EUSEC
EU SECURITY CERTIFICATION

European Security Certification Framework

D3.1 Architecture for security controls 1.0

PROJECT NUMBER: 731845

PROJECT TITLE: EU-SEC

DUE DATE: 31/12/2017	DELIVERY DATE: 28/12/2017
AUTHOR: A. PANNETRAT	PARTNERS CONTRIBUTED: CSA, Fraunhofer
DISSEMINATION LEVEL: PU	NATURE OF THE DELIVERABLE: R
INTERNAL REVIEWERS: Fraunhofer, MFSR	

*PU = Public, CO = Confidential

**R = Report, P = Prototype, D = Demonstrator, O = Other

This project has received funding from
the European Union's HORIZON Framework
Program for research, technological development and
demonstration under grant agreement no 731845

EXECUTIVE SUMMARY

Any form of certification requires stakeholders to specify the scope of what is certified. In the context of continuous auditing based certification, we need to specify what controls are being audited, when they are being audited (frequency), how they are being audited and to which information system they apply (the subject of the audit). Since the EU-SEC project aims to use automated auditing whenever feasible, thereby eliminating or reducing human intervention, this information needs to be machine-readable whenever necessary in order to be used to configure relevant automated auditing tools.

This deliverable proposes a data structure that addresses those needs.

The JSON data structure we propose in this document is called a "certification objective". The root of this hierarchical data structure specifies the subject of the assessment (e.g. a cloud service) and is matched with a list of requirements, which is typically a set of "control objectives" from a standard such as the CCM or ISO/IEC 27001. These requirements are further broken down into more concrete technical objectives (SLOs and SQOs) that must be evaluated individually on a pre-defined frequency. The data structure makes a particular distinction between "automated objectives" and "assisted objectives". Automated objectives represent properties that can be fully assessed without human intervention, whereas "assisted objectives" are objectives that require human assessment. Combining these two types of objectives allows us to cover all audit tasks in a certification process.

The end of the document provides a preliminary specification of the protocols that are used to transfer this data structure to the various tools in the EU-SEC toolset, namely CTPD, Clouditor and STARwatch.

ABBREVIATIONS

API	Application programming interface
CTP	Cloud Trust Protocol
CTPD	Cloud Trust Protocol Daemon
CCM	Cloud Control Matrix
CSA	Cloud Security Alliance (https://cloudsecurityalliance.org/)
CAIQ	Consensus Assessments Initiative Questionnaire (CAIQ)
GDPR	General Data Protection Regulation EU (2016/679)
ISMS	Information Security Management System
ISO/IEC 27001	ISO/IEC 27001:2013 Information technology – Security techniques - Information security management systems - Requirements
JASON	JavaScript Object Notation
PLA	Privacy Level Agreement
SLO	Service Level Objective (see D1.4)
SQO	Service Qualitative Objective (see D1.4)
TLS/SSL	Transport Layer Security/ Secure Sockets Layer
TRL	Technology Readiness Level

TABLE OF CONTENTS

1.1 OBJECTIVES OF THE DELIVERABLE 8

1.2 HIGH LEVEL EXAMPLE 8

1.3 ORGANISATION OF THIS WORK 10

2.1 PRODUCT AND TOOLS 11

2.1.1 CSA Cloud Trust Protocol and ctpd 11

2.1.2 ClouditOR 13

2.1.3 CSA Starwatch 15

2.2 RESEARCH 16

2.2.1 The CUMULUS project 16

2.2.2 The SPECS project 17

2.2.3 Next Generation Certification (NGCert) 17

3.1 OVERVIEW OF REQUIREMENTS 18

3.2 CLASS DIAGRAM 18

3.3 CONVENTIONS 22

3.4 JSON REPRESENTATION 24

3.4.1 Certification objective 24

3.4.2 Assisted assessment 26

3.4.3 Automated assessment 27

3.5 DEFINING AUTOMATED OBJECTIVES 30

3.6 JAVASCRIPT LIMITED BOOLEAN EXPRESSIONS 31

3.6.1 Basic Grammar 32

3.6.2 Definitions borrowed from other standards 34

3.6.3 Types 34
3.6.4 Functions 35
3.6.5 Boolean operators ||, && 39
3.6.6 Comparison Operators <, <=, >, >=, ==, != 40
3.6.7 Multiplicative and additive binary operators: +, -, *, /, % 41
3.6.8 Identifiers 42
3.6.9 Evaluating JSLBE expressions 43
4.1 OVERVIEW 44
4.2 CONTINUOUS CERTIFICATION CONFIGURATOR 45
4.3 INTERFACING WITH CTPD 45
4.3.1 Conventions used in the API 46
4.3.2 Uploading a Certification objective 47
4.3.3 Getting information about a certification objective 48
4.3.4 Removing a certification objective 48
4.4 INTERFACