Definitive Guide to Open Cybersecurity Schema Framework (OCSF) Mapping

Contributed by: Jonathan Rau & Aurora Starita

Table of Contents

Introduction to OCSF

The Open Cybersecurity Schema Framework (OCSF) is an open-source and collaborative effort across the industry to **define a vendor- and platform-agnostic schema for security and IT observability data**. It has been contributed to by Query, Amazon Web Services (AWS), Splunk, Cisco, Crowdstrike, and several dozen other organizations and individuals.

The OCSF schema provides **standardization and normalization of data into a hierarchical data model** that can be used in Security Information & Event Management (SIEM), Extended Detection & Response (XDR), security data lakehouses, and much more for analytics, just-in-time querying, machine learning, and effective storage of data.

It can be overwhelming for newcomers to learn about the schema and all it entails. The best place to start is with our beginner's guide or the white paper Understanding the Open Cybersecurity Schema Framework by Paul Agbabian at Splunk, or for a navigable experience, the official OCSF Schema Server.

Finally, to find out why Query.ai uses the OCSF as our data model for normalization and searching see our blog post: OCSF: Why We Choose It for Query by Jeremy Fisher, CTO at Query.

We realize that not everyone learns in the same way. While reading and navigating the schema can help some, starting from a known good point is also helpful. Refer to the Mappings GitHub repo for examples of various security systems mapped into the OCSF schema by the community.

However, this paper will teach you how to map from start to finish. We will focus on the **theory of mapping**, teach you some **intricacies of how to map data**, why **mapping data into OCSF is important**, and provide a deeper understanding of **how the OCSF data model works**.

By the end, you will understand the nuances and the “minimum necessary” concept of **mapping into OCSF**.

Yapping About Mapping

The concept of "mapping" speaks to **transforming a raw log, event, or finding from a source platform into the OCSF schema**. You "map" the data by determining which OCSF attribute(s) your source data will fit into. Using various libraries in your language of choice—such as Python, Golang, Rust, or otherwise—you parse the upstream data into the final OCSF event.

Another term used is **"producing"** OCSF or being an OCSF **"producer"**, and this refers to security and IT observability tools providing output in native OCSF format by default.

Seems simple right? Yo, not so fast. There is a lot more nuance than just simply trying to find the best fit, as multiple attributes can use the same upstream data, not to mention the different data types in the OCSF schema as well. As noted, OCSF provides both ***normalization*** and ***standardization***.

Normalization

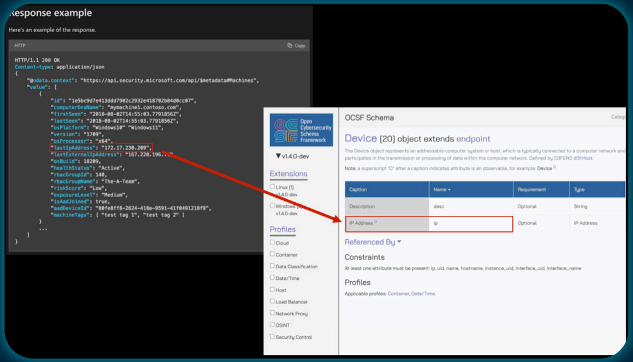
Normalization is provided by virtue of the schema itself. It provides **generalized attributes that can accept many mappings**. For instance, the OCSF attribute of `ip` inside of the Device object (a collection of thematically related attributes) can be used to normalize the IP address data of a device.

In this case, **Device** can refer to a laptop or desktop record in an Configuration Management Database (CMDB), it can refer to a Desktop-as-a-Service (DaaS) virtual workstation such as an AWS WorkSpaces instance or Amazon AppStream 2.0 instance, or it can refer to a virtual machine in VMWare eSXI.

The `ip` attribute can be mapped into from several different upstream keys such as "IPAddress", "IPV4_Address", "ip_address", "lastIpAddress", "DeviceIp", "PrivateIp", and any other variation.

Normalization is important because this allows for predictable and repeatable patterns of analysis, querying, and/or visualization of the results and makes it easier for mapping the data.

Likewise, **it makes mapping or producing OCSF repeatable**. Within the schema there are descriptions provided for every attribute to help consumers, mappers, and producers understand the intended usage as well.



```
HTTP/1.1 200 OK
Content-type: application/json
{
    "@odata.context": "https://api.security.microsoft.com/api/$metadata#Machines",
    "value": [
        {
            "id": "1e5bc9d7e413ddd7902c2932e418702b84d0cc07",
            "computerDnsName": "mymachine1.contoso.com",
            "firstSeen": "2018-08-02T14:55:03.7791856Z",
            "lastSeen": "2018-08-02T14:55:03.7791856Z",
            "osPlatform": "Windows10" "Windows11",
            "version": "1709",
            "osProcessor": "x64",
            "lastIpAddress": "172.17.230.209",
            "lastExternalIpAddress": "167.220.196.",
            "osBuild": 18209,
            "healthStatus": "Active",
            "rbacGroupId": 140,
            "rbacGroupName": "The-A-Team",
            "riskScore": "Low",
            "exposureLevel": "Medium",
            "isAadJoined": true,
            "aadDeviceId": "80fe8ff8-2624-418e-9591-41f0491218f9",
            "machineTags": [ "test tag 1", "test tag 2" ]
        }
        ...
    ]
}
```

OCSF Schema

Device [20] object extends endpoint

The Device object represents an addressable computer system or host, which is typically connected to a computer network and participates in the transmission or processing of data within the computer network. Defined by D3FEND d3f:Host.

