Understanding the Open Cybersecurity Schema Framework

Author: Paul Agbabian

Date: August 2023

Status: RFC - Corresponds to schema version 1.0.0

Version: 1.13

Introduction to the Framework and Schema

This document describes the Open Cybersecurity Schema Framework (OCSF) and its taxonomy, including the core cybersecurity event schema built with the framework.[1]

The framework is made up of a set of data types and objects, an attribute dictionary, and the taxonomy. It is not restricted to the cybersecurity domain nor to events, however the initial focus of the framework has been a schema for cybersecurity events. A schema browser for the schema can be found at schema.ocsf.io.

OCSF is agnostic to storage format, data collection and ETL processes. The core schema is intended to be agnostic to implementations. The schema framework definition files and the resulting normative schema are written as JSON.

Personas

There are four personas that are users of the framework and the schema built with the framework.

The *author* persona is who creates or extends the schema. The *producer* persona is who generates events natively into the schema, or via a translation from another schema. The *mapper* persona is who translates or creates events from another source to the schema. The *analyst* persona is the end user who searches the data, writes rules or analytics against the schema, or creates reports from the schema. The analyst may also be considered the *consumer* persona.

For example, a vendor may write a translation from some external source format into the schema but also extend the schema to accommodate source specific attributes or operations. The vendor is operating as both the mapper and author personas. A SOC analyst that collects the data in a SIEM system writes rules against the events and searches events during investigation. The SOC analyst is operating as the analyst persona. Finally, a vendor that emits events natively in OCSF form, even if translated, is a data producer.

Taxonomy Constructs

There are 5 fundamental constructs of the OCSF taxonomy:

1. Data Types, Attributes and Arrays
2. Event Class
3. Category
4. Profile
5. Extension

The scalar data types are defined on top of primitive data types such as strings, integers, floating point numbers and booleans. Examples of scalar data types are Timestamp, IP Address, MAC Address, and User Name.

An *attribute* is a unique identifier name for a specific validatable data type, either scalar or complex.

Complex data types are termed objects. An *object* is a collection of contextually related attributes, usually representing an entity, and may include other objects. Each object is also a data type in OCSF. Examples of object data types are Process, Device, User, Malware and File.

[1] OCSF includes concepts and portions of the ICD Schema, developed by Symantec, a division of Broadcom and has been generalized and made open under Apache 2 license with their permission.

Arrays support any of the data types.

Most scalar data types have constraints on their valid values or ranges, for example Enum integer types are constrained to a specific set of integer values. Enum integer typed attributes are an important part of the framework constructs and used in place of strings where possible to ensure consistency.

Complex data types, or objects, can also be validated based on their particular structure and attribute requirements. Attribute requirements are discussed in a subsequent section.

Appendix A and B describe the OCSF Guidelines and data types respectively.[2]

The *attribute dictionary* of all available attributes, and their types are the building blocks of the framework. Event classes are particular sets of attributes from the dictionary.

Events in OCSF are represented by _event classes _which structure a set of attributes that attempt to describe the semantics of the event in detail. An individual event is an instance of an event class. Event classes have schema-unique IDs. Individual events may have globally unique IDs.

Each event class is grouped by category, and has a unique `category_uid` attribute value which is the category identifier. Categories also have friendly name captions, such as System Activity, Network Activity, Findings, etc. Event classes are grouped into categories for a number of purposes: a container for a particular event domain, documentation convenience and search, reporting, storage partitioning or access control to name a few.

Profiles overlay additional related attributes into event classes and objects** **allowing for cross-category event class augmentation and filtering. Event classes register for profiles that when optionally applied, can be mixed into event classes and objects, by a producer or mapper. For example, System Activity event classes may also include attributes for malware detection or vulnerability information when an endpoint security product is the data source. Network Activity event classes from a host computer may carry the device, process and user associated with the activity. A Security Control profile or Host profile can be applied in these cases, respectively.

Finally, *extensions* allow the schema to be extended** **using the framework without modification of the core schema. New attributes, objects, event classes, categories and profiles are all available to extensions. Existing profiles can be applied to extensions, and new extension profiles can be applied to core event classes and objects as well as to other extensions.

The schema browser visually represents the categories, event classes, dictionary, data types, profiles and extensions in a navigable portal. The schema for an event class, and an event example for a class can be generated via menu options of the browser, which also serves as a validation server via the server APIs, whose documentation is also available from the browser.

Comparison with MITRE ATT&CK[3] Framework The MITRE ATT&CK Framework is widely used in the cybersecurity domain. While the purpose and content type of the two frameworks are different but complementary, there are some similarities with OCSF's taxonomy that may be instructive to those with familiarity with ATT&CK.

Categories are similar to Tactics, which have unique IDs. Event Classes are similar to Techniques, which have unique IDs. Profiles are similar to Matrices[4], which have unique names. Type IDs are similar to Procedures which have unique IDs. Profiles can filter the Event Classes and Categories similar to how Matrices filter Techniques and Tactics.

Differences from MITRE ATT&CK are that in OCSF, Event Classes are in only one Category, while MITRE ATT&CK Techniques can be part of multiple Tactics. Similarly MITRE ATT&CK Procedures can be used

[2]For the most up-to-date guidelines and data types, refer to the schema browser at https://schema.ocsf.io.
[3]MITRE ATT&CK™: https://attack.mitre.org/
[4]MITRE ATT&CK™ Matrix: https://attack.mitre.org/matrices/enterprise/

in multiple Techniques. MITRE ATT&CK™ has Sub-techniques while OCSF does not have Sub-Event Classes.[5]

OCSF is open and extensible by vendors, and end customers while the content within MITRE ATT&CK™ is released by MITRE.

Attributes

Attributes and the dictionary are the building blocks of a schema. This section discusses OCSF attribute conventions, requirements, groupings, constraints, and some of the special attributes used in the core cybersecurity schema.

In general, an attribute from the dictionary has the same meaning everywhere it is used in a schema. Some attributes can have a meaning that is overloaded depending on the event class context where they are used. In these cases the description of the attribute will be generic and include a 'see specific usage' instruction to override its description within the event class context rather than in the dictionary.

Conventions

OCSF adheres to naming conventions in order to more easily identify attributes with similar semantics. These conventions take the form of standard suffixes and prefixes. The standard suffixes are:

`_id, _ids, _uid, _uuid, _ip, _name, _info, _detail, _time, _dt, _process, _ver`

Arrays Attribute names used for arrays end with `s`. For example `category_ids`. A MITRE ATT&CK™ array is named `attacks`.

Unique IDs Attribute names for classification values that are unique within the schema end with `_uid`. Schema classification attributes that have the `_uid` suffix are integers, preset by the schema definition (i.e. they must be populated as defined by the schema).

Certain schema-unique attributes that also have a friendly name or caption have the same prefix but by convention use the `_name` suffix. For example, `class_uid` and `class_name`, or `category_uid` and `category_name`.

Other attributes with the `_uid` suffix convention may be strings or integers, depending on their purpose, although the majority are strings.

A `uid` core attribute is used wherever a producer or mapper populates an identifier for an entity object. Entity objects also have a corresponding `name` attribute by convention. Both are of type string (`string_t`).

Attribute names for values that are globally unique end with _uuid. They do not have friendly names. For example GUIDs.

Enum Attributes Attributes that are of an Enum integer type end with `_id`. Enum constant identifiers are integers from a defined set where each has a friendly name label. Arrays of enum attributes end with `_ids`.

By convention, every Enum type has two common values with integer value 0 for `Unknown` and 99 for `Other`.

If a source event has missing values that are required by the event class for that event, an `Unknown` value should be set for Enum types which is also the default.

If a mapped event attribute does not have a defined enumeration value corresponding to a value of the event, `Other` is used which indicates that a sibling string attribute is populated with the custom attribute value. The sibling string attribute has the same name, minus the suffix. For example, `activity_id` and `activity`, or `severity_id` and `severity`.

[5]The internal source definition of an OCSF schema can be hierarchical but the resulting compiled schema does not expose sub classes.

Sibling string attributes are optional, but if the enum value is `Other` (`99`) then the sibling string **must** be populated with the custom label (i.e. not "Other").

For all defined enumeration integer values, including `Unknown`, the enum label text for the item **may** populate the sibling string attribute. That is, both the integer value and the string attribute are set. If the Enum attribute is required, then both the integer attribute and the sibling string attribute **should** be populated. Attribute requirements are discussed in a subsequent section.

Attribute Requirement Flags

Attributes in the context of an event class have a requirement flag, that depends on the semantics of the event class. Attributes themselves do not have a requirement flag, only within the context of event classes.[6]

The requirement flags are:

- Required
- Recommended
- Optional

Event classes are designed so that the most essential attributes are required, to give enough meaning and context to the information reported by the data source. If an attribute is required, then a consumer of the event can count on the attribute being present, and its value populated. If a required attribute cannot be populated for a particular event class, a default value is defined by the event class, usually `Unknown`.[7]

Recommended attributes should be populated but cannot be in all cases and unlike required attributes are not subject to validation. They do not have default values. Optional attributes may be populated to add context and when data sources emit richer information.

Some event classes may specify constraints on recommended attributes.

Constraints

A *Constraint* is a documented rule subject to validation that requires at least one of the specified recommended attributes of a class to be populated. Constraints are used in classes where there are attributes that cannot be required in all use cases, but in order to have unambiguous meaning, at least one of the attributes in the constraint is required. Attributes in a constraint must be Recommended.

The two constraints are: `at_least_one` and `just_one`. These will be explained further in the section on Event Classes.

Attribute Groups

Attributes are grouped for documentation purposes into *Primary*, *Classification*, *Occurrence*, and *Context* groups. Classification and Occurrence groupings are independent of event class and are defined with the attribute in the dictionary. Primary and Context attributes' groupings are based on their usage within a given event class.

Each event class has primary attributes, the attributes that are indicative of the event semantics in all use cases. Primary attributes are typically Required, or Recommended per event class, based on their use in each class. Primary attributes in the Base Event class apply to all event classes.

Attributes that are important for the taxonomy of the framework are designated as Classification attributes. The classification attributes are marked as Required as part of the Base Event class. Their values are nominally `Unknown`or `Other`and will be overridden within specific event classes.

[6] Event class validation is enforced via the required attributes, in particular the classification attributes, which by necessity need to be kept to a minimum, as well as attribute data type validation and the event class structure

[7] Required attributes that cannot be populated due to information missing from a data source must be carried with the event as *unknown* values - asserting that the information was missing.

Attributes that are related to time and time ranges are designated as Occurrence attributes. The occurrence attributes may be marked with any requirement level, depending on their usage within an event class.

Attributes that are used for variations on typical use cases, to enhance the meaning or enrich the content of an event are designated as Context attributes. The context attributes may be marked with any requirement level, but most often are marked as Optional.

Timestamp and Datetime Attributes

Representing time values is one of the most important aspects of OCSF. For an event schema it is even more important. There are time attributes associated with events that need to be captured in a number of places throughout the schema, for example when a file was opened or when a process started and stopped. There are also times that are directly related to the event stream, for example event creation, collection, processing, and logging. The nominal data type for these attributes is `timestamp_t` based on Unix time or number of milliseconds since the Unix epoch. The `datetime_t` data type represents times in human readable RFC3339 form.

The Date/Time profile when applied adds a sibling attribute of data type `datetime_t`wherever a `timestamp_t`attribute appears in the schema.

The following terms are used below:

Event Producer – the system (application, services, etc.) that generates events. Related to the producer persona.

Event Consumer – the system that receives the events generated by the event producer. Related to the analyst persona.

Event Processor – a system that processes and logs, including an ETL chain, the events received by the event consumer. Related to the mapper and analyst personas.

The core time attributes may be present in all events as they are from the Base Event class. They are:

- `original_time: string`
  The original event time, as created by the event producer as part of the Metadata object of the Base Event class. The time format is not specified by OCSF and as such is a non-validated string. The time could be UTC time in milliseconds (1659378222123), ISO 8601 (2019-09-07T15:50-04:00), or any other value (12/13/2021 10:12:55 PM).
- `time`:`timestamp_t`
  The normalized event occurrence time. Normalized time means the original event time `original_time` is corrected for the clock skew of the source if any, and batch submission delay and after it was converted to the OCSF `timestamp_t`.
- `processed_time: timestamp_t`
  The time when the event (or batch of events) was sent by the event processor to the event consumer. The processed time can be used to determine the clock skew at the earliest known event source. Clock skew occurs when the UTC clock time on one computer differs from the UTC clock time on another computer. It is assumed that the transport latency is very small compared to the clock skew, therefore if the `processed_time` is very close to the `logged_time`, no correction should be made, notwithstanding any known hops.
- `logged_time: timestamp_t`
  The time when the event consumer logged the event. It must be equal or greater than the normalized event occurrence time.
- `modified_time: timestamp_t`
  The time when the event was last updated or enriched. It must be equal or greater than the normalized event occurrence time. It could be less-than, equal, or greater-than the `logged_time` .
- `start_time/end_time: timestamp_t`
  The start and end event times of the Base Event class are used when the event represents some activity that happened over a time range, for example a vulnerability or virus scan, or a discovery run. The

other use-case is event aggregation. Aggregation is a mechanism that allows for a number of events of the same event type to be summarized into one for more efficient processing. For example netflow events. In this use case, the `count` integer attribute is also populated.

