Understanding the Open Cybersecurity Schema Framework

Author: Paul Agbabian
Date: October 2022
Status: RFC
Version: 1.9

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, 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. 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.

For example, a vendor may write a translation from a native source format into the schema but also extend the schema to accommodate vendor 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 is a data producer.

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

Taxonomy Constructs

There are 6 fundamental constructs of the OCSF taxonomy:

1. Data Types, Attributes and Arrays
2. Attribute Dictionary
3. Event Class
4. Category
5. Profile
6. 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, Pathname, and User Name.

An *attribute* is a unique identifier name for a specific field and a corresponding 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 possibly includes other objects. Each object is also a data type in OCSF. Examples of object data types are Process, Device, User, Malware and File.

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.

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, Security 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 which can be optionally applied, or 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 Malware profile or Host and User profiles can be applied in these cases.

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 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.

Comparison with MITRE ATT&CK[2] 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[3], 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 in multiple Techniques. MITRE ATT&CK has Sub-techniques while OCSF does not have Sub-Event Classes.[4]

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

---

[2] MITRE ATT&CK: https://attack.mitre.org/

[3] MITRE ATT&CK Matrix: https://attack.mitre.org/matrices/enterprise/

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

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, _uid, _uuid, _ip, _name, _info, _time, _process, _ver`

Attribute names for values that are unique within the schema end with `_uid`. 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`.

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

Attributes that refer to a source event literal value are prefixed with `ref_`. For example, `ref_event_code`, `ref_time`, `ref_event_name`.

Enum Attribute Conventions

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.

By convention, every Enum type has two common values with integer value 0 for `Unknown` and -1 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 desired enumeration value corresponding to a value of the raw event, `Other` is used which indicates that a companion string attribute is populated

with the custom attribute value. The string attribute has the same name, minus the suffix. For example, `activity_id` and `activity`, or `severity_id` and `severity`.

For all defined enumeration integer values, the label for the item also populates the companion string attribute. That is, both the integer value and the string attribute are always set. If the Enum attribute is required, then both the integer attribute and the string attribute are required. Attribute requirements are discussed in a subsequent section.

Reserved Attribute Conventions

Reserved attributes are populated by a collection, processing or storage system and when defined within an event class are not populated by the producer or mapper personas. Their names are prefixed with an underscore by convention.

The core schema does not have any reserved attributes.

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.[5]

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`.[6]

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.

---

[5] 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

[6] 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. Required attributes that are mapped from a source event (rather than produced natively) may also be populated by a collection or processing system, most notably the schema `version` attribute of the `metadata` attribute's object.

Some event classes may specify constraints on recommended attributes. Constraints will be discussed in the Event Class section.

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.

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 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.[7] The `datetime_t` data type represents times in human readable RFC3339 form.

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.

[7] Timestamp_ex profile adds sibling attributes to timestamp_t attributes based on RFC3339 text format.

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.

- `ref_time: string`
  The original event time, as created by the event producer. The time format is not specified by OCSF. 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 `ref_time` was corrected for the clock skew 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. Clock skew occurs when the clock time on one computer differs from the clock time on another computer. It is assumed that the transport latency is very small compared to the clock skew, otherwise no correction should be made.

- `logged_time: timestamp_t`
  The time when the event consumer logged the event. It must be equal or greater than the event time.

- `modified_time: timestamp_t`
  The time when the event was last updated or enriched. It must be equal or greater than the event 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 are used when the event represents some activity that happened over time, for example a vulnerability or virus scan. The other use-case is event aggregation. Aggregation is a mechanism that allows for a high number of events of the same event type that are raised 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 ref_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, discussed next.

Metadata

Metadata is an object referenced by the primary 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 reserved, 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. Note that a reserved attribute may have any of the three requirement flags.

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 or submitted to a logging system.[8]

Version. OCSF core schema version uses Semantic Versioning Specification (SemVer), e.g. `0.11.0`, 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.

Observables

Observable is an object referenced by the primary base event array attribute `observables`. It is populated from other attributes produced or mapped from the source event. An Observable object (`observable`) surfaces in one place across any event while the security indicators that populate it 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).

[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 (and not necessarily the event source in the event there is an intermediate collection system or forwarder).

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.

Further, there are other attributes that may also need to be surfaced from the same event, which is why `observables` is an array attribute of the base event class. The interesting attributes of scalar or object data types are represented as strings, with an attribute type discriminator to indicate the original type:

```
"observables": [
{
"name": "actor_process",
"type": "Process",
"type_id": 25,
"value": "Notepad.exe"
},
{
"name": "file.name",
"type": "File Name",
"type_id": 7,
"value": "Notepad.exe"
}]
```

Enrichments

Enrichment is an object referenced by the primary base event array attribute `enrichments`. An Enrichment object (`enrichment`) 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": "0.11.0"
},
"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 Class

Events are represented by instances of Event Classes, which are a particular set 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.

Each event class has a unique `class_uid` attribute value which is the event class identifier. Event class friendly names populate the `class_name` attribute and are descriptive of some type of activity, such as File Access Activity or Process Activity.

The semantics of the class are defined by the specific activity, via the `activity_id` attribute, such as File Opened or Process Started. Other attributes of the class indicate the details such as the file name, or the process name.

Every event class has an activity, disposition or outcome, via the `activity_id` and `disposition_id` Enum attributes, constrained to the values appropriate for each event class. The `activity_id` indicates what specific activity the event is reporting. The `disposition_id` indicates what the outcome or state of the activity was at the time of event capture.

Not all event classes 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.

The unique combination of a `class_uid` and `activity_id` or `disposition_id` is represented by the `type_uid` derived attribute. When `disposition_id` is populated as part of an event class it is used rather than `activity_id` as it is more representative of what happened during the activity.

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, aside from inconsistent naming and typing of extracted fields, driving the need for normalization, 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 are always within the scope of the event class definition and not tied to the attributes themselves.

Base Event Class Attributes

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.

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

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

Examples of recommended base attributes are `timezone_offset, status_id / status, product.`

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

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

The `type_uid` attribute is constructed by the combination of the event class of the event (`class_uid`) and its activity (`activity_id`) or disposition (`disposition_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 Audit: Logon"

A snippet of a File Activity event example is shown below.

```
{
   "category_uid": 1,
   "class_uid": 1004,
   "activity_id": 2,
   "event_name": "Endpoint File Activity:Read",
   "event_time": "1970-01-20T02:09:34.676997Z",
   "event_uid": 100400,
   "message": "File foobar.json opened",
   "severity_id": 1,
   "time": 1649374676992
}
```

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.

Category

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. Note, these are not final.

System Activity	Network Activity	Audit Activity	Findings	Cloud Activity
File Activity	Network Activity	Account Change	Security Finding	Cloud API
Folder Activity	HTTP Activity	Authentication		Cloud Storage Activity
Kernel Activity	DNS Activity	Authorization		Cloud Virtual Machine
Memory Activity	DHCP Activity	Entity Change		
Module Activity	SSH Activity			
Peripheral Activity	RDP Activity			
Process Activity				
Scheduled Job Activity				
Registry Key Activity				
Registry Value Activity				
Resource 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.

Profile

Profiles are overlays on event classes, effectively a dynamic mix-in class of attributes and objects with their requirements and constraints.[9] 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 `profiles` attribute. 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.

For example, a `Malware` profile that adds MITRE ATT&CK 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 security profile. A `Host` profile and a `User` profile can add `Device`, `Process` and `User` objects to network activity event classes when the network activity log source is a user's computer. A cloud provider profile could mix-in cloud platform specific information onto network activity events.

Proposals for three built-in profiles for `Malware`, `Host` and `User` are shown in the below table with their attributes.

Malware Profile	**Host Profile**	**User Profile**
`disposition_id / disposition`	`device`	`user`
`attacks`	`actor_process`	`is_user_present`
`cvssv2`		`user_entities`
`malware / related_malware`		`accounts`
`quarantine_uid`		`user_result`

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 cloud or Windows environments.

