Validated Exploitable Data Flow (VXDF) Standard – Architectural Blueprint

1. Introduction and Scope

Overview: Defines the purpose, scope, and context of the VXDF standard, and outlines the key problems it addresses (e.g. false positives in vulnerability detection) and the high-level goals of the specification. This section sets the stage for why VXDF is needed and how it will be used.

- 1.1 Background and Purpose: Describes the motivations for VXDF, including the need to distinguish truly exploitable vulnerabilities from theoretical ones by coupling static/dynamic analysis results with proof of exploit. Introduces how VXDF will enable reliable result interchange and CI/CD gating by eliminating false positives.

- 1.2 Scope: Defines what is covered by the VXDF standard (the JSON-based data format for representing validated exploit paths). Clarifies the focus on confirmed exploitability evidence and data flows. Also notes any aspects that are out of scope (for example, it is not a vulnerability detection technique itself, and it's not a full SBOM format).

- 1.3 Key Objectives and Principles: Summarizes the core objectives of VXDF – for instance, interoperability between security tools, deterministic behavior in pipelines, proof-centric reporting, and open development (OWASP-style) approach. Mentions that VXDF may evolve into a formal standard in the future and is built with that rigor in mind.

- 1.4 Intended Audience: Identifies the target audience for the specification (SAST/DAST/SCA tool vendors, application security engineers, CI/CD pipeline integrators, compliance and risk management teams).

- 1.5 Relationship to Existing Standards: Provides a high-level introduction to how VXDF relates to or complements existing standards like SARIF (for static analysis results) and SBOM standards such as CycloneDX/SPDX, as well as vulnerability scoring systems (CVSS). This serves as a prelude to more detailed mapping later in the document.

2. Normative Language and Terminology

Overview: Establishes the conventions for normative requirements in the specification and defines important terms used throughout the document. This ensures a common understanding of language and concepts.

- 2.1 Normative Keywords: Specifies how requirement levels are indicated using RFC 2119 keywords (“MUST”, “SHALL”, “SHOULD”, “MAY”, etc.) . Explains that these terms are to be interpreted as described in RFC 2119 and its updates, to distinguish mandatory requirements from recommendations . For example, VXDF MUST use JSON format as defined by ECMA-404 , and producers SHOULD include all relevant evidence for each vulnerability flow.

- 2.2 Terminology: Defines key terms and acronyms used in VXDF in a consistent manner:

    - Validated Vulnerability: A security vulnerability that has been confirmed to be exploitable in the given context, typically represented in VXDF as an exploit path with supporting evidence.

    - Exploit Path / Data Flow: The sequence of steps (sources, sinks, and intermediate nodes) through which malicious data or control flows to exploit a vulnerability.

    - Source: An entry point where untrusted or attacker-controlled data enters the system (e.g. an input parameter, external request).

    - Sink: A vulnerable function or location in code where the exploit occurs (e.g. an API call that executes a command injection).

    - Evidence: Artifacts or data that confirm the exploitability of a vulnerability (such as a runtime output, stack trace, or proof-of-concept demonstrating the issue).

    - SAST/DAST/SCA: Security testing approaches – Static Application Security Testing (code analysis), Dynamic Application Security Testing (running application testing), Software Composition Analysis (dependency vulnerability analysis). These terms are used when describing the tools or methods that generate VXDF entries.

    - SBOM: Software Bill of Materials – a detailed inventory of components (for context, SBOM formats like CycloneDX or SPDX may incorporate or reference VXDF data for vulnerabilities).

    - VEX: Vulnerability Exploitability eXchange – a format for indicating whether a known vulnerability is exploitable in a product’s context. VXDF provides the detailed proof that can feed into a VEX statement.

    - (Additional terms like CI/CD, CVSS, OWASP can be defined here as needed, or in the Glossary Appendix, to ensure clarity.)

3. Use Case Scenarios

Overview: Illustrative scenarios demonstrating how VXDF is applied in practice. Each scenario highlights a particular context in which representing validated exploitable data flows is valuable. These use cases guide the requirements and design of VXDF.

- 3.1 Validated SAST Finding (Static Code Analysis): A static analysis tool finds a potential vulnerability in code. VXDF is used to represent the finding after it has been validated (for example, by manual code review or automated reasoning that confirms a data flow from a controllable source to the vulnerable sink). The use case shows how the static result (perhaps originally in SARIF) is distilled into a single canonical exploit path with proof, eliminating false positives.

- 3.2 Validated DAST Exploit (Dynamic Analysis): A dynamic scanner or penetration test identifies a vulnerability (e.g., an SQL injection) and actually exploits it in a test environment. VXDF captures the exact HTTP request/response or sequence of steps used to exploit the vulnerability, along with the data flow through the application that led to the exploit. This scenario demonstrates how runtime validation results are documented in a standard way, enabling replay or analysis by other tools.

- 3.3 Third-Party Dependency Exploitability (SCA Integration): A Software Composition Analysis tool flags a known vulnerability in a third-party library (via a CVE in an SBOM). VXDF is used to confirm whether that vulnerability is truly exploitable in the application. For example, it might include a static trace showing a call to the vulnerable library function with attacker-controlled data. If exploitable, the library vulnerability is represented as a VXDF exploit path (mapping to the component in the SBOM and the CVE). If not exploitable, no VXDF entry is produced (and this could feed a VEX statement that the product is not affected).

- 3.4 Integration into SBOM and VEX for Compliance: Describes using VXDF outputs in the context of compliance and risk management. For instance, an organization generates a CycloneDX or SPDX SBOM for their application and wants to include or reference only the exploitable vulnerabilities (to focus remediation efforts). VXDF provides the machine-readable evidence of exploitability that can be referenced in a SBOM's vulnerability section or used to create a VEX document asserting exploit status. This scenario shows feeding validated findings into compliance artifacts like security attestations.