Note: a superscript "O" after a caption indicates attribute is an observable, for example: Device O.

Caption	Name	Requirement	Type
Description	desc	Optional	String
IP Address O	ip	Optional	IP Address

Referenced By

Constraints

At least one attribute must be present: ip, uid, name, hostname, instance_uid, interface_uid, interface_name

Profiles

Applicable profiles: Container, Date/Time.

Standardization

Standardization is important to provide **more predictability and repeatability**, but also, to act as a **mechanism to validate the schema** against. This Is useful for ensuring that public contributions to OCSF are valid, but also helps to confirm that mappings are done correctly.

With both normalized and standardized schema elements in mind, it becomes simpler to write detection content or other queries against data mapped to OCSF given the strongly typed and verifiable data model.

Instead of accounting for the various ways severity can be represented—numerically or with strings, with many variations—it's simpler to write against a numerically scaled and normalized attribute, such as `severity_id`.

Likewise, when looking for a specific device by its IP, it's much simpler to standardize against `device.ip`, instead of the many variations and locations IP data can be placed in downstream datasets.

OCSF Basics for Mapping

Attributes

At the simplest level, OCSF is a **collection of key-value pairs that are called attributes**. Every attribute has a human readable Caption and an actual "name" which is the Key.

For example, the OCSF attribute `message` is captioned as "Message" and `severity_id` is captioned as "Severity ID". The value of the attribute is whatever the upstream data source value is. You do not use the Captions of attributes when you are mapping data.

Another note on standardization: Standardization provides consistency on how the data is represented, specifically the data types and formats.

For example, the aforementioned `message` attribute is defined as a string whereas `severity_id` is an integer which is defined by an enumeration (enum). The enum correlates integers with a specific value such as 1 stands for Informational and 6 stands for Fatal.

Every single attribute in OCSF has a data type and some can be more complex such as an array of strings or string-based enumeration. Both the OCSF Server and the actual OCSF dictionary.json define these types.

An important metaschema point to note is that **attributes can also refer to other objects as their data type**. For the most part, attributes are scalar (strings, integers, booleans, floats), but attributes can also use an object as a data type.

```
{
"message": "this is my OCSF event!",
    "severity_id": 1
}
```

Objects

Objects can most easily be thought about as literal JSON objects (or Python dictionaries) which is a literal data type that contains a collection of key-value pairs and is denoted with curly braces (`{}`).

Event Classes (which we'll cover more thoroughly shortly) define the intended normalization and standardization of generic security data and are made up of attributes and objects. Like the Event Class, objects are aligned to a "thing" or "entity" that is encountered within security or IT observability data.

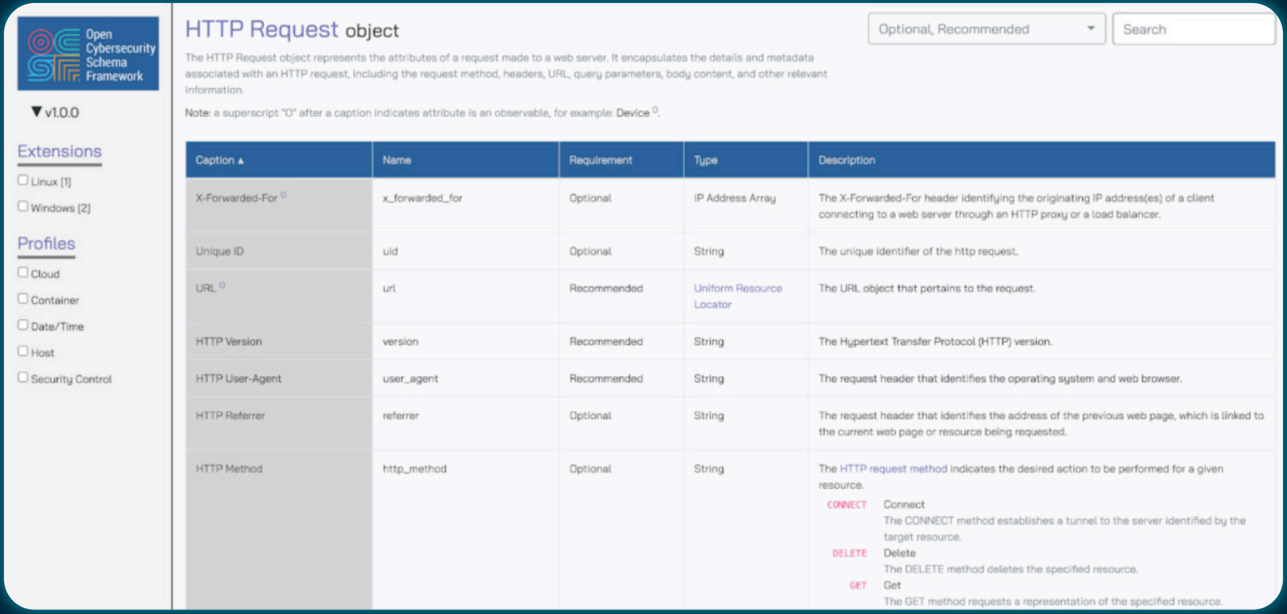
For instance, the Device object is a collection of attributes to represent any device (workstation, server, laptop, VM, DaaS, etc.) such as the IP address (`ip`), the ownership data (`owner`, which is in turn its own object), the Unique ID (`uid`), or an Active Directory Common Name (`uid_alt`).

There are several objects within the HTTP Activity event class, such as the HTTP Request object which defines common attributes seen in HTTP requests such as the method (`http_method`) or the User Agent (`user_agent`).

The important thing to remember is that the attributes—either directly inside of the Event, or within an object—**are not exclusively mapped**.

You may have upstream data such as an IP address or a unique ID of a connection such as a trace ID or other GUID that can be used in other objects.

For instance, you could use the Trace ID within AWS ALB logs and map it to the Unique ID (uid) attribute inside of the Metadata object, the Connection Info object, and the HTTP Request object.



HTTP Request object

The HTTP Request object represents the attributes of a request made to a web server. It encapsulates the details and metadata associated with an HTTP request, including the request method, headers, URL, query parameters, body content, and other relevant information.

Note: a superscript "O" after a caption indicates attribute is an observable, for example: Device O.

Caption ▲	Name	Requirement	Type	Description
X-Forwarded-For O	x_forwarded_for	Optional	IP Address Array	The X-Forwarded-For header identifying the originating IP address(es) of a client connecting to a web server through an HTTP proxy or a load balancer.
Unique ID	uid	Optional	String	The unique identifier of the http request.
URL O	url	Recommended	Uniform Resource Locator	The URL object that pertains to the request.
HTTP Version	version	Recommended	String	The Hypertext Transfer Protocol (HTTP) version.
HTTP User-Agent	user_agent	Recommended	String	The request header that identifies the operating system and web browser.
HTTP Referrer	referrer	Optional	String	The request header that identifies the address of the previous web page, which is linked to the current web page or resource being requested.
HTTP Method	http_method	Optional	String	The HTTP request method indicates the desired action to be performed for a given resource. `CONNECT` Connect The CONNECT method establishes a tunnel to the server identified by the target resource. `DELETE` Delete The DELETE method deletes the specified resource. `GET` Get The GET method requests a representation of the specified resource.

Categories, Event Classes, and Extensions

Zooming out on the schema, when you map there are some themes to keep in mind. OCSF at the top-most level is organized by **Categories** which are containers of similar **Event Classes** (sometimes called an Event or a Class).

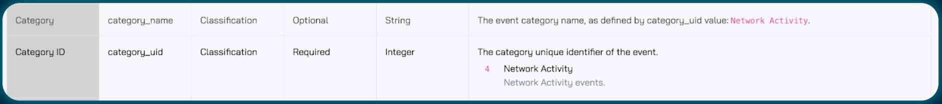
The Event Classes **define the intended usage for normalization**, such as the HTTP Activity event class intended to normalize and standardize any HTTP data, logs, or events from an upstream source.

For instance, you can normalize Web Application FIrewalls (WAF), Next-Gen Firewalls (NGFW), Intrusion Detection Systems (IDS) with HTTP rulesets, layer 7 load balancer logs, and anything else with HTTP traffic in it to this Event Class.

The Categories **group like Events**, and so the Network Activity category contains the HTTP Activity event class but it also contains others such as SSH Activity and DNS Activity.

Similarly, the Identity & Access Management category contains Authentication and Group Management, the Discovery category contains several Inventory Information event classes, and so on.

The Category is reflected in the `category_uid` and `category_name` attributes which can be helpful to query or report on using visualizations to quickly identify the grouped Events.



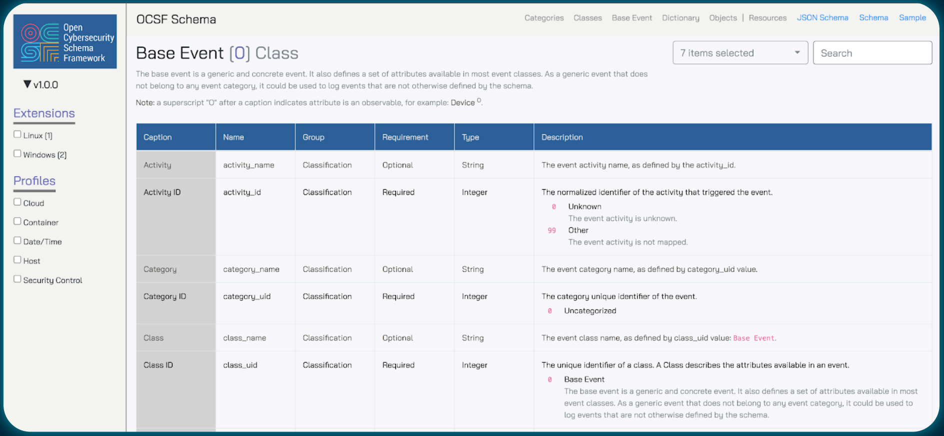
Category	category_name	Classification	Optional	String	The event category name, as defined by category_uid value: `Network Activity`.
Category ID	category_uid	Classification	Required	Integer	The category unique identifier of the event. 4 Network Activity Network Activity events.

The Event Classes define the actual schemata for downstream logs, events, findings, alerts, incidents, or any other "happening" worth recording and mapping.

On top of being very hierarchical and strongly typed, OCSF has another important feature for its scope: **extensibility**. The concept of extensibility is defined within the metaschema that allows other parts of the schema to be referenced and ***extended***.

For example, the **"Base Event"** is the genesis point for every other Event Class within the schema, and it defines the basic attributes and objects that encompass all Event Classes.

Common attributes such as the `category_id`, `message`, `severity_id`, and several other attributes are present here as well as specialized objects such as enrichments and observables.



Base Event [0] Class

The base event is a generic and concrete event. It also defines a set of attributes available in most event classes. As a generic event that does not belong to any event category, it could be used to log events that are not otherwise defined by the schema.

Note: a superscript "O" after a caption indicates attribute is an observable, for example: Device ^{O}.

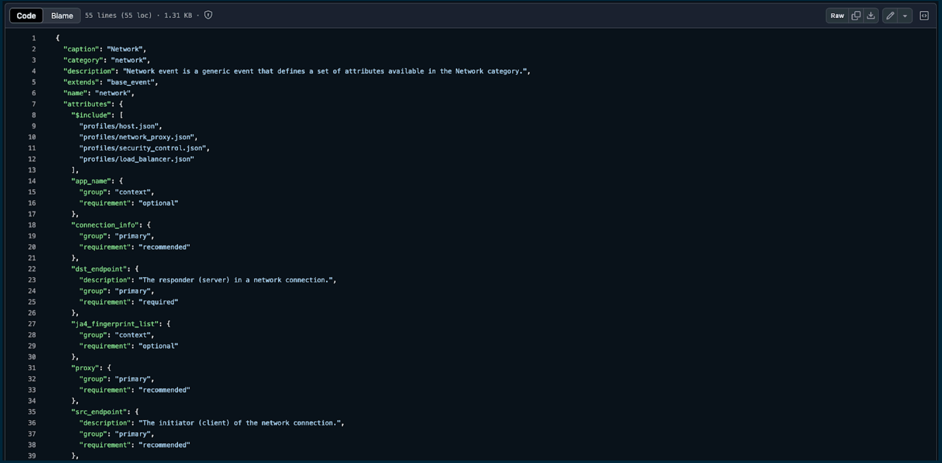
Caption	Name	Group	Requirement	Type	Description
Activity	activity_name	Classification	Optional	String	The event activity name, as defined by the activity_id.
Activity ID	activity_id	Classification	Required	Integer	The normalized identifier of the activity that triggered the event. 0 Unknown The event activity is unknown. 99 Other The event activity is not mapped.
Category	category_name	Classification	Optional	String	The event category name, as defined by category_uid value.
Category ID	category_uid	Classification	Required	Integer	The category unique identifier of the event. 0 Uncategorized
Class	class_name	Classification	Optional	String	The event class name, as defined by class_uid value: `Base Event`.
Class ID	class_uid	Classification	Required	Integer	The unique identifier of a class. A Class describes the attributes available in an event. 0 Base Event The base event is a generic and concrete event. It also defines a set of attributes available in most event classes. As a generic event that does not belong to any event category, it could be used to log events that are not otherwise defined by the schema.

Categories, Event Classes, and Extensions *(cont.)*

The Base Event is extended by the Category Event, which further defines specific enumerations or provides specific Captions and Descriptions atop them.

The Category Events are a "base event" in their own right and group like-attributes and their enumerations for all Events in a Category.

For instance the simple **Network event class** (not viewable in the OCSF Server) ***extends*** the Base Event and populates more objects that all Event Classes within the Category share.