[9] Refer to Proposal 3: Profiles in the document OCSF Schema Collaboration: Initial Decisions

For example, AWS services log events with an ARN (AWS Resource Name) and an AWS IAM Account. An AWS specific profile can be added to any event class or category of classes that includes arn and IAM account attributes. Splunk Technical Add-ons would define a profile that would be added to all events with Splunk's standard source, sourcetype, host, attributes.

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 following **would** all include the Host profile and **may** include the Malware profile:
File Activity
Folder Activity
Kernel Activity
Memory Activity
Module Activity
Peripheral Activity
Process Activity
Resource Activity
Scheduled Job Activity

Windows Activity

The following **would** include the Host profile and **may** include the Malware profile:
Registry Key Activity
Registry Value Activity

Network Activity

The following **may** include the Host profile and **may** include the Malware profile:
DNS Activity
HTTP Activity
Network Activity

Audit Activity

The following **would** include the User profile, **may** include the Host profile and **would not** include the Malware profile:
Account Change
Authentication

Authorization
Entity Activity

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 Malware profile to their events. An endpoint security vendor can apply the Host, User and Malware 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]

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

[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.

a 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**[Dev]	**Remediation**[Dev]	**Diagnostic**[Dev]
Incident Creation[Dev]	Clipboard Content Protection[Dev]	File Remediation[Dev]	CPU Usage[Dev]
Incident Associate[Dev]	Compliance[Dev]	Folder Remediation[Dev]	Memory Usage[Dev]
Incident Closure[Dev]	Compliance Scan[Dev]	Unsuccessful Remediation[Dev]	Status[Dev]
Incident Update[Dev]	Content Protection[Dev]	Startup Application Remediation[Dev]	Throughput[Dev]
Email Delivery Finding[Dev]	Information Protection[Dev]	User Session Remediation[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.

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

- Use present tense unless the attribute describes historical information.

- 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.
  Example: `host.host_ip` should be `host.ip`.

- Avoid abbreviations when possible.
  Some exceptions can be made for well-accepted abbreviations. Example: `ip`, or names such as `os`, `geo`.

- 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

The predefined data types. The data type of a value specifies what kind of data that value can have. Note type^{O} denotes an observable type. _t attributes in parentheses denote internal JSON schema type notation.

Attribute	Base Type	Constraint	Description
boolean_t		false, true	Boolean value. One of `true` or `false`.
email_t^{O}	String	^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$	Email address. For example: `john_doe@example.com`.
file_hash_t^{O}	String	Max length: 64	File hash. A unique value that corresponds to the content of the file.
file_name_t^{O}	String	^[a-zA-Z0-9._ -]+$	File name. For example: `text-file.txt`.
float_t			Real floating-point value. For example: `3.14`.
hostname_t^{O}	String	^(([a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9\-]*[a-zA-Z0-9])\.)*([A-Za-z0-9]|[A-Za-z0-9][A-Za-z0-9\-]*[A-Za- z0-9])$	Unique name assigned to a device connected to a computer network. A domain name in general is an Internet address that can be resolved through the Domain Name System