- 3.5 CI/CD Pipeline Gate (Deterministic Build Stopper): Shows how VXDF can be used as a gate in continuous integration/continuous deployment. A security tool in the pipeline produces VXDF entries for any findings it can actually exploit. The pipeline checks the VXDF output – if any high-severity exploitable flows are present, the build fails deterministically. Because VXDF filters out theoretical issues, the pipeline avoids false positives and only stops for real, proven risks. This use case underscores the value of

VXDF in automated decision-making environments.

- 3.6 Enhanced Vulnerability Prioritization (Exploit-Aware Scoring): Illustrates how security teams might use VXDF to adjust vulnerability severity and prioritization. For example, two findings might both have a CVSS base score of 9.0, but if one is confirmed exploitable (VXDF evidence present) and the other is not, the team can prioritize the VXDF-backed issue higher. VXDF can feed into risk scoring systems by providing the evidence to justify marking a vulnerability as "Exploited" or adjusting the CVSS Temporal Exploitability sub-score. This scenario highlights the benefit of VXDF in risk assessment workflows.

(Each use case above will include a brief narrative and possibly a diagram or example JSON snippet in the full specification, to demonstrate how VXDF represents that scenario.)

4. Document Structure

Overview: Explains how the VXDF specification document is organized, to help readers navigate the standard. This section outlines the division between normative content (the rules and schema of VXDF) and informative content (examples, use cases, guidance).

- 4.1 Organization of this Specification: Describes the major parts of the document (Introduction, Core Data Model, Schema, etc. as listed in this outline) and what each part contains. It clarifies which sections are considered normative (mandatory for compliance, e.g. the data model and schema definition) and which are non-normative or explanatory (use cases, examples, guidance).

- 4.2 Reading Guide: Provides tips on how different audiences should read the document. For instance, tool implementers may focus on the schema and conformance sections, whereas security practitioners might focus on the use cases and interpretation of VXDF content. It may also mention that appendices contain supplemental material like the full JSON schema and examples.

- 4.3 Notational Conventions: (If needed) References any notational or stylistic conventions in the document. For example, how examples are formatted or how JSON schema snippets are presented (font, style), and that all JSON examples are in UTF-8. (This is similar to SARIF's "Conventions" section and CycloneDX's notation references.)

(Section 4 ensures the reader understands the layout of the spec – this is especially useful if the document is lengthy, much like the SARIF and CycloneDX specs.)

5. VXDF Core Data Model (Information Model)

Overview: This is the heart of the VXDF specification, defining the JSON-based information model. It describes all core objects, their properties, and relationships, in a manner similar to how SARIF and CycloneDX enumerate their schema elements. Each subsection below corresponds to a major element of the VXDF JSON structure, with detailed definitions of fields.

- 5.1 VXDF Document Container: Defines the top-level JSON object for a VXDF file. This includes:
    - 5.1.1 Schema Version: A required field indicating the VXDF version (e.g., "version": "1.0"), to support format evolution and backwards compatibility.
    - 5.1.2 Metadata: A container for metadata about the VXDF file and the analysis context (detailed in 5.2).
    - 5.1.3 Findings: An array of validated exploit flows (detailed in section 5.3). Each element in this array is an ExploitFlow object representing a single validated vulnerability instance.
    - 5.1.4 Summary (Optional): Optional aggregate information, such as the total count of findings, or a high-level summary of the risk (e.g., number of critical exploitable vulnerabilities). This may be included for convenience but is derivable from the list of findings.
- 5.2 Metadata: Describes the contextual and provenance information included in a VXDF document:
    - 5.2.1 Tool Information: Identifies the tool or system that generated the VXDF (name, version, vendor, etc.), similar to SARIF's tool metadata. If multiple tools contributed (e.g., SAST + a validator), it can list all contributors.
    - 5.2.2 Target Environment: Describes the system or application that was tested. This could include the application name and version, repository identifier, build ID, or environment details (e.g., test environment vs. production simulation). It provides context on where the exploit paths were validated.
    - 5.2.3 Execution Context: Metadata about when and how the validation was performed (timestamp of analysis, configuration used, any specific execution parameters, etc.). For example, a dynamic test might note the base URL of the deployed application, or a static validation might note the branch or commit ID of the code.
    - 5.2.4 Policy and Settings (Optional): Any relevant policy thresholds or settings in effect (e.g., only vulnerabilities of certain severity were validated, or certain sinks

were in scope). This helps interpret the results in context.

    - 5.2.5 Relationships/References: If applicable, references to related documents or identifiers, such as a report ID, an associated SARIF file, or an SBOM document ID from which this VXDF was derived. This links VXDF back to broader workflows.

- 5.3 ExploitFlow (Validated Finding) Object: Defines the structure of a single validated exploit path entry in the findings list. Each ExploitFlow object contains:

    - 5.3.1 Unique Identifier: A unique ID for the finding (within the VXDF file or globally unique if needed), to allow cross-referencing. This could be a GUID or a composite identifier (like a combination of vulnerability ID and location).

    - 5.3.2 Vulnerability Description: A human-readable title or description of the vulnerability. This may include a brief summary of the issue (e.g., “SQL Injection in user login function”) and possibly references like CWE identifiers or CVE IDs if it maps to a known vulnerability type.

    - 5.3.3 Category/Tags: Categorizations such as CWE classification (e.g., CWE-89 for SQL Injection) and tags indicating the type of flaw or the component it affects. This helps in filtering and aggregating results.

    - 5.3.4 Affected Component or Location: Details about where the vulnerability lies. For code issues, this might be a file name and line number or function name where the sink occurs (and possibly where the source is). For a dependency issue, this could identify the vulnerable library (name, version).

    - 5.3.5 Impact and Severity: The severity level (e.g., High, Medium, Critical) of this vulnerability in the context of the application, potentially with CVSS score metrics. This field can incorporate exploit-aware context – for example, noting that the CVSS Exploitability sub-score is maximum because an exploit was proven.

    - 5.3.6 Data Flow Trace: An embedded or linked representation of the exploit path’s data flow, from source to sink (detailed in section 5.4). This shows the steps that make this vulnerability exploitable.