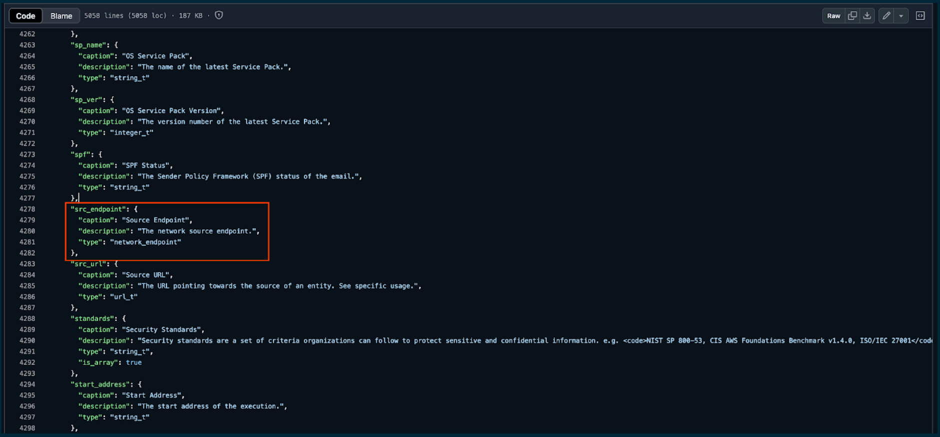


```
{
  "caption": "Network",
  "category": "network",
  "description": "Network event is a generic event that defines a set of attributes available in the Network category.",
  "extends": "base_event",
  "name": "network",
  "attributes": {
    "$include": [
      "profiles/host.json",
      "profiles/network_proxy.json",
      "profiles/security_control.json",
      "profiles/load_balancer.json"
    ],
    "app_name": {
      "group": "context",
      "requirement": "optional"
    },
    "connection_info": {
      "group": "primary",
      "requirement": "recommended"
    },
    "dst_endpoint": {
      "description": "The responder (server) in a network connection.",
      "group": "primary",
      "requirement": "required"
    },
    "ja4_fingerprint_list": {
      "group": "context",
      "requirement": "optional"
    },
    "proxy": {
      "group": "primary",
      "requirement": "recommended"
    },
    "src_endpoint": {
      "description": "The initiator (client) of the network connection.",
      "group": "primary",
      "requirement": "recommended"
    },
```

Finally, the specific Event Class itself can also add additional attributes and objects, as well as further define captions, descriptions, and enumerations atop the Category Event that it is extended from.

The cascading extensions are not noticeable outside of the raw JSON files that define the schema; they are transparent when viewing the OCSF schema documentation (via the OCSF Server).

This is why the OCSF schema documentation is such a great resource for producing mappings into OCSF, as using the raw JSON will often provide an incomplete picture of the full depth of any given part of the Schema.



```
    },
    "sp_name": {
      "caption": "OS Service Pack",
      "description": "The name of the latest Service Pack.",
      "type": "string_t"
    },
    "sp_ver": {
      "caption": "OS Service Pack Version",
      "description": "The version number of the latest Service Pack.",
      "type": "integer_t"
    },
    "spf": {
      "caption": "SPF Status",
      "description": "The Sender Policy Framework (SPF) status of the email.",
      "type": "string_t"
    },
    "src_endpoint": {
      "caption": "Source Endpoint",
      "description": "The network source endpoint.",
      "type": "network_endpoint"
    },
    "src_url": {
      "caption": "Source URL",
      "description": "The URL pointing towards the source of an entity. See specific usage.",
      "type": "url_t"
    },
    "standards": {
      "caption": "Security Standards",
      "description": "Security standards are a set of criteria organizations can follow to protect sensitive and confidential information. e.g. <code>NIST SP 800-53, CIS AWS Foundations Benchmark v1.4.0, ISO/IEC 27001</code
      "type": "string_t",
      "is_array": true
    },
    "start_address": {
      "caption": "Start Address",
      "description": "The start address of the execution.",
      "type": "string_t"
    },
```

Note: For hosting your own local version of the OCSF Server to explore the OCSF schema documentation, read more at the official GitHub **here**.

The extensions are also used in objects, such as the **Network Endpoint** object being extended by both the Destination Endpoint and Source Endpoint objects, or the fact that the Owner object within **Device** is just a **User** object underneath the covers. For further information on how this works, refer to references to the OCSF "dictionary" in the **Understanding OCSF** readme.

OCSF Has It All, but Do You Need All of It?

The most important determination for how you map is the intended downstream usage, not completeness of mapping.

While completeness is commendable, if the data will not be used with analytics, human analysts, visualizations, machine learning, or artificial intelligence (AI) systems, **it does not make sense to fully map at every occasion**.

