifiers in VXDF: VXDF can aid mapping by including package identifiers in its content. For example, if each flow had a field for `component` or `library`, that would directly map to an SBOM entry. In v1.0, we didn't include a dedicated field for "component name" beyond `metadata.target`. But an extension could: e.g., `x-purl` (Package URL) or `x-spdxID` could be added in metadata or per flow if needed. Then a machine could join VXDF flows to SBOM components easily. We recommend that if a tool knows the specific package or library a flow is in (say the vulnerable code is in a dependency), it use an extension to note that, using a PURL or SPDX identifier. That would allow future automation where you go from SBOM -> find component -> see if there's a VXDF for that component's vulnerabilities.

Licenses and Patents: SPDX primarily tracks licenses; VXDF doesn't deal with that at all. So there's no overlap or mapping needed there, except to note that including code snippets might technically raise license questions (if you copy code from a proprietary codebase into VXDF, you are distributing a piece of that code). However, that's usually small snippets for security reasons (fair use likely, or internal use). From a mapping perspective, one might ensure that if an SBOM lists licenses, and you plan to share VXDF externally, you're not violating licenses by sharing code snippets. This is a legal concern rather than technical mapping.

Summary: VXDF and SPDX serve different roles:

- SPDX tells you *what* components and known issues exist, often using known vulnerability IDs (CVE).

- VXDF tells you *how* a particular vulnerability can be exploited in the code (often a custom or specific finding not necessarily tied to a CVE if it's an internal discovery, or providing detail beyond a CVE description).

Mapping between them is mostly about identification:

- Ensure the VXDF is clearly associated with the right component (via name, version, or a unique ID).

- In the SBOM or adjacent VEX document, reference the VXDF report for details.

As an example of combined usage: A company produces an SBOM for their product release and also a VXDF report of all the vulnerabilities found during testing of that release. The SBOM might include an note: “No known exploitable vulnerabilities except those listed in Company-VXDF-Report.json” which is included as an attachment. Or if one were using SPDX JSON format, one could include a custom extension section with an array of VXDF flow summaries. However, that gets into extending SPDX itself.

For now, the simplest practical mapping is **via external reference**: treat the VXDF file as a supplementary document linked to the SBOM or to vulnerability advisory. SPDX allows a concept called "security notice" reference. If that were used, it might link to a human-readable advisory. We can repurpose it to point to a machine-readable VXDF.

In conclusion, **VXDF and SPDX integration** is done by linking and shared identifiers:

- Use consistent naming (package names, versions, and if possible PURLs/CPEs) in VXDF metadata so that an SBOM consumer knows what the flows apply to.

- Use SBOM fields (ExternalRef, etc.) to point to the VXDF or to label known vulnerabilities which a VXDF then explains.

As both standards evolve (with SPDX possibly adding more vulnerability info, and VXDF possibly adding component identifiers), this integration can become tighter. The goal is that a user can go from an inventory of software to understanding detailed exploitability without ambiguity.

Appendix C: Reference Implementation Sketch (Pseudo-CLI)

To foster adoption, a reference implementation of VXDF could be provided. This would likely include a library for reading/writing VXDF and a simple Command-Line Interface (CLI) tool for common operations. Below we sketch how such a CLI might work, using hypothetical commands to illustrate functionality. This is *not* an exhaustive list, but gives an idea of how users and integrators might interact with VXDF data using tools.

1. Validating a VXDF file
A basic feature is to validate a VXDF document against the specification schema. For example:

$ vxdf validate report.vxdf.json

This command would parse `report.vxdf.json` and check it against the VXDF schema (perhaps included in the tool or fetched). If the file is valid, it might output “Valid VXDF (1.0)”. If not, it would output errors, e.g., “Error: flow[2].severity is not one of [Critical, High, ...]” indicating where the issue lies. This helps producers debug their output and ensures consumers can trust a file.