    - 5.3.7 Evidence References: Links to the evidence that validate the exploit (the actual proof, detailed in section 7). This could be references to entries in an evidence collection (e.g., an ID that corresponds to a separate evidence object with full details) or inline snippets for quick reference (such as a short log excerpt).

    - 5.3.8 Remediation Guidance (Optional): Optionally, a field containing mitigation or fix advice (if provided by the tool or auditor). While VXDF's focus is on exploit data flows, including remediation notes can help integrate with ticketing systems or developer guidance. (This might be pulled from source data like static analysis rules or vulnerability databases.)

    - 5.3.9 Status/Resolution (Optional): For tracking purposes, a field indicating if this finding has been addressed (e.g., "open", "mitigated", "false-positive"). In VXDF, presumably all included findings are true positives (validated), so this might typically be "confirmed" or left out; however, this field could be used when VXDF is updated over time or linked to issue trackers.

- 5.4 Data Flow Trace Representation: Specifies how the sequence of steps that comprise an exploit path is represented in VXDF. This section breaks down the structure that models the path from a source to a sink:

    - 5.4.1 Trace Model Overview: Explains that a data flow trace is an ordered list of TraceStep objects capturing the propagation of malicious input through the system to the vulnerability. Each step can include the code location, the operation or function call, and how data is passed.

    - 5.4.2 TraceStep Object: Defines the fields of a trace step:

        - Location: The program location of this step (file, line, or component reference; could also include function name or program element).

        - Type of Step: Whether this step is a Source, Sink, or an Intermediate step (e.g., data transformation or propagation point). Possibly an enumeration or a flag.

        - Description: A brief description of what happens at this step (e.g., "Data read from HTTP request", "Input passed through sanitization function X (which failed)", "SQL query constructed and executed"). This helps readers understand the flow.

        - Conditions (Optional): Any important conditions or decisions that affect exploitability at this step (for example, an if check that was bypassed, or a required configuration setting).

        - Pointer to Evidence: If available, a link to evidence specific to this step (like a screenshot of a debugger at this point, or a log line).

    - 5.4.3 Source Step: Additional notes on representing sources. A source step might include information like the source type (user input, file input, network

input, etc.) and an identifier (e.g., parameter name or source ID).

    - 5.4.4 Sink Step: Additional notes on representing the sink (the final step where the exploit occurs). For sink steps, the trace object might include an indication of the outcome (e.g., “malicious data executes as code here”) or link to evidence of the exploit result.

    - 5.4.5 Multiple Flows and Correlation: Clarifies how multiple distinct exploit paths for the same vulnerability are handled. (For example, if two different sources reach the same sink, are they separate ExploitFlow entries or combined? VXDF could either list them separately or group them under one finding with multiple traces. This specification must define the approach for consistency.)

- 5.5 Additional Data Model Elements: Describes any other core structural elements not covered above, such as:

    - 5.5.1 Property Bags / Extensions: If VXDF allows arbitrary key-value pairs for custom data (similar to SARIF’s property bags or CycloneDX’s properties), the structure and usage of that is defined here. This would enable tools to include tool-specific or additional info without breaking compliance.

    - 5.5.2 Relationships: If there is a need to express relationships between findings (e.g., one exploit depends on another, or a chain of vulnerabilities), the model may provide a way to reference other ExploitFlow entries or external documents. This subsection would outline how to use such references if applicable (for example, linking a validated vulnerability to a broader threat scenario or attack chain).

(Section 5 will be quite detailed in the actual standard, likely with JSON schema excerpts and tables defining each property, similar to the style of SARIF’s “File format” section and CycloneDX’s data model description. Every field will have definitions, data types, and any constraints or requirements.)

6. Schema Definition and Conformance

Overview: Specifies the formal JSON schema for VXDF and the rules for conformance. This section ensures that producers and consumers of VXDF understand how to validate a VXDF file and what it means to be compliant with the standard.

- 6.1 Formal JSON Schema: Introduces the JSON schema document that precisely defines VXDF’s structure. It will likely reference an included schema file (e.g., an appendix or an external URL). This schema is the normative source for field definitions and types. Any special schema conventions (like use of $id, $schema version, etc.) are

noted. For example, it may reference the draft JSON Schema standard used to define it . (The full schema text may reside in Appendix A for readability.)

- 6.2 Conformance Criteria for VXDF Documents: Defines what it means for a VXDF file to be considered valid:
    - It MUST be a well-formed JSON document and MUST adhere to the VXDF JSON schema (all required fields present, data types correct, and any enumerated values within allowed sets).
    - All normative rules described in Section 5 (data model) must be respected. For instance, if the schema allows an array, but the spec text further limits its length or requires certain combinations of fields, those conditions must also be met for full conformance.
    - If a VXDF document contains an extension or custom property (as allowed in Section 9), it MUST NOT violate or alter the meaning of any standard field.
- 6.3 Conformance Requirements for Producers: Outlines expectations for tools or processes that generate VXDF:
    - A producer (tool) MUST output VXDF that conforms to the schema and rules of this standard. It should not introduce its own JSON structure that isn't permitted by the spec (unless using defined extension points).
    - If a producer omits optional sections, the resulting VXDF should still be valid (the schema will permit optional fields to be absent).
    - Producers SHOULD populate all relevant information to maximize usefulness (for example, include evidence and metadata as available, not leave them blank if they can be provided).
- 6.4 Conformance Requirements for Consumers: Outlines expectations for tools that read/ingest VXDF:
    - A consumer tool MUST interpret the data according to this specification's semantics. It should use the evidence and dataflow information to inform its behavior (e.g., a CI gate should check the severity field and evidence before failing a build).
    - Consumers SHOULD ignore or safely handle any unknown extension fields (so that forward compatibility is possible), as long as the document is otherwise valid. They should not reject a VXDF file outright if it has extra data that doesn't affect

core compliance.

    - If a consumer encounters a VXDF file that does not conform (invalid schema), it SHOULD report an error or warning. Conversely, if the file is valid, it MUST NOT throw errors for recognized content .

- 6.5 Compatibility and Profiles (if applicable): (Optional) If the standard foresees different conformance profiles or levels (for example, a minimal VXDF vs. extended VXDF), this would describe them. By default, VXDF 1.0 may define a single conformance level, but this section leaves room for future profile definitions (e.g., a stripped-down VXDF for resource-constrained environments).

- 6.6 Localization and Internationalization: (Optional, if relevant) If the VXDF format contains any human-readable messages or content that might need localization (similar to how SARIF handles message strings), mention how that should be handled. VXDF likely has mostly technical data, but any text fields (like descriptions) could be localized. If so, the schema might allow multi-language representations or references to external resources.

(Section 6 ensures that there is a clear contract for what a valid VXDF file is and how tools interact with it, much like the “Conformance” sections in other standards . It likely references RFC 2119 keywords heavily to set these rules.)

7. Evidence Model and Validation Types

Overview: Defines how proof of exploit is represented in VXDF. This section enumerates the types of evidence that can be included and the structure for capturing that evidence. It emphasizes the proof-centric nature of VXDF, detailing how to document that a vulnerability is not just theoretical but actually exploitable.

- 7.1 Evidence Overview: Explains the purpose of the evidence model in VXDF. Each ExploitFlow (from section 5.3) can link to one or more pieces of evidence. Evidence in VXDF provides the factual basis for claiming a vulnerability is exploitable, whether it’s a snippet of runtime data, a code excerpt, or a procedural proof. This subsection notes that evidence can be of different forms (static vs dynamic, manual vs automated), and VXDF aims to standardize how these are recorded.

- 7.2 Evidence Types: Enumerates the categories of evidence that VXDF supports. Each bullet below corresponds to a category, with a description of what it entails:

    - Static Analysis Evidence: e.g. code snippets or static dataflow graphs. This could be a relevant excerpt of source code or pseudocode showing the vulnerability. For instance, a snippet of code highlighting where user input flows into a

vulnerable function, possibly accompanied by annotations.

    - Dynamic Runtime Evidence: e.g. HTTP requests/responses, console output, log entries, or memory dumps that occurred during exploitation. This might include an HTTP request that was sent to exploit the issue and the response demonstrating success (such as a dump of database records or a specific system response).

    - Proof-of-Concept Script or Steps: e.g. a step-by-step description or script (possibly in a text form) that can reproduce the exploit. This might not be machine-executable within VXDF, but provides a human-readable procedure verifying the exploit. VXDF might include it as a text blob or a link to an external file.

    - Environmental/Configuration Evidence: e.g. information showing that the environment was set up in a way that the exploit is possible. For instance, a certain feature flag was enabled, or a particular security control was disabled during test. This type of evidence indicates that under normal conditions the exploit might or might not work, and what conditions were necessary.

    - Manual Verification Notes: If a human tester verified the issue, this could be a short narrative or note confirming the exploit, possibly including screenshots or references (for example, an expert's commentary that the data flow was inspected and found exploitable, even if an automated tool didn't execute it fully).

- (Each evidence item in VXDF will typically have a type label corresponding to one of the above categories, making it easy for consumers to understand what kind of proof is provided.)

- 7.3 Evidence Object Structure: Defines the schema for an evidence item in VXDF. Likely, evidence items could be collected in a separate array or embedded within each finding. This subsection details the fields of an evidence object:

    - 7.3.1 Evidence ID: A unique identifier for the evidence item, so that it can be referenced from an ExploitFlow (5.3.7). If evidence is embedded directly in the finding, an ID may be optional; if in a separate list, IDs are required for linking.

    - 7.3.2 Type: The category/type of evidence (as outlined in 7.2, e.g., "static-code", "http-request", "poc-script", etc.). This could be an enumeration of allowed types.

    - 7.3.3 Description: A short text describing what the evidence is and its significance. For example, "HTTP response showing user data exfiltration" or "Stack trace confirming code execution".

    - 7.3.4 Content: The actual evidence content. Depending on type, this could be:
        - A text blob (for example, the raw HTTP request and response, or a code snippet).
        - A file reference or URI (if the evidence is too large or binary, such as a pcap file or a screenshot image, VXDF might allow linking to an external resource or embedding a base64 string).
        - A structured sub-object (for example, a key-value map for specific data points, or an array if the evidence has multiple parts).
    - 7.3.5 Associated Trace Step: Optionally, a reference to which step in the data flow (section 5.4) this evidence is associated with, if applicable. For instance, a log entry might be linked to the sink step to show the impact at that point.
    - 7.3.6 Reproducibility: (Optional) An indicator of how the evidence was obtained (automated vs manual) or how easily the exploit can be reproduced. This could be a simple flag or rating. For example, something akin to a boolean `“automated”: true if a tool automatically produced this evidence, or a text like “manual” if a human step was involved.
    - 7.3.7 Integrity Protection (Optional): If there is a need to ensure evidence integrity, this could include a hash of the evidence content or a signature (though digital signing of the whole VXDF file might be covered elsewhere, an individual evidence item could also carry a checksum especially if stored externally).
- 7.4 Validation Methods and Outcomes: Describes the different validation methods that might be recorded and how to indicate them:
    - 7.4.1 Method Classification: VXDF may allow tagging each ExploitFlow or evidence with how it was validated: e.g., "validationMethod": "DAST" or "SAST+manual" etc. This subsection would list acceptable values or schemes for indicating whether the exploit was confirmed by a dynamic test, static analysis reasoning, manual pen-testing, or a combination. It formalizes the notion of validation type (SAST, DAST, SCA, Manual, Hybrid).
    - 7.4.2 Confidence Level: Although VXDF by design includes only confirmed exploits, there may be a field for degree of confidence or validation strength. For example, an exploit validated by a fully automated test might be marked as “Proof-of-Concept Executed”, whereas one validated by expert code review might be “Theoretical Proof” but still high confidence. This could help consumers differentiate evidence robustness. If included, this section defines the scale or categories (e.g., “Confirmed-Exploited”, “Confirmed-Not-Exploited”,

"Not-Reproducible", though likely VXDF only holds confirmed exploits, so perhaps a simpler binary or omission for anything not confirmed).

- 7.4.3 False Positive Handling: In principle, VXDF should not contain false positives at all. This subsection might state that any item in VXDF is assumed to be a true positive. If an item is later found to be a false positive, it should be removed from VXDF or marked as such in an updated run (rather than having a field marking it false within VXDF). Essentially, it clarifies that VXDF's scope is confirmed issues only, so it doesn't represent non-exploitable cases (those are handled via VEX or simply absence from the report).

(Section 7 provides the schema and guidelines for including evidence. It mirrors the rigor of how CycloneDX, for example, defines complex sub-objects like "evidence" and "affects" in its vulnerability model, tailoring it to exploit proof data. The presence of evidence is what sets VXDF apart from simpler vulnerability listings, so this section is critical.)

8. Interoperability and Mapping (SARIF, SPDX/SBOM, SCA)

Overview: Describes how VXDF interacts with other standards and tool outputs. VXDF is not created in a vacuum – it often will be generated from or used alongside existing formats. This section provides guidelines for mapping information to and from VXDF, ensuring it can be integrated smoothly into workflows involving SARIF, SBOMs (SPDX/CycloneDX), and vulnerability management.

- 8.1 Mapping from SARIF (Static Analysis Results): Provides a strategy to transform or augment SARIF results into VXDF format. For example:
  - How a SARIF result with a code flow (if the static tool provides a code flow trace) would correspond to a VXDF ExploitFlow (5.3). This might include mapping SARIF's ruleId or result.message to VXDF's vulnerability description, SARIF result.locations to the data flow steps, etc.
  - If SARIF results have suppression or baselineState information, note whether that is relevant (VXDF likely only includes unsuppressed, confirmed issues).
  - Provides an example of taking a SARIF file and producing a VXDF output once the static results are validated. (Perhaps a small table of SARIF field -> VXDF field mappings could be included for clarity.)
  - Emphasizes that SARIF can hold many results including false positives, whereas VXDF will contain a filtered subset that have proof. Tools may output both: a full

SARIF report and a VXDF report, serving different purposes.

- 8.2 Mapping from SCA/SBOM (SPDX or CycloneDX): Describes how vulnerability information from Software Composition Analysis and SBOM standards can be related to VXDF:
    - For an SPDX or CycloneDX BOM that lists vulnerabilities for components, VXDF can be used to detail which of those vulnerabilities are actually exploitable in the system. This subsection might outline how to reference a component in VXDF: for example, using a Package URL (PURL) or CycloneDX BOM Reference to identify the component in question .
    - If CycloneDX's vulnerability model is considered, how fields like analysis justification or impact could correlate with VXDF's evidence. For instance, CycloneDX has a field for exploitable or affects, which VXDF would substantiate with actual data flows and proofs.
    - Provide guidance that an SBOM could include an external reference to a VXDF file (or even embed VXDF data as an extension), to enrich the SBOM with exploit details. For SPDX, perhaps using their extension mechanisms or linking via vulnerability identifiers.
    - Mentions that when an SCA tool finds a CVE, an additional step (like static reachability analysis) can produce a VXDF entry. This two-step process (find via SCA, confirm via static/dynamic analysis) can be standardized so that the SCA finding gets an annotation like "confirmed_exploitable: true" and is backed by a VXDF reference.
- 8.3 VEX (Vulnerability Exploitability eXchange) Integration: Explains how VXDF supports or complements VEX documents:
    - VEX is typically a simple statement of whether a product is affected by a known vulnerability or not (and maybe why or under what conditions). VXDF can serve as the detailed evidence behind a VEX affected status. For example, if a VEX says Product X is affected by CVE-1234, the VXDF document can be cited to show exactly how CVE-1234 is exploited in Product X. Conversely, if Product X is not affected by CVE-1234, there would be no VXDF entry, and possibly the analysis notes (in VEX or elsewhere) would explain that no exploit path was found.
    - If a VEX format (like CSAF or CycloneDX VEX) allows linking to evidence or notes, VXDF could be referenced as supporting material. This subsection may provide a pattern for including a VXDF reference in a VEX statement (e.g., via an

URL or an attached JSON snippet).

    - Clarifies that VXDF is primarily about positive evidence (the vulnerability is exploitable). For negative assertions (not exploitable), VXDF's role is indirect (documenting the attempt or analysis might be outside VXDF, or simply the absence of VXDF entries is itself an indicator when everything else was in scope).

- 8.4 Relationship to CVSS and Other Risk Scores: (Optional but likely informative) Discusses how the data in VXDF can inform scoring systems like CVSS:

    - CVSS scoring has environmental and temporal metrics (Exploit Code Maturity, Remediation Level, etc.). A confirmed exploit path in VXDF effectively demonstrates that exploit code exists (at least a proof-of-concept), which could correspond to a higher Exploitability score (e.g., CVSS Exploit Code Maturity = High or Functional). VXDF could thus be used to justify setting those values in a risk assessment.

    - If organizations maintain a risk register or use other frameworks (OWASP Risk Rating, etc.), VXDF evidence provides the concrete info needed for impact analysis. This subsection might remain high-level, as CVSS integration is not a direct file format mapping but a usage consideration.

(Section 8 ensures VXDF is not isolated. By providing clear mapping guidelines, it helps implementers merge VXDF into existing security programs. The structure here mirrors how other standards provide annexes or sections on interoperability – for example, SARIF spec might discuss how to embed SARIF in other systems, and CycloneDX provides mappings to SPDX. VXDF will similarly ensure it can slot into the ecosystem.)