For instance, the Observables object acts as a lookup table for various important attributes that may appear multiple times within an Event, such as recording different hashes of a malware sample. If you do not intend to query or visualize Observables, or if you do not have more than one attribute that is also an Observable, you do not need to map it.

Within the metaschema there are definitions of required, recommended, and optional fields that can honestly be ignored.

It pains me to admit this as a serial contributor, consumer, and producer of OCSF data, but the truth is unless mapping the data for your processes and playbooks is required, you should consider **"under-mapping"** exactly what you require.

The obvious caveat (again as an OCSF contributor and producer) is if you are working within the context of an OCSF system that will perform full validation: **follow the constraints and requirements!**

However, that doesn't really answer the question of **"what should I map?"** nor **"how do I map something?"**.

In the next sections, you will learn what should be included in any mapping—regardless of what it is.

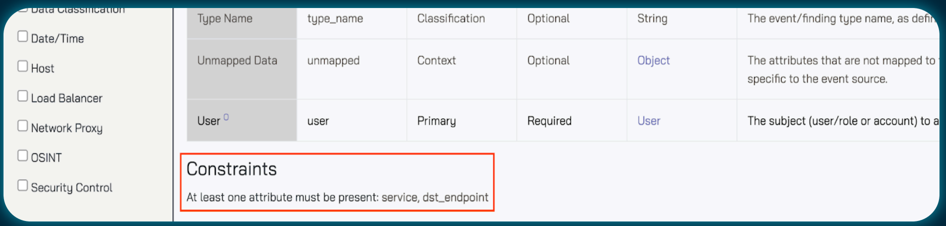
Minimum Necessary Mapping

If you take nothing else away from this section it is that you should have a constant feedback loop with your team. Constantly partner with your threat hunters, detection engineering teams, data engineers and scientists, and anyone else consuming the OCSF data. Find out what important fields or transformations they're expecting to be in the final mapped event. Templatize and continuously test changes and decide as a team.

In the **OCSF metaschema**—the schema rules about the schema framework—every single attribute has as Requirement: either Optional, Recommended, or Required. These Requirement values are set by the contributors of the specific attributes or objects within the wider schema and are built upon by Constraints.

Constraints are defined at the object or Event Class level and mandate that certain attributes (or other nested objects) are mapped. For instance, the **Authentication** event class has Constraints to ensure that either **Service** or **Destination Endpoint** are defined.

Requirements and Constraints are great—contributors typically put these in to serve as helpful hints in the spirit of "minimum necessary". That said, you can safely ignore these constraints and requirements. Unless you are performing incredibly strict mapping validation beyond just ensuring that the schema attributes match, you are safe with skipping attributes that are defined as "Required."



☐ Data Classification
☐ Date/Time
☐ Host
☐ Load Balancer
☐ Network Proxy
☐ OSINT
☐ Security Control

Type Name	type_name	Classification	Optional	String	The event/finding type name, as defi
Unmapped Data	unmapped	Context	Optional	Object	The attributes that are not mapped to specific to the event source.
User [0]	user	Primary	Required	User	The subject (user/role or account) to a

Constraints

At least one attribute must be present: service, dst_endpoint

"Okay, I'll take your word for it, where ***should*** I start?"
- You (probably)

Glad you asked! There is great variability based on the ultimate downstream use cases(s) for the Event, as there is for any given Event, but there are attributes and objects you should always consider mapping.

Key Attributes

Firstly are the main attributes within the "base event", the top-level attributes for an Event Class itself, such as HTTP Activity. The following list is not exhaustive nor should it be taken as mandatory, there are times where you may not even have the data to map directly into these attributes.

Attribute	Description
Timestamp (`time`)	Arguably the most important attribute, this allows you to time order and conduct basic "eyeball analytics" across your Events. There are certain cases where you may need to use other "specialized" timestamps as well if you create this timestamp for Events such as Device Inventory Information or OSINT Inventory Information where an upstream platform doesn't provide a "last seen" date.
Category ID (`category_id`)	This maps the Category and must always be mapped. It's helpful for analytics, searching, and visualizing across the entire Category.
Activity ID (`activity_id`)	The normalized "thing that happened". This is an enumeration with integers corresponding to a defined activity. Often Activity ID enumerations are shared across Events in the same Category: such as Closed Findings across Detections, Compliance, and Vulnerability Findings.
Class ID (`class_id`)	This is the actual ID for the Event Class in question. Obviously you must always map them, especially if you're producing or mapping more than one Event Class
Severity ID (`severity_id`)	The normalized severity is the same across all Events (as of now), this one is a bit controversial at times as for a majority of events you'll likely normalize this to 1 (Informational) or 99 (Other) for more "steady state" types of Events such as Process Activity or File Hosting Activity.
Status ID (`status_id`)	Sometimes this overlaps with the Activity ID, but depending on the Event Class it can provide a sort key at a higher order than Activity ID (e.g., Success vs. Failure) or more specific than Activity ID (e.g., In Progress, Resolved, Suppressed).
Status Detail (`status_detail`)	You can normalize the Status of any given Event with Status ID and you can preserve the raw errors, state changes, overall status, or otherwise in Status Detail.