Time Zone The time zone where the event occurred is represented by the `timezone_offset` attribute of data type Integer. Although time attributes are otherwise UTC except for the pass through attribute original_time, most security use cases benefit from knowing what time of day the event occurred at the event source.

`timezone_offset` is the number of minutes that the reported event time is ahead or behind UTC, in the range -1,080 to +1,080. It is a recommended attribute of the Base Event class.

Metadata

Metadata is an object referenced by the required Base Event attribute `metadata`. As its name implies, the attribute is populated with data outside of the source event. Some of the attributes of the object are optional, such as `logged_time` and `uid`, while the `version` attribute is required - the schema version for the event. It is expected that a logging system *may* assign the `logged_time` and `uid` at storage time.

Metadata attributes such as `modified_time` and `processed_time` are optional. `modified_time`is populated when an event has been enriched or mutated in some way before analysis or storage. `processed_time`is populated typically when an event is collected and submitted to a logging system.[8]

Version. OCSF core schema version uses Semantic Versioning Specification (SemVer), e.g. `0.99.0,` which indicates to consumers of the event which attributes may be found in the event, and what the class and category structure are. The convention is that the major version, after `1.0.0`, or first part, remains the same while versions of the schema remain backwards compatible with previous versions of the schema and framework. As new classes, attributes, objects and profiles are added to the schema, the minor version, or second part of the version increases. The third part is reserved for corrections that don't break the schema, for example documentation or caption changes.

Extensions, discussed later, have their own versions and can change at their own pace but must remain compatible and consistent with the major version of the core schema that they extend. The optional `extension` attribute of type Schema Extension carries the version of an extension.

Observables

Observable is an object referenced by the primary Base Event class array attribute `observables`. It is populated from other attributes produced or mapped from the source event. An Observable object surfaces attribute information in one place irrespective of event class, while the security relevant indicators that populate the observable may occur in many places across event classes. In effect it is an array of summaries of those attributes regardless of where they stem from in the event based on their data type or object type (e.g. `ip_address`, `process`, `file` etc).

For example, an IP address may populate multiple attributes: `public_ip, intermediate_ips, ip`(as part of objects Endpoint, Device, Network Proxy, etc.). An analyst may be interested to know if a particular IP address is present anywhere in any event. Searching for the IP address value from the Base Event `observables` attribute surfaces any of these events more easily than remembering all of the attributes across all event classes that may have an IP address.

There are three important attributes in the Observable object: `name`, `value`, and `type_id`. For scalar attributes within an event, all three observable attributes are populated, where the `type_id` declares what the type of attribute is, the `name` is the fully qualified attribute name within the event, and `value` is the value of that attribute.

[8]Note that a non-trivial difference between the processed_time and the logged_time in UTC may indicate a clock synchronization problem with the source of the event (but not necessarily the actual source of the event if there is an intermediate collection system or forwarder).

For complex (object type) attributes, Observable.`name` is the pointer or reference to the attribute, but as an object has more than one value, Observable.`value` is not populated.

```
"observables": [
{ "name": "actor.process.name",
"type": "Process Name",
"type_id": 9,
"value": "Notepad.exe"
},
{ "Name": "tls.ja3_hash",
"Type": "Fingerprint",
"Type_id": "30"
},
{ "name": "file.name",
"type": "File Name",
"type_id": 7,
"value": "Notepad.exe"
}]
```

Enrichments

Enrichment is an object referenced by the Base Event array attribute `enrichments`. An Enrichment object describes additional information added to the event during collection or event processing but before an immutable operation such as storage of the event. An example would be looking up location data on an IP address, or IOCs against a domain name or file hash.

Because enriching data can be extremely open-ended, the object uses generic string attributes along with a JSON `data` attribute that holds an arbitrary enrichment in a form known to the processing system. Similar to the Observable object, `name` and `value` attributes are required to point to the event class attribute that is being enriched. Unlike Observable, there is no predefined set of attributes that are tagged for enrichment, therefore only a recommended `type` attribute is specified (i.e. there is no `type_id` Enum).

Also unlike Observable, which is synchronized with the time of the event, it is assumed that there is some latency between the event time and the time the event is enriched, hence the Base Event class `metadata.modified_time` should be populated at the time of enrichment.