9. Extension and Versioning Mechanisms

Overview: Defines how VXDF can be extended for custom use cases and how versioning of the specification is handled. This section is crucial for ensuring the standard can evolve and be adapted without breaking existing implementations, much like CycloneDX and SARIF allow custom fields and handle new versions gracefully.

- 9.1 Extensibility: Describes the provisions for adding custom data to a VXDF document beyond the defined schema, in a way that won't conflict with standard fields:

    - VXDF may designate a specific object or property (e.g., a "properties" dictionary) where tools can add arbitrary name/value pairs. This is similar to SARIF's concept of property bags and CycloneDX's properties element, allowing

vendor-specific or additional info.

    - Rules for extensions: for example, custom properties SHOULD use a unique naming convention (perhaps prefixing with a vendor or tool name) to avoid collisions. They MUST NOT alter the semantics of required VXDF fields. Consumers are expected to ignore any extension fields they don't recognize (as noted in 6.4).

    - Possibly an example: a tool might add an extension field like "myToolSpecialMetric": 42 under a finding's properties to convey extra info not in the core spec. This section would reassure that doing so is allowed as long as the VXDF schema is designed to permit it.

    - Mentions that future official extensions or profiles could be introduced, but those would be incorporated via the versioning mechanism (not as ad-hoc fields).

- 9.2 Versioning Scheme: Outlines how versions of VXDF are managed and indicated:

    - 9.2.1 Version Identification: The VXDF version is included in each document (see 5.1.1). This subsection states the format (e.g., semantic versioning "1.0", "1.1", etc.) and how it should be interpreted. For instance, a major version change might introduce non-backwards-compatible changes, whereas minor/patch versions are backwards-compatible enhancements or fixes.

    - 9.2.2 Compatibility Guarantees: Explains the expectations of compatibility. For example: "VXDF 1.x consumers should be able to read any 1.x VXDF document, ignoring fields from later minor versions that they do not understand." If a VXDF 2.0 comes out, it might not be compatible with 1.x, hence requiring updates to tools.

    - 9.2.3 Deprecation Policy: If certain fields or constructs are to be deprecated in future versions, how will that be handled? This likely states that deprecated fields will be marked in the spec, possibly still allowed for a period for backwards compatibility, and eventually removed in a major release. It encourages producers to move away from deprecated usage as soon as possible.

    - 9.2.4 Version Negotiation: (If relevant) In scenarios where systems exchange VXDF, how to handle version mismatches. Perhaps a note that if a consumer encounters a newer version, it should attempt a best-effort parse or report the incompatibility gracefully.

- 9.3 Profiles or Levels (Optional): If VXDF might have subsets or profiles (for example, a "Core VXDF" vs. an "Extended VXDF with additional fields"), this section would outline that concept. In v1.0 there may be no profiles, but it leaves the door open. (Alternatively,

this could be mentioned under extensibility or versioning as needed.)

- 9.4 Registration of Extensions (Optional): If the community develops common extensions (like industry-specific fields), will there be a registry or listing? For instance, OWASP could maintain a list of known extension keys that others can reuse. This is more of a governance detail but could be touched upon: encouraging collaboration on extensions rather than everyone inventing their own for the same thing.

(Section 9 is akin to those in other standards where they clarify how to extend the data model and how new versions come out. This ensures the spec is not static and can adapt to future needs while maintaining interoperability.)

10. Tooling and Reference Implementations

Overview: Discusses the ecosystem around the VXDF standard – reference implementations, libraries, or tools that support VXDF, as well as guidance for tool developers. This section may not define the standard itself, but it aids adoption by pointing to resources and examples.

- 10.1 Reference Parser/Library: Indicates that a reference implementation (or skeleton) is provided to handle VXDF data. For instance, an open-source library that can read and write VXDF JSON, validate against the schema, and maybe convert from other formats (like SARIF) to VXDF. If this exists (or is planned), it would be described here. The goal is to lower the barrier for tool vendors to implement VXDF support.

- 10.2 CLI Tools and Utilities: Mentions any command-line or utility tools provided. For example, a tool that can take two VXDF files and diff them (to see what new exploits appeared), or a tool to pretty-print or visualize VXDF content (perhaps generating a PDF or HTML report from a VXDF JSON). Another example: a CI plugin that reads a VXDF file and enforces a policy (failing build on certain conditions).

- 10.3 Integration Guides: Provides guidance or references to how common security testing tools could integrate VXDF:

    - For SAST vendors: how to produce VXDF from their existing results (maybe via a SARIF-to-VXDF adapter or building evidence collection into their tools).

    - For DAST tools: suggestions on capturing requests/responses as evidence for VXDF.

    - For SCA tools: how to incorporate static analysis to verify exploitability and output VXDF (possibly working in tandem with SAST).

    - This subsection might not have normative content, but gives practical advice or points to external guides or examples.

- 10.4 Example Implementations: If any early adopters or prototypes exist (for instance, an OWASP project or a particular vendor that implemented VXDF in their scanner), mention them as case studies. This helps validate the standard and provide real-world context. (E.g., “OWASP SuperScan tool outputs VXDF alongside SARIF to provide exploit validation,” or a reference implementation integrated into OWASP Zap for DAST results.)

- 10.5 Testing and Certification: Describes whether there is a test suite or certification program for VXDF implementations. For example, an official set of VXDF example files and expected outcomes that tool developers can use to verify their implementation (this could tie into Appendix B: Test suite). It could also mention if there’s an option for tools to be formally certified as “VXDF-compliant” in the future, encouraging quality and consistency.

- 10.6 Community Resources: Points readers to community-driven support, such as a GitHub repository for the VXDF standard, forums or chat channels for discussion, and contribution guidelines. While more of a governance detail, it’s useful here to encourage tool makers and users to engage with the VXDF project (report issues, suggest features, etc., as part of the open development approach).

(Section 10 is largely informative. It mirrors how standards like CycloneDX often provide tooling or usage notes. Since VXDF is initially an OWASP-style open spec, this section emphasizes community and practical adoption support to ensure the standard doesn’t just live on paper but is used in the field.)