Key Attributes *(cont.)*

Attribute	Description
Message (`message`)	A human-readable label, description, title, or easy to view piece of information about the Event. This can be a process name, a command line, the unnormalized activity, or an alert or finding title.
Type UID (`type_uid`)	A Class-specific Activity ID, this is calculated by adding the Activity ID to the Class ID and multiplying it by 1000. This is helpful when you only want to get a specific Activity for a specific Class that has overlap in a Category, such as Detection Finding: Create (200401) instead of filtering across two or more attributes.
Metadata UID (`metadata.uid`)	This is the UID attribute in the Metadata object. This is essentially the deduplication key or GUID for any given Event. This can be the trace ID, generated GUID, a UUID5, or proper finding ID.
Metadata Correlation UID (`metadata.correlation_uid`):	Similar to the Metadata UID, this value can be mapped 1:1 directly from downstream sources such as M365 logs, or you can map a common data point from similar normalized elements—such as a User ID, a SID, a hostname, or otherwise.
Product Name (`metadata.product.name`)	The name of the product that produces the downstream raw Event. This can be helpful to query against and quickly see source contributors such as filtering on Crowdstrike Falcon and seeing the various Detection Findings, Vulnerability Findings, Incident Findings, and Device Inventory Information events coming from various Falcon endpoints such as `Detects` or `Hosts`.
Vendor Name (`metadata.product.vendor_name`)	The name of the vendor of the product that produces the downstream raw Event. This is another higher order of sorting and filtering especially if you have a suite of tools such as Microsoft Intune, Microsoft Defender for Endpoint, and Microsoft EntraID to quickly filter on contributing findings.

Mapping With Nuance

Overall, there are not a lot of attributes that should (or must) be normalized in OCSF. Of course, there is some more nuance to go over. Specific Categories and/or specific Event Classes may have additional "base event" attributes—typically integer based enumerations—such as Confidence ID (`confidence_id`) or Risk Level ID (`risk_level_id`) within the Detection Finding event class.

There are other variations, such as Auth Protocol ID (`auth_protocol_id`), within the Authentication event class and many other specific attributes—always check the official schema to check available attributes. Again, work backwards from what data points will be required or otherwise useful to hunting threats or responding to incidents, and then work within the individual datasets for normalization.

Outside of the Base Event level, every Event Class is made up of a collection of one or more objects, which, as you learned earlier, is a collection of attributes that align to a specific "thing".

For instance, the Authentication event class contains several **objects** such as Source and Destination Endpoints (both extend the Network Endpoint object) which contain **attributes** about the source or destination of an authentication, respectively (and obviously). Your logs may not contain this sort of detail.

Again, OCSF is a Framework of many schemas to normalize and standardize data sets into. Just because the object is there—regardless of what the Constraints specify—**it doesn't matter if:**

- It's not of any use to you.
- You don't have data to normalize into the attributes, regardless.

This is a trap I have fallen into many times myself: **going overboard with mapping** into OCSF when I was not making any use of the attributes and thus ended up with multiple duplicate values normalized into similar attributes all across the given Event Class.

This is another reminder to **work backwards from pain and jobs to be done**, and to have a **governance layer and source code control** on your data artifacts so that there is not any ambiguity or confusion.

Even if you **"over-mapped"**, there is always a chance for confusion if your data pipelines and products are not well documented.

All that said, there are absolutely teams out there who want to map as much as possible because they use all of the attributes and mapped data for a variety of reasons. On the opposite end of normalization, there exists a handful of objects that can be used for more or less **"custom" mapping** or for **more flexible searching**.

Mapmaxxing

Enrichment Object

First, the Enrichments (`enrichments`) object is **an array type that contains downstream-specific context data or post-hoc enrichments** such as adding geolocations, reputation assessment, and other additions into the downstream event before normalization into a given Event Class.

The attributes within Enrichments are very generic, allowing for flexibility of enrichment but without much specificity. Of note, the Enrichments object contains the Data (`data`) attribute which is a proper JSON datatype within the schema.

Given that this object is an array type, special care needs to be taken for business intelligence applications and query interfaces such as using an object-oriented programming language, a scripting language, or SQL syntaxes.

You may incur additional performance penalties attempting to loop through and unnest values inside of an array such as using CROSS JOIN and UNNEST together.

Certain SQL dialects may not have access to those keywords or operators, and certain business intelligence tools may not support arrays at all with manual preprocessing outside of the application.

Unmapped Object

Next, the Unmapped (`unmapped`) object **does not contain any attributes at all**. It is meant to be used as a Map data type which is a complex data type that contains key value pairs where the value can be a scalar (string, integer, etc.) or can be an array or another map.

The intended usage is meant to **preserve the raw, downstream key name** as literal as possible. This is not the same as a "custom data" object, it is meant to hold key value pairs that you want to preserve but cannot otherwise be normalized.

Using the Unmapped object requires **well documented data governance**; if you are using a data warehouse or other SQL interface, each dialect has nuances around processing nested key value pairs in data types such as Maps.

Observerables Object

Last, but certainly not least, there are the **Observables** (`observables`). Observables are an array of Observable objects which contain a type (e.g., CVE ID, IP Address, User Agent), the value, and location within the schema it is referenced using dot-notation.

For example, your IP Address Observable that references the Device IP will have a "name" of "device.ip". If there are multiple types of observable attributes such as multiple IP addresses, or if you placed values into `enrichments` or `unmapped`, then Observables become more of a boon than a hassle.

There are drawbacks to Observables. A little over 2 dozen (as of OCSF 1.4.0-dev) scalar values are covered, and some are incredibly ambiguous or broadly applicable such as the Resource UID observable.

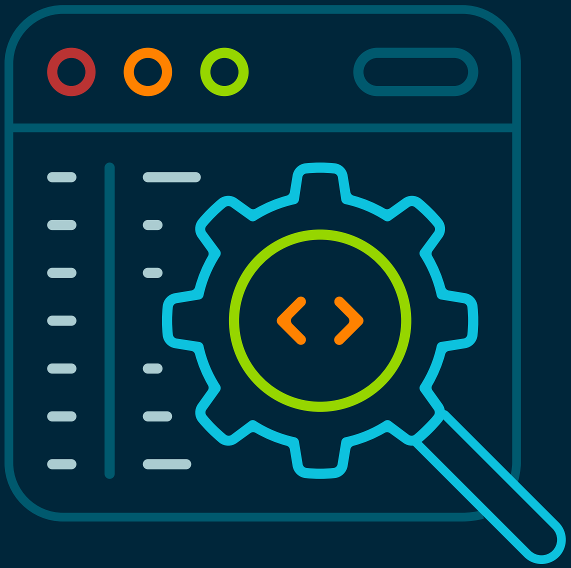
Additionally, the same performance and parsing challenges faced with Enrichments will be faced with Observables as it is also an array-typed object.

If I have not repeated myself enough—whether you are going with a minimum necessary approach, or being very liberal with your mapping, **ensure that a continuous improvement and feedback loop is present.**

Always work with your consumers: be they from IT Observability, SRE, DevOps, Detection Engineering, Security Operations, Threat Hunting teams, or otherwise. Understand what data points are important, which are nice to have, which are unnecessary, and how that fits within the OCSF.

Always attempt to **map the minimum necessary attributes** listed in this section to have a predictable base of data to work from, especially when working with massive amounts of aggregated OCSF data. If you do need to use "looser" objects such as Enrichments, Observables, and/or Unmapped, ensure that the mappings and intended use cases are **well defined and documented**.

Lastly, for **Event Class-specific objects**, ensure you're not overmapping or creating confusing scenarios when many similar objects are mapped out at the same time.



Wrap Up

In this paper on mapping data into OCSF you learned all about the theory of mapping. You learned an overview of OCSF and how the various concepts such as attributes, objects, and Events tie together and how to interpret them.

You also learned about the importance of continuous improvement, data governance, and feedback loops to ensure that your mapping efforts are wholly beneficial to consumers.

Finally, you learned about minimum necessary mappings into OCSF and how to pick and choose specific or broader objects and attributes to utilize.

As mentioned, we've focused on the theory of mapping and a little bit about the "why" we map into OCSF. If you're interested in building upon this knowledge, check out our blog on Mapping Amazon Application Load Balancer Access Logs to the Open Cybersecurity Schema Framework (OCSF).

You may also be interested in the Query Absolute Beginner's Guide to OCSF and Security Data Lakes on Amazon S3, both of which relate to OCSF mapping.

In the event you do not want to worry about mapping, data governance, Extraction, Transformation, and Loading (ETL), or any other data subject ever again: check out Query Federated Search. Our entire data model is based upon OCSF.

We map popular security-relevant data sources into OCSF and also allow you to express your search using OCSF terms. We translate all queries for you into target syntax or dialect, be it SQL, KQL, SPL, or otherwise, and we do not ever store nor replicate your data.

To book a call to explore if we are a fit, go here.

Until next time. Stay Dangerous.

Query: Making Open Federated Search for Security a Reality

Query aims to deliver visibility into all relevant data for security teams. We provide a federated search solution that allows operators to access data at the source and in your data lakes, creating opportunities for more nimble and cost efficient data storage architectures.

Our customers are using Query to expand visibility for security investigations, threat hunting, and incident response. They are drastically reducing the time and complexity of repetitive search tasks and improving outcomes for investigations. Expose your security data with Query.

Ready to expedite your security investigations with open federated search for security?

For more information visit: www.query.ai