For example

```
"metadata": {
    "logged_time": 1659056959885810,
    "modified_time": 1659056959885807,
    "processed_time": 1659056959885796,
    "sequence": 69,
    "uid": "1310fc5c-0edb-11ed-88fc-0242ac110002",
    "version": "1.0.0-RC3"
},
"enrichments": [
{
"data": {"hash": 0c5ad1e8fe43583e279201cdb1046aea742bae59685e6da24e963a41df987494},
"name": "ip",
"provider": "media.defense.gov",
"type": "IP Address",
"value": "103.216.221.19"
},
{
"data": {"yara_rule": wellmail_unique_strings \{ meta: description =
"Rule for detection of WellMail based on unique strings contained in the binary"
```

```
author = "NCSC" hash = "0c5ad1e8fe43583e279201cdb1046aea742bae59685e6da24e963a41df987494"
strings: $a = "C:\\Server\\Mail\\App_Data\\Temp\\agent.sh\\src"
$b = "C:/Server/Mail/App_Data/Temp/agent.sh/src/main.go" $c = "HgQdbx4qRNv"
$d = "042a51567eea19d5aca71050b4535d33d2ed43ba" $e = "main.zipit"
$f = "@[^\\s]+?\\s(?P.*?)\\s" condition: uint32(0) == 0x464C457F and 3 of them \}"},
"name": "ip",
"provider": "media.defense.gov",
"type": "IP Address",
"value": "103.216.221.19"
}]
```


Event Classes

Events are represented by instances of Event Classes, which are particular sets of attributes and objects representing a log line or telemetry submission at a point in time. Event classes have semantics that describe what happened: either a particular activity, disposition or both.

It is the intent of the schema to allow for the mapping of any raw event to a single event class. This is achieved by careful design using composition rather than a multiple inheritance approach. In order to completely capture the information in a rich data source, many attributes may be required.

Unfortunately, not every data source emits the same information for the same observed behavior. In the interest of consistency, accuracy and precision, the schema event classes specify which dictionary attributes are essential, (recommended or required), while others are optional as not all are needed across different data sources. Attribute requirements, aside from Classification attributes from the Base Event class, are always within the scope of the event class definition and not tied to the attributes themselves.

By convention, all event classes extend the Base Event event class. Attributes of the Base Event class can be present in any event class and are termed Base Attributes.

Base Event Class Attributes

The Base Event class has required, recommended, and optional attributes that apply to all core schema classes. The required attributes must be populated for every core schema event. Optional Base Event class attributes may be included in any event class, along with event class-specific optional attributes. Individual event classes will include their own required and recommended attributes.

Examples of required base attributes are `class_uid`, `category_uid`, `activity_id`, `severity_id`.

Examples of recommended base attributes are `timezone_offset, status_id, message.`

Examples of optional base attributes are `activity_name`, `start_time`, `end_time`, `count`, `duration`, `unmapped`.

**Each event class has a unique class_uid attribute value which is the event class identifier. It is a required attribute whose value overrides the nominal Base Event class value of 99. Event class friendly names are defined by the schema, optionally populate the class_name attribute and are descriptive of the specific class, such as File System Activity or Process Activity.

**Every event class has a category_uid attribute value which indicates which OCSF Category the class belongs to. An event class may be of only one category. Category friendly names are defined by the schema, optionally populate the category_name attribute and are descriptive of the specific category the class belongs to, such as System Activity or Network Activity.

**Every event class has an activity_id Enum attribute, constrained to the values appropriate for each event class. The semantics of the class are further defined by the activity_id attribute, such as Open for File System Activity or Launch for Process Activity. By convention, activity_id Enum labels are present tense imperatives. The Enum label optionally may populate the activity_name attribute, which is a sibling to the activity_id Enum attribute but as a Classification group attribute, follows the _name suffix convention.

Special Base Attributes

There are a few base attributes that are worth calling out specifically. These are the `unmapped` attribute, the `raw_data` attribute and the `type_uid`attribute.

While most if not all fields from a raw event can be parsed and tokenized, not all are mapped to the schema. The fields that are not mapped may be included with the event in the optional `unmapped` attribute.

The `raw_data` optional attribute holds the event data as received from the source. It is unparsed and represented as a String type.

The `type_uid` required attribute is constructed by the combination of the event class of the event (`class_uid`)and its activity (`activity_id`). It is unique across the schema hence it has a `_uid` suffix. The `type_uid` friendly name, `type_name`, is a way of identifying the event in a more readable and complete way. It too is a combination of the names of the two component parts.