2. Pretty-printing or summarizing a VXDF
A user might want a human-readable summary:

$ vxdf summarize report.vxdf.json

Output (for example):

VXDF Report (Version 1.0) - Target: Acme WebApp v2.3
Generated by AcmeSecurityScanner 1.4 on 2025-04-01T15:30:00Z

Flows (2):
[F-1] High - SQL Injection in login form (CWE-89)
[F-2] Medium - Reflected XSS via search query parameter (CWE-79)

This gives a quick overview of how many issues and their headlines. One could then drill down:

$ vxdf show report.vxdf.json F-1

This might display detailed info for flow F-1:

Flow F-1: SQL Injection in login form
Severity: High (SQL Injection, CWE-89)
Source: LoginController.java:123 (function processLogin)
Sink: UserDAO.java:45 (function findUserByName)
Evidence: By entering `admin' OR '1'='1` as username, login bypass was achieved.
Steps:
1. LoginController.java:123 - read username parameter from request
2. LoginController.java:130 - call findUserByName with userInput
3. UserDAO.java:40 - build SQL query with user input
4. UserDAO.java:45 - execute query (vulnerability triggers here)

Such an interactive CLI output helps developers quickly glean details without manually parsing JSON.

3. Merging VXDF files
If you have multiple VXDF reports (say from different tools or different microservices) and want to combine them:

$ vxdf merge scan1.vxdf.json scan2.vxdf.json -o combined.vxdf.json

This would produce `combined.vxdf.json` containing flows from both inputs. The tool would need to handle merging carefully – ensuring unique IDs (maybe renumbering or prefixing if there are collisions), merging metadata if needed (perhaps keeping a list of generators or noting multiple sources). The CLI could also allow merging all files in a directory, etc. This is useful for consolidation of results.

4. Converting other formats to VXDF
One big use-case: converting a SARIF file to VXDF (for those results that are confirmed exploitable), or conversely, producing a SARIF from VXDF. For instance:

$ vxdf convert --from sarif scan.sarif.json --to vxdf output.vxdf.json

This would take a SARIF static scan result and produce a VXDF. However, because SARIF findings may not be validated, perhaps the converter would only pick those marked in some way or allow the user to filter. Alternatively, the converter might include a stub evidence like "(No dynamic validation provided; treating static result as potential)" which is somewhat against the spirit of VXDF unless the user explicitly wants to wrap unvalidated issues. Perhaps more practically, a conversion might be paired with a verification step (if possible).

Conversely:

$ vxdf convert --to sarif report.vxdf.json --output converted.sarif.json

Would embed VXDF info into a SARIF report as discussed in Appendix B. This could allow using SARIF viewers to inspect VXDF data.

Similarly, one could imagine:

$ vxdf convert --to html report.vxdf.json --output report.html

To produce an HTML report (with nice formatting, maybe even embedding code snippets... (continued)

4. Converting VXDF to other formats
Beyond JSON, users may want to generate reports or integrate with other systems. The reference CLI could offer conversion options:

To HTML (or PDF): Generate a human-friendly report. For example:

$ vxdf convert --to html report.vxdf.json -o report.html

- This might produce an HTML file containing a nicely formatted summary of flows, maybe with collapsible code snippets for each step, etc. This is useful for sharing with management or teams who prefer not to read raw JSON. The tool would handle embedding the content safely (escaping any dangerous bits as discussed in Security Considerations). The resulting HTML could highlight source and sink code in context, list evidence, etc.

To SARIF: As discussed, one could convert VXDF to a SARIF file:

$ vxdf convert --to sarif report.vxdf.json -o converted.sarif.json

- This would allow using SARIF viewers (like IDE plugins or GitHub code scanning interface) to visualize the flows. The conversion would map fields as per Appendix B.

From SARIF or other: Similarly, if a user has a SARIF and wants to create a VXDF skeleton:

$ vxdf convert --from sarif tool-findings.sarif.json -o potential-flows.vxdf.json

- This could attempt to generate VXDF flows from SARIF results, but since SARIF results may not have evidence, the tool might mark the flows with a placeholder evidence like `"description": "From static analysis (not yet validated)"`. Such a VXDF could then be updated once validation is done. (This is an advanced use and would be clearly documented as not producing fully valid flows unless evidence is added.)

5. Comparing VXDF reports (diffing)

When VXDF is used continuously (for example, every build or release has a VXDF report of found vulnerabilities), it's useful to see what changed. A CLI might allow comparison:

$ vxdf diff old.vxdf.json new.vxdf.json

The tool would then output differences, such as:

Comparison of VXDF reports:
- Removed 1 flow from old report (no longer present in new):
  [F-3] Medium - XSS in user profile (fixed in new version)
- Added 2 new flows in new report:
  [N-1] High - XML External Entity injection in Export feature

[N-2] Low - Information exposure through error messages

The tool could match flows by `id` if the same IDs are used, or by some heuristic (same source/sink positions) if IDs differ between runs. This helps track progress (e.g., are we fixing issues faster than introducing new ones?) and is especially helpful in CI to identify if a new build introduced a high-severity issue.

6. Policy checks and integration in CI

A reference implementation could also enable simple policy enforcement. For example, an organization might decide that no Critical or High issues are allowed before release. The CLI could have a command or flag to enforce this:

$ vxdf check --max-allowed-severity=Medium report.vxdf.json

This would examine the highest severity in `report.vxdf.json`. If there's any High or Critical, it would exit with a non-zero status and perhaps print a message like "Policy violation: Found High severity flow F-1 (SQL Injection)". In a CI pipeline, this non-zero exit could fail the build, preventing promotion of code with serious vulnerabilities. Conversely, if all issues are Medium or below (as allowed by the flag), it exits 0 and the pipeline continues (maybe logging "All issues are within acceptable severity range"). Additional policy options might include checking that every flow has certain fields (though by spec they will), or that evidence is recent (e.g., not older than X days). These go beyond the core spec, but illustrate how a reference tool can help integrate VXDF into automated workflows.

The above commands are illustrative. A real implementation might combine functionalities (for instance, `vxdf validate` might by default also summarize). The key point is that a reference CLI empowers users to adopt VXDF easily:

- It lowers the barrier to validating and viewing VXDF files.
- It provides utilities to combine and convert VXDF data, maximizing interoperability (e.g., feeding into reporting dashboards or other formats).
- It demonstrates best practices (like the `check` for CI, encouraging use of VXDF in gating decisions, which ties back to the goal of focusing on real issues).

Such a reference implementation would likely be open-source, allowing community contributions. It would serve as both a tool and an executable form of the spec (ensuring the spec is implementable). Over time, as VXDF evolves, the reference CLI would be updated accordingly.

Appendix D: 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 E: Extension Mechanism

While VXDF aims to cover the common core of validated data flow reporting, it's impossible to foresee every need. Different organizations or tools may want to include additional data specific to their context. To accommodate this without fragmenting the standard, VXDF includes a clear extension mechanism based on **namespacing**.

Using `x-` Prefix: The specification dictates that any non-standard field be named starting with `x-`. This convention is similar to how HTTP headers or other formats handle custom fields. For example:

- A company might have an internal risk scoring system and want to include that for each flow. They could add `"x-riskScore": 4.7` (perhaps representing a CVSS-like score or some internal metric).

- A tool might want to include the ID of the finding in its own database: `"x-internalId": 12345`.

- If a flow relates to an existing bug tracker ticket or CVE, one might use `"x-ticket": "PROJ-99"` or `"x-cve": "CVE-2025-1234"` (though future VXDF versions might officially support linking to CVEs).

These fields would appear alongside standard ones. Because of the JSON schema rules, any `x-` field is allowed (as long as it follows JSON types) and will not cause validation errors.

Guidelines for Extensions:

- **Purposeful Naming:** Choose a name that clearly identifies the purpose and possibly the organization. A generic name like `x-priority` might coincidentally be used by two tools to mean different things. If your extension is very tool-specific, you might even include the tool name: e.g., `x-ScannerXFoo`. However, if it's potentially of broader interest, keep it generic but still unique enough by context.