11. Security and Privacy Considerations

Overview: Highlights the security and privacy implications of using VXDF. Because VXDF deals with vulnerability details and proof of exploits, it’s important to handle VXDF files with care. This section provides guidance to avoid introducing new risks when creating, storing, or sharing VXDF data.

- 11.1 Sensitive Data Exposure: VXDF evidence may include sensitive information (e.g., database dumps, user data, system details). Producers SHOULD avoid including any credentials, personal data, or other sensitive info unless necessary for the evidence. If such information is included (perhaps unavoidably, as part of exploit proof), consumers must treat the VXDF file as sensitive. Consider data minimization or redaction strategies in the evidence (for instance, mask usernames or use synthetic data in proofs where possible).

- 11.2 Handling of Exploit Artifacts: VXDF might contain actual exploit payloads or scripts. There is a risk that these could be misused if in the wrong hands, or accidentally triggered. Guidance: treat VXDF files as you would vulnerability details – share on a need-to-know basis. If the VXDF includes an exploit script, perhaps store it encrypted or ensure it's only run in safe test environments. Producers may opt to provide a descriptive narrative of an exploit rather than a live exploit code if there's a concern.

- 11.3 Access Control and Storage: Recommends that VXDF reports be access-controlled in repositories or tracking systems. Because a VXDF is essentially a map to exploit an application, it can be a blueprint for attackers if obtained. Organizations should restrict who can view VXDF outputs (especially before the vulnerabilities are fixed). Also, consider secure storage (e.g., not in public CI logs or artifacts by default).

- 11.4 Integrity and Authenticity: Emphasizes the importance of ensuring VXDF data is not tampered with. Since decisions (like CI gating or risk acceptance) might be based on VXDF, an attacker might attempt to alter a VXDF file (e.g., remove a finding or fake evidence). It's recommended to use digital signatures or other integrity checks for VXDF files, especially when transferring between systems. (For example, signing the VXDF JSON or its important elements, similar to how CycloneDX supports a BOM signature, could be a practice to adopt in future versions.)

- 11.5 Privacy of Individuals: If any part of the exploit validation involves user accounts or testers, ensure no personal identifiable information (PII) of those individuals is recorded without consent. For instance, if a tester's admin account was used to demonstrate an exploit, don't include their actual username and password in the VXDF; instead just note it was done with a test account.

- 11.6 Misuse of VXDF Files: A caution that while VXDF is intended for defensive use (developers and security teams fixing issues), it could potentially be misused by malicious actors as a "how-to exploit" guide. This reiterates the need for careful distribution. It also means producers might sometimes omit certain evidence details if they deem it too dangerous to include (balance between proof and risk).

- 11.7 Security of Tools Producing VXDF: As a meta-consideration, tools generating VXDF should themselves be secure, since they handle exploit data. There should be awareness that generating a VXDF (especially during dynamic testing) should not accidentally harm the system (for instance, executing a payload that has side effects). This might be out of scope of the spec, but a note can be given to ensure tool developers follow safe practices (like cleaning up any changes made during an exploit test).

(Section 11 is analogous to the "Security Considerations" section seen in many standards (including IETF RFCs and others). It's largely advisory but important for the safe adoption of

VXDF. Given the nature of VXDF, this section is particularly important to prevent the standard from inadvertently causing security issues.)

12. Media Type and File Format Definitions

Overview: Defines how VXDF documents are identified and packaged at a file level, including any official file extension, MIME media type registration, and encoding considerations. This ensures that VXDF files are recognized and handled correctly by tools and systems (for example, for content negotiation or file association).

- 12.1 Media Type (MIME): Specifies the official media type string for VXDF JSON documents. For example, it might use a vendor tree or standards tree type such as application/vnd.owasp.vxdf+json (if seeking IANA registration later) or simply application/json with a profile parameter. This section would indicate the intended registration (e.g., “VXDF will register a MIME type for easier identification of VXDF files”). It may draw parallels to SARIF’s media type (which is application/sarif+json) and CycloneDX (application/vnd.cyclonedx+json for JSON SBOM).

- 12.2 File Extension: Recommends a conventional file extension for VXDF files (for example, .vxdf.json or simply .vxdf). Since VXDF is JSON, the extension might be .json, but to distinguish it, using a specific extension like .vxdf could help practitioners immediately identify the file type. This section will settle on a recommended practice (and tools should default to that when generating files).

- 12.3 Character Encoding: States that VXDF files MUST be encoded in UTF-8 (which is default for JSON, but it’s good to be explicit). If there are any Unicode normalization considerations for the content (likely minimal, since it’s mostly data and some text).

- 12.4 Line Endings and Formatting: (Informative) Since JSON is whitespace-agnostic, producers can format the output for readability. This might mention that examples in the spec are pretty-printed with two-space indentation (for instance), but actual files can be minified or pretty – it doesn’t affect conformance. If any normative requirement exists (probably not, except maybe that the file should not include binary data outside of allowed JSON encoding, which is covered by JSON rules).

- 12.5 Bundling and Packaging: If a VXDF document needs to be bundled with other files (like large evidence artifacts), how is that handled? Possibly out-of-scope, but could mention that large evidence could be packaged separately (e.g., a zip file with a VXDF and associated evidence files). If relevant, provide a suggestion or reference (maybe something like using the ZIP-based packaging approach as used by some standards).

- 12.6 Multiple Document Handling: Clarifies whether multiple VXDF reports can be concatenated or combined. Likely not (each VXDF is a self-contained JSON). But if there’s any need to aggregate, that would be up to external tooling (for example,

merging VXDFs from multiple tools into one big VXDF – which would just result in one larger findings array). Probably just note that one JSON file = one report for a given scope.

(Section 12 aligns with how other standards define their file type. It gives VXDF an identity in file systems and network protocols. Following the example of SARIF and CycloneDX, it's important to have a clear MIME type and extension so that tools and users can quickly recognize VXDF files.)