The value is calculated as: `class_uid * 100 + activity_id`. For example:

`type_uid` = `3001 * 100 + 1 = 300101`

`type_name` = "Authentication: Logon"

A snippet of a File Activity event example with random values is shown below[9]:


```
{
"activity_id": 11,
"activity_name": "Decrypt",
"actor": {},
"category_name": "System Activity",
"category_uid": 1,
"class_name": "File System Activity",
"class_uid": 1001,
"device": {},
"end_time": 1685403212867803,
"file": {},
"message": "entry queue amateur",
"metadata": {},
"observables": [],
"severity": "Low",
"severity_id": 2,
"start_time": 1685403212792508,
"status": "img logs grove",
"status_detail": "barrier filled clothes",
"time": 1685403212834047,
"type_name": "File System Activity: Decrypt",
"type_uid": 100111
}
```


Constraints

As discussed in a previous section, an event class can have constraints that are more versatile than simple Required attribute requirements. When at least one of a set of recommended attributes must be present, the class can assert the `at_least_one` constraint:


```
 "constraints": {
   "at_least_one": [
     "ip",
```


[9]Objects have been collapsed to save space. You can generate full examples with dummy data at https://schema.ocsf.io/doc/swagger.json or from within the browser.

```
      "mac",
      "name",
      "hostname"
    ]
  }
```

Or the `just_one` constraint:

```
"constraints": {
    "just_one": [
      "privileges",
      "group"
    ]
  }
}
```

Associations

Attributes within an event class are sometimes associated with each other and in some cases only one of them is present in the event while another may be looked up at processing or storage time. OCSF denotes this within a class definition via the association construct:

```
"associations": {
    "actor.user": [
      "src_endpoint"
    ],
    "dst_endpoint": [
      "user"
    ],
    "src_endpoint": [
      "actor.user"
    ],
    "user": [
      "dst_endpoint"
    ]
  }
```

In this example from the Authentication class, the `user` as actor associates with its endpoint, the `src_endpoint` attribute of the class, while the target `user` associates with its endpoint, the `dst_endpoint` of the class. Note that the associations in this class are bi-directional, which is common, although uni-directional associations are also possible in other situations.

The construct may be useful for automated processing systems where a lookup service is available for an attribute that isn't or can't be populated via the source event producer. In these cases the `processor_time` should be populated at the time of the association, as with other types of enrichments.

Categories

A Category organizes event classes that represent a particular domain. For example, a category can include event classes for different kinds of events that may be found in an access log, or audit log, or network and system events. Each category has a unique `category_uid`attribute value which is the category identifier. Category IDs also have `category_name`friendly name attributes, such as System Activity, Network Activity, Audit, etc.

An example of categories with some of their event classes is shown in the below table.

System Activity	Network Activity	Identity & Access Management	Findings	Discovery	Application Activity
File System Activity	Network Activity	Account Change	Security Finding	Device Inventory Info	Web Resources Activity
Kernel Extension Activity	HTTP Activity	Authentication		Device Config State	Application Lifecycle
Kernel Activity	DNS Activity	Authorize Session			API Activity
Memory Activity	DHCP Activity	Entity Management			Web Resrouce Access Activity
Module Activity	RDP Activity	User Access Management			
Scheduled Job Activity	SMB Activity	Group Management			
Process Activity	SSH Activity				
	FTP Activity				
	Email Activity				
	Network File Activity				
	Email File Activity				
	Email URL Activity				

Finding the right granularity of categories is an important modeling topic. Categorization is weakly structural while event classification is strongly structural (i.e. it defines the particular attributes, their requirements, and specific Enum values for the event class).

Many events produced in a cloud platform can be classified as network activity. Similarly, many host system events include network activity. The key question to ask is, do the logs from these services and hosts provide the same context or information? Would there be a family of event classes that make sense in a single category? For example, does the NLB Access log provide context/info similar to a Flow log? Does network traffic from a host provide similar information to a firewall or router? Are they structured in the same fashion? Do they share attributes? Would we obscure the meaning of these logs if we normalize them under the same category? Would the resultant category make sense on its own or will it lose its contextual meaning all together?

Using profiles, some of these overlapping categorical scenarios can be handled without new partially redundant event classes.

Profiles

Profiles are overlays on event classes and objects, effectively a dynamic mix-in class of attributes with their requirements and constraints. While event classes specialize their category domain, a profile can augment existing event classes with a set of attributes independent of category. Attributes that must or may occur in any event class are members of the Base Event class. Attributes that are specialized for selected classes are members of a profile.

Multiple profiles can be added to an event class via an array of profile values in the optional `profiles` attribute of the Base Event class. This mix-in approach allows for reuse of event classes vs. creating new classes one by one that include the same attributes. Event classes and instances of events that support the

profile can be filtered via the `profiles` attribute across all categories and event classes, forming another dimension of classification.