- **Avoid Conflicts:** Since all extensions use the same prefix space, there is a small risk two parties use the same name for different concepts. While the spec cannot prevent that, communication in the community can. For example, if a popular extension emerges, others should avoid reusing that name for something else.

- **No Override of Standard Meaning:** An extension should not be used to replace or duplicate a standard field in a way that confuses interpretation. For instance, don't use `x-severity` if it's basically the same info as `severity`. Use the official field. Extensions are for new or additional info, not for alternate encodings of existing info. (If a different severity scheme is needed, one could still map to the standard field as best as possible and then include the raw score as an extension.)

- **Documentation:** Producers of VXDF with custom fields should document them for the consumers. If you publish a VXDF file to others, provide a short explanation of any `x-` fields included (either in a separate document or even within the VXDF's metadata as an `x-` note).

- **Transition to Standard:** If an extension field proves broadly useful, propose it for inclusion in the next VXDF version. The goal is not to have permanent fragmentation. For example, suppose many people start adding `x-fixGuidance` with tips on how to fix the issue. The working group might decide to create an official `remediation` field in v1.1 or v2.0 that serves this purpose, and then everyone can migrate to that and drop `x-fixGuidance`. Thus, the extension mechanism is a pathway for innovation that can feed back into the standard.

Structured Extensions: The extension fields themselves can hold complex objects or arrays, not just primitive values. For instance:

```
"x-experiment": {
   "analysisId": "ABC123",
   "extraFindings": [ ... ]
}
```

The schema allows any shape for `x-` fields (it doesn't validate their internal structure). This means you could even nest standardized sub-objects inside an extension if needed. However, be mindful that consumers who aren't aware of your extension will ignore it entirely (they won't parse into its internals). So if an extension is critical for understanding the vulnerability, that's a sign it probably should be a standard field instead.

Examples of Potential Extensions:

- `x-runtimeEnviron`: If a dynamic test is done, maybe info about the environment (e.g., "Java 11 on Windows") which could be relevant if the exploit depends on it.
- `x-exploitabilityScore`: Some system might compute a score 0-100 of how easy it is to exploit beyond just severity.
- `x-patternId`: If the issue relates to a known pattern or template (like a specific secure coding guideline ID).
- `x-fixCommit`: a link or reference to a source control commit that fixed this issue (useful in retrospective documentation).
- `x-discussion`: maybe a URL to a discussion thread or an internal wiki about this vulnerability.

All such data can ride along in VXDF without breaking anything.

Consumer Handling: By rule, consumers ignore unknown `x-` fields. However, a consumer might provide a feature to surface them generically. For instance, a viewer could list "Additional info:" and then dump the JSON of the extension fields or at least display keys. Advanced consumers could even be configured to recognize certain extensions (if, say, a company writes a custom internal extension and updates their internal tools to show it specially). But outside that context, extensions won't be understood, which is fine.

Caution: One should not rely on an extension for any critical logic in inter-organizational exchange unless you are sure the other side knows about it. For example, don't put the only indication of something essential in an `x-` field when sharing with a third party, because their tooling might drop it. Always ensure the main content of VXDF suffices for understanding, and extensions only enrich it.

In summary, the extension mechanism is flexible and meant to encourage innovation and accommodate special needs, while protecting the base standard. It ensures forward compatibility (a VXDF v1.0 consumer can safely handle a v1.0 file with extra fields it doesn't

know about) and avoids the need to fork the format for niche requirements. The VXDF community will monitor common extensions and work to incorporate those that add general value.

Appendix F: Reference Test Suite

To verify implementations and ensure consistency, a reference test suite will be developed alongside the VXDF specification. This test suite will contain a collection of VXDF example files and possibly automated scripts to validate behavior. Its purposes are:

- **Validate Parsers/Generators:** Tool developers can run their VXDF reading/writing code against the suite to check for compliance.
- **Prevent Regressions:** As the spec evolves, the test suite helps catch if a change would break previously valid files or common usage patterns.
- **Demonstrate Edge Cases:** Some tests illustrate tricky or boundary conditions so implementations handle them correctly.