13. Governance and Proposal Lifecycle

Overview: Describes how the VXDF specification is managed, how changes are proposed and accepted, and the community process behind it. This reflects the open nature of VXDF's development (initially OWASP-driven) and provides a path toward future standardization.

- 13.1 Open Collaboration Model: Explains that VXDF is developed as an open project (e.g., under OWASP or a similar foundation). Contributions are welcome from the community. This section might describe the use of a public version control (like a GitHub repository) where issues and pull requests can be submitted for spec changes. It underscores transparency in the development process.

- 13.2 Proposal Process: Outlines the steps to propose changes or enhancements to VXDF. For example, a contributor might write a proposal or an RFC-like document, discuss it on community forums or meetings, and then it gets reviewed by maintainers or a working group. It could mirror processes from other open standards (for instance, how the SARIF standard had a committee in OASIS – here it might be an OWASP working group).

- 13.3 Version Release Cycle: Describes how often and how new versions of the spec are released. Perhaps initially VXDF might have frequent minor updates as it matures, with an aim to stabilize at 1.0 and then follow semantic versioning. If there's an official roadmap or milestones (like targeting an OASIS or ISO standardization in the future), it can be mentioned. For example: "VXDF 1.0 is expected to be finalized through the OWASP project process, after which the project may contribute it to a standards organization for further ratification."

- 13.4 Governance Body: If one exists, identify the group responsible for decision making. This could be a core team or a technical committee. If under OWASP, possibly the project leaders; if it transitions to a formal body, mention that intent. Also note any liaisons with other standards bodies (maybe working with CycloneDX team for SBOM integration, or with OASIS SARIF maintainers for alignment).

- 13.5 Intellectual Property and Licensing: States that the VXDF specification is published under an open license (for example, Creative Commons or an Apache 2.0 license for documentation). This ensures that the standard can be freely used and implemented. (While we are not including an actual IPR policy text like OASIS does, it's good to reassure that VXDF is meant to be an open standard without encumbrances.)

- 13.6 Future Evolution: Notes any anticipated future directions, such as potential incorporation of VXDF into regulatory frameworks or broader security standards. Also mentions that as the ecosystem evolves (new types of evidence, new tool categories), the governance process will adapt the VXDF standard accordingly. Essentially, a commitment that VXDF will remain up-to-date with the threat landscape and user needs through the collaborative process.

(Section 13 is important for those who want to contribute or understand the longevity of the standard. It mirrors, in spirit, the forewords or process sections in formal standards (though in less formal terms since it's an open project). By including this, we signal that VXDF is professionally managed and here to stay, with community buy-in.)

14. Appendices

Overview: Additional supporting material, provided in appendices for clarity. These sections include the full normative schema, example documents, and other supplementary information that would clutter the main text but are essential for implementers. Appendices are typically non-normative unless otherwise noted (the JSON Schema may be considered normative).

- 14.1 Appendix A: VXDF JSON Schema (Normative) – Contains the complete JSON schema definition for VXDF. This is the authoritative machine-readable specification of the format. It will list all object definitions, properties, types, required fields, default values (if any), and constraints (pattern regex, value ranges, enumerations, etc.). Having the schema in an appendix allows implementers to refer to it in one place. (If the schema is too large, this appendix might reference an external file URL, but ideally a copy is included here for completeness.)

- 14.2 Appendix B: Conformance Test Suite Description – Describes a set of test cases or example VXDF instances used to verify implementations. For example, it might outline a few scenarios:

    - A VXDF file with one simple exploit flow (to test basic compliance).

    - A VXDF file with multiple findings, various evidence types (to test complexity).

    - Files that intentionally break certain rules (to test that validators catch errors).

This appendix would not list all test files in full, but summarize what areas they cover and possibly where to find them (e.g., linking to a public repository of JSON examples and expected outcomes). It might also describe how to run a validation (e.g., using the JSON schema with a validator tool).

- 14.3 Appendix C: Example VXDF Documents – Provides one or more full examples of VXDF in practice, likely in pretty-printed JSON form with explanatory comments. Each example could correspond to one of the use case scenarios from section 3. For instance:
    - Example 1: A simple web app SQL injection validated by DAST, showing the data flow and an HTTP request evidence.
    - Example 2: A vulnerable library usage validated by static analysis, showing a code trace and linking to a CVE.

      These examples help readers and implementers understand concretely how the spec is used. The appendix would label parts of the JSON and possibly include commentary on each section of the example.

- 14.4 Appendix D: Glossary of Terms and Acronyms – A consolidated glossary of terms used in the document. While section 2.2 defines terminology inline, this appendix provides an easy reference list in alphabetical order. It would include terms like SAST, DAST, VXDF, SBOM, CVSS, etc., along with brief definitions. This reduces confusion and serves as a quick lookup. (It may also include any terms introduced later or abbreviations that weren't spelled out in earlier sections.)

- 14.5 Appendix E: Change Log (Optional) – If the specification undergoes revisions, a log of changes by version can be maintained here (e.g., what changed from draft to 1.0, or between minor versions). This helps implementers quickly identify what's new in each revision without re-reading the entire spec. This is more useful once multiple versions exist.

- 14.6 Appendix F: References (Optional) – Lists any external documents referenced in the specification. This could include RFCs (like RFC 2119 for normative language, RFC 3986 for URIs, etc.), other standards (SARIF spec, CycloneDX spec, CVSS documentation), and relevant publications. Each reference would be cited in the main text where appropriate. (If this spec follows a format like OASIS or others, normative and non-normative references might be split; but as an open spec, a single references list might suffice.)

(The appendices provide all the detailed resources needed to implement and understand VXDF fully. By mirroring the thoroughness of SARIF's annexes and CycloneDX's additional content,

we ensure VXDF documentation is self-contained and implementer-friendly. Not every reader will go through the appendices, but they are crucial for completeness and precision.)

End of VXDF Standard Blueprint