For example, a `Security Controls` profile that adds MITRE ATT&CK™ Attack and Malware objects to System Activity classes avoids having to recreate a new event class, or many classes, with all of the same attributes as the System Activity classes. A query for events of the class will return all the events, with or without the security information, while a query for just the profile will return events across all event classes that support the `Malware`profile. A `Host` profile can add `Device`, and `Actor` objects to Network Activity event classes when the network activity log source is a user's computer. Note that the `Actor`object includes `Process`and `User`objects, so a Host profile can include all of these when applied. A Cloud profile could mix-in cloud platform specific information onto Network Activity events.

The `profiles` attribute is an optional array attribute of the Base Event class. The absence of the `profiles` attribute means no profile attributes are added as would be expected. Attributes defined with a profile have requirements that cannot be overridden, since profiles are themselves optional; it is assumed that the application of a profile is because those attributes are desired and can be populated.

However some classes, such as System Activity classes, build-in the attributes of a profile, for example the `Host` profile attributes `device` and `actor` are defined in the class. When a class definition includes the profile attributes, it still registers for that profile in the class definition so as to match any searches across events for that profile. In this case the class defined attribute requirement definitions take precedence.

Core schema profiles for `Security Control`, `Host`, `Cloud`, `Container` and `Linux` (for the Linux extension described later) are shown in the below table with their attributes.

Security Control	**Host**	**Cloud**	**Container**	**Linux**
attacks	actor	api	container	group
disposition_id / disposition	device	cloud	namespace_pid	euid
malware				egid
				auid

A special `Date/Time` profile adds `Datetime` typed time attributes in every class where there is a `Timestamp` time attribute. This allows for human readable RFC-3339 strings paired with epoch UTC integer values.

Other profiles could be product oriented, such as Firewall, IDS, VA, DLP etc. if they need to add attributes to existing classes. They can also be more general, platform oriented, such as for Mac, Linux or Windows environments.

The core schema comes with a Linux profile via the Linux platform extension.

Vendors can add profiles via extensions. For example, Splunk Technical Add-ons might define a profile that could be added to all events with Splunk's standard `source`, `sourcetype`, `host` attributes.

Disposition

The `disposition_id` attribute of the Security Control profile indicates the outcome or state of the event class' activity at the time of event capture and is an Enum with a standard set of values, such as Blocked, Quarantined, Deleted, Delayed.

Only event classes that register for the profile may have a `disposition_id`but all have an `activity_id.` A typical use of `disposition_id` is when a security protection product detects a threat and blocks it. The activity might have been a file open, but if the file was infected, the disposition would be that the file open was blocked. As of this writing, `disposition_id`is added to core schema classes only via the Security Controls profile.

Profile Application Examples

Using example categories and event classes from a preceding section, examples of how profiles might be applied to event classes are shown below.

System Activity The event classes **would** all include the Host profile and **may** include the Security Controls or Cloud profile.

Network Activity The event classes **may** include the Host profile and **may** include the Security Controls or Cloud profile.

Identity & Access Management The event classes **would** include the Host profile, (due to actor.user), **may** include the Cloud profile, and **would not** include the Security Control profile.

Personas and Profiles

The personas called out in an earlier section, producer, author, mapper, analyst, all can consider the profile from a different perspective.

Producers, who can also be authors, can add profiles to their events when the events will include the additional information the profile adds. For example a vendor may have certain system attributes that are added via an extension profile. A network vendor that can detect malware would apply the Security Controls profile to their events. An endpoint security vendor can apply the Host, User and Security Controls profile to network events.

Authors define profiles, and the profiles are applicable to specific classes, objects or categories.

Mappers can add the profile ID and associated attributes to specific events mapped to logs in much the same way producers would apply profiles.

Analysts, e.g. end users, can use the browser to select applicable profiles at the class level. They can use the profile identifier in queries for hunting, and can use the profile identifiers for analytics and reporting. For example, show all malware alerts across any category and class.

Extensions

OCSF schemas can be extended by adding new attributes, objects, categories, profiles and event classes. A schema is the aggregation of core schema entities and extensions.

Extensions allow a particular vendor or customer to create a new schema or augment an existing schema.[10] Extensions can also be used to factor out non-essential schema domains keeping a schema small. Extensions to the core schema use the framework in the same way as a new schema, optionally creating categories, profiles or event classes from the dictionary. Extensions can add new attributes to the dictionary, including new objects. Extended attribute names can be the same as core schema names but this is not a good practice for a number of reasons. As with categories, event classes and profiles, extensions have unique IDs within the framework as well as versioning.[11]

As of this writing, two platform extensions augment the core schema: Linux and Windows. The Linux extension adds a profile, while the Windows extension adds three classes to the System Activity category.