Contents of the Test Suite:

1. **Valid Example Files:** A set of `.json` files that are correct VXDF instances. These will range from simple (maybe one flow with minimal fields) to complex (multiple flows, with all optional fields populated, long step traces, etc.). Each file will have a short description. For example:
    - `valid-minimal.json`: Smallest possible VXDF (one flow, just required fields).
    - `valid-fullsample.json`: A file like the one in Appendix A, utilizing many features.
    - `valid-extfields.json`: A file that includes some `x-` extension fields to ensure that doesn't break validation.
    - `valid-multiflow.json`: Many flows to test performance and ordering (maybe 50 flows).
2. **Invalid Files:** JSON files that intentionally break one rule or another, to test validators:
    - `invalid-missing-field.json` (e.g., a flow without an `id`).

- `invalid-bad-enum.json` (e.g., severity is "Severe" which is not allowed).
- `invalid-extra-field.json` (a non-`x-` unknown field present).
- `invalid-wrong-type.json` (e.g., line number given as string). These help ensure that validation logic catches errors. A conformant parser should reject these and possibly list the reasons.

3. **Boundary/Edge Cases:**
    - Extremely long text in fields (to test that no truncation or overflow occurs).
    - Special characters in strings (like newline in snippet, or non-ASCII characters) to ensure JSON encoding issues are handled.
    - Zero-step flows vs with steps (to confirm either is acceptable).
    - Perhaps a case with multiple evidence entries.
    - A case with nonstandard file path formats (to see that it's just treated as string).

Automation: The test suite repository will include a script (perhaps Python or using JSON Schema validators) that goes through all `valid-*.json` and confirms they pass schema validation, and goes through `invalid-*.json` to confirm they fail validation. This double-checks the schema itself and serves as a quick compliance check for any JSON schema validator tool.

Tool makers can run similar tests: e.g., feed `valid-*.json` into their VXDF loader and ensure it loads without errors and data matches expectations, and feed `invalid-*.json` to ensure their loader flags an error.

Continuous Integration: The VXDF spec project will likely have CI set up such that any change to the schema or spec triggers the test suite. If someone proposes a change that inadvertently makes a previously valid file invalid (or vice versa in a wrong way), tests would catch it. This helps maintain backward compatibility or at least awareness when something changes.

Adding New Tests: As issues are discovered in the field, we can add corresponding tests. For example, if a certain combination of fields confuses some parser, we add a test to ensure clarity. Or if we add a new feature in v1.1, new tests for that will accompany.

Interoperability Testing: Besides static files, another aspect is ensuring different producers and consumers interoperate. The test suite might collect outputs from various reference

implementations and try them in others. While not part of the reference suite initially, the community could share real-world VXDF samples (sanitized) to enrich the test cases over time.

Performance Consideration: While JSON Schema validation is straightforward, we might also have tests for performance (like how a parser handles a file with, say, 10,000 flows). That's more for implementers to consider, but not a strict part of conformance. However, we could include a large sample as a stress test.

Usage of Test Suite by Others: For example, if OWASP has a project on vulnerability reporting, they can incorporate the VXDF test suite to ensure their output is correct. Or a vendor can publicly state they pass all reference tests (which gives confidence in their implementation).

In conclusion, the reference test suite is an essential companion to the spec: it turns the written rules into practical examples and checks. It will evolve with the spec. Ultimately, when someone says "we support VXDF", running them through the test suite is a way to verify that claim.

Appendix G: 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 H: 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.

In closing, the VXDF project is committed to measuring its impact and being responsive. By defining clear metrics and having a forward-looking plan, we aim to ensure that VXDF not only starts strong but continues to deliver value throughout its lifecycle – helping security teams and developers work together more effectively to secure software.