Attribute	Base Type	Constraint	Description
			(DNS). For example: `r2-d2.example.com`.
ip_t^{O}	String	**Max length: 40** /^(?>(?>([a-f0-9][{1,4})(?>:(?1))){7}|(?!(?:.*[a-f0-9](?>:|$)){8,})((?1)(?>:(?1)){0,6})?::(?2)?)\|(?>(?>(?1)(?>:(?1)){5}:\|(?!(?:.*[a-f0-9]:){6,})(?3)?::(?>((?1)(?>:(?1)){0,4}):)?)?(25[0-5]\|2[0-4][0-9]\|1[0-9][{2}\|[1-9]?[0-9])(?>\.(?4)){3}))$/iD	Internet Protocol address (IP address), in eitherIPv4 or IPv6 format.
port_t	Integer	0-65,535	IP TCP/UDP port number. For example: `80` or `22`.
integer_t			Signed integer value.
json_t			Embedded JSON value. A value can be a string, or a number, or true or false or null, or an object or an array. These structures can be nested. See www.json.org.
long_t			8-byte long, signed integer value.
mac_t^{O}	String	**Max length: 32** ^([0-9A-Fa-f]{2}[:-]){5}([0 - 9A-Fa-f]{2})$	Media Access Control (MAC) address. For example: `18:36:F3:98:4F:9A`.
object_t			Object is an unordered set of name/value pairs. For example:`{ip: 92.24.47.250, type: IP Address}`
path_t	String	^[\pL0-9_]+[\pL0-9 ~!@#%&*\-./_]*$	File or folder full path name. For example: `/home/user/tmp/text-file.txt`.
process_name_t^{O}	String		Process name. For example: `Notepad`.
resource_uid_t^{O}	String	**Max length: 64**	Resource unique identifier. For example, S3 Bucket name or EC2 Instance ID.
string_t		**Max length: 65,535**	UTF-8 encoded byte sequence.

Attribute	Base Type	Constraint	Description
subnet_t	String	**Max length: 40**	Subnet mask in Classless Inter-Domain Routing (CIDR) notation. For example `192.168.200.0/24`.
datetime_t	String	^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(?:.\d+)?[A-Z]?(?:[.-](?:08:\d{2}|\d{2}[A-Z]))?$	The user-friendly time format as defined by RFC-3339. For example `1985-04-12T23:20:50.52Z`.
timestamp_t	Long		The time format is the number of milliseconds since the Epoch 01/01/1970 00:00:00 UTC. For example `1618524549901`.
url_t^{O}	String		Uniform Resource Locator (URL) string. For Example: `http://www.example.com/download/trouble.exe`.
username_t^{O}	String		User name. For example: `john_doe`.

Appendix C - Schema Construction and Extension

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

The repository is structure is as follows:

categories.json

dictionary.json

ocsf-schema

ocsf-schema/enums

ocsf-schema/events

ocsf-schema/extensions

ocsf-schema/includes

ocsf-schema/objects

ocsf-schema/profiles

ocsf-schema/templates

The following is extracted from CONTRIBUTING.md:

How do I add an event_class?

1. Determine all the `attributes` (including fields and objects) you would want to add in the `event_class`
2. Check the dictionary and the /objects folder, many of your desired attributes may already be present.
3. Define the missing attributes → Adding a `field` , Adding an `object`.
4. Determine which category you would want to add your event_class in, note it's `name`
5. Create a new file → `<event_class_name.json>` inside the category specific subfolder in the /events folder. Template available here
6. Define the `event_class` itself → Adding an `event_class`.
7. Finally, verify the changes are working as expected in your local ocsf-server.

Adding/Modifying an `attribute`

1. All the available `attributes` - `fields` & `objects` in OCSF are and will need to be defined in the attribute dictionary, the dictionary.json file and /objects folder if defining an object.
2. Determine if a new attribute is required for your change, it might already be defined in the attribute dictionary and/or the /objects folder.
3. Before adding a new attribute, review OCSF grammar & conventions || TASK - Add a grammar.md

How to define a field in the dictionary?

To add a new field in OCSF, you need to define it in the dictionary.json file as described below.

Sample entry in the dictionary -

```
"uid":
 {
   "caption": "Unique ID", // "previously name"
   "description": "The unique identifier. See specific usage.",
   "type": "string_t"
 }
```

Choose a unique field you want to add, `uid` in the example above and populate it as described below.

1. `caption` ➜ Add a user friendly name to the field.
2. `description` ➜ Add concise description to define the attributes.
    i. Note that `field` descriptions can be overridden in the `event_class/object`, therefore if it's a common field (like name, label, uid etc) feel free to add a generic description, specific descriptions can be added in the `event_class/object` definition. For example,
    ii. A generic definition of `uid` in the dictionary -
        a. `uid : The unique identifier. See specific usage.`
    iii. Specific description of `uid` in the `vulnerability` object -
        a. `uid : Unique Identifier/s of the reported vulnerability. e.g. CVE ID/s"`
3. `type` ➜ Review OCSF data_types and ensure you utilize appropriate types while defining new fields. | not allowed to change...
    i. All the available data_types can be accessed here || TASK - Create a data_types.md file in the repo
    ii. They are also accessible in your local instance of the ocsf-server - http://localhost:8000/data_types
4. `is_array` ➜ This a boolean key:value pair that you would need to add if the field you are defining is an array.
    i. e.g. `"is_array": true`

How to define an object?

1. All the available `objects` need to be defined as individual field entries in the dictionary, the dictionary.json file and as distinct .json files in the /objects folder.
2. Review existing Objects, determine if a modification of the existing object would be sufficient or if there's a need for a completely new object.

A sample .json object file,

```
{

  "caption": "Vulnerability Details", // "previously name"

  "name": "vulnerability", // "previously type"

  "description": "The vulnerability object describes details related to the observed vulnerability.",

  "extends": "object",

  "attributes": {

    "desc": {
```

```
      "description": "The description of the vulnerability",
      "requirement": "recommended"
    },
    "kb_articles": {
      "requirement": "optional"
    }
  }
}
```

1. Create a new file ➜ `<object_name.json>` in /objects folder.
2. Use the template available here, to get started with .json file definition.
3. `caption` ➜ Add a user friendly name to the object
4. `description` ➜ Add a concise description to define the object.
5. `extends` ➜ Ensure the value is `object` (All objects in OCSF extend a base definition of object)
6. `name` ➜ Add a unique name of the object
7. `attributes` ➜ Add the attributes that you want to define in the object,
    i. `requirement` ➜ For each attribute ensure you add a requirement value. Valid values are `optional`, `required`, `reserved`, `recommended`

Sample entry in the dictionary,

```
  "vulnerability":
    {
      "caption": "Vulnerability", // "previously name"
      "description": "The vulnerability object describes details related to the observed vulnerability",
      "type": "vulnerability"
    }
```

Choose a unique object you want to add, `vulnerability` in the example above and populate it as described below.

1. `caption` ➜ Add a user friendly name to the object
2. `description` ➜ Add a concise description to define the object.

3. `type` ➜ Add the type of the object you are defining.
4. `is_array` ➜ This a boolean key:value pair that you would need to add if the object you are defining is an array.
    i. e.g. `"is_array": true`

Adding/Modifying an `event_class`

1. All the available Event Classes are defined as .json files in the /events folder.
2. Review existing Event Classes, determine if a modification of the existing class would be sufficient or if there's a need for a completely new event_class.
3. To define a new class,
    i. Create a new file ➜ `<event_class_name.json>` inside the category specific subfolder in the /events folder.
    ii. Use the template available here, to get started with the .json definition.
    iii. `uid` ➜ Select an integer in the range 0 - 99. Ensure the integer is unique within the category.
        a. Note: Without `uid`, an event_class won't be visible in the ocsf-server.
    iv. `caption` ➜ Add a user friendly name to the event_class.
    v. `description` ➜ Add a concise description to define the attributes.
    vi. `name` ➜ Add a unique name of the event_class. Ensure it matches the file name to maintain consistency.
    vii. `extends` ➜ Ensure the value is `base_event`.
    viii. `attributes` ➜ Add the attributes that you want to define in the event_class,
        a. `group` ➜ For each attribute ensure you add a group value. Valid values are - `classification`, `context`, `occurrence`, `primary`
        b. `requirement` ➜ For each attribute ensure you add a requirement value. Valid values are `optional`, `required`, `reserved`, `recommended`

Extending the Schema

To extend the schema create a new directory in the `schema/extensions` directory. The directory structure is the same as the top level schema directory and it may contain the following files and subdirectories:

categories.json	Create it to define a new event category to reserve a range of class IDs.
dictionary.json	Create it to define new attributes.
events/	Create it to define new event classes.

objects/	Create it to define new objects.

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