Another use of extensions to the core schema is the development of new schema artifacts, which later may be promoted into the core schema or to a platform extension. Another use of extensions is to add vendor specific extensions in addition to the core schema. In this case, a best practice is to prefix the schema artifacts with a

[10] An extension does not need to extend the core schema base class if it is a new schema.

[11] Reserved identifier ranges are registered within a file in the project GitHub repository. Extended events should populate the `metadata.version` attribute with the extended schema version.

short identifier associated with the extension range registered.[12] Lastly, as mentioned above, entirely new schemas can be constructed as extensions.

Examples of new experimental categories, new event classes that contain some new attributes and objects are shown in the table below with a `Dev` extension superscript convention. In the example, extension classes were added to the core Findings category, and three extension categories were added, Policy, Remediation and Diagnostic, with extension classes.

Findings	**Policy**$^{\text{Dev}}$	**Remediation**$^{\text{Dev}}$	**Diagnostic**$^{\text{Dev}}$
Incident Creation$^{\text{Dev}}$	Clipbaord Content Protection$^{\text{Dev}}$	File Remediation$^{\text{Dev}}$	CPU Usage$^{\text{Dev}}$
Incident Associate$^{\text{Dev}}$	Compliance$^{\text{Dev}}$	Folder Remediation$^{\text{Dev}}$	Memory Usage$^{\text{Dev}}$
Incident Closure$^{\text{Dev}}$	Compliance Scan$^{\text{Dev}}$	Startup Application Remediation$^{\text{Dev}}$	Throughput$^{\text{Dev}}$
Incident Update$^{\text{Dev}}$	Content Protection$^{\text{Dev}}$	User Session Remediation$^{\text{Dev}}$	
Email Delivery	Information Protection$^{\text{Dev}}$		
Finding$^{\text{Dev}}$			

A brief discussion of how to extend the schema is found in Appendix C.

Appendix A - Guidelines and Conventions

Guidelines for attribute names

- Attribute names must be a valid UTF-8 sequence.
- Attribute names must be all lower case.
- Combine words using underscore.
- No special characters except underscore.
- Reserved attributes are prefixed with an underscore.
- Use present tense unless the attribute describes historical information.
- `activity_id`enum labels should be present tense. For example, `Delete`. `disposition_id`enum labels should be past tense. For example, `Blocked`.
- Use singular and plural names properly to reflect the attribute content.
  For example, use `events_per_sec` rather than `event_per_sec`.
- When an attribute represents multiple entities, the attribute name should be pluralized and the value type should be an array.
  Example: `process.loaded_modules` includes multiple values – a loaded module names list.
- Avoid repetition of words where possible.
  Example: `device.device_ip` should be `device.ip`.
- Avoid abbreviations when possible.
  Some exceptions can be made for well-accepted abbreviations. Example: `ip`, or `os`.
- For vendor extensions to the dictionary, prefix attribute names with a 3-letter moniker in order to avoid name collisions. Example: `aws_finding, spk_context_ids`.

Appendix B - Data Types

Refer to https://schema.ocsf.io/data_types for the OCSF data types and their validation constraints.

Appendix C - Schema Construction and Extension

The OCSF schema repository can be found at https://github.com/ocsf/ocsf-schema.

[12]The Schema Browser will label extensions with a superscript.

The repository is structured as follows:

File or Folder	Purpose
categories.json	the schema categories are defined and must be present for classes to be in a category
dictionary.json	the schema dictionary is where all attributes must be defined
version.json	the schema semver version, every change to the schema requires this file be updated
enums/	the schema enum definitions, optional if enums are shared
events/	the schema event classes
extensions/	the schema extensions, a similar structure is set per extension
includes/	the schema shared files
objects/	the schema object definitions
profiles/	the schema profiles

For information and examples about how to add to the schema, see CONTRIBUTING.md in the OCSF GitHub.

Extending the Schema

To extend the schema create a new directory using a unique extension name (e.g. dev) in the extensions directory. The directory structure is the same as the top level repository structure above, and it may contain the following files and subdirectories, depending on what type of extension is desired:

File or Folder	Purpose
categories.json	Create to define a new event category to reserve a range of class IDs
dictionary.json	Create to define new attributes
events/	Create to define new event classes
objects/	Create to define new objects
profiles/	Create to define new profiles

In order to reserve an ID space, and make your extension public, add a UID to your extension name in the OCSF Extensions Registry here to avoid collisions with core or other extension schemas. For example, the dev extension would have a row in the table as follows:

Extension Name	Type	UID	Notes
Development	dev	999	The development schema extensions

New categories and event classes will have their unique IDs offset by the UID.

More information about extending existing schema artifacts can be found at extending-existing-class.md.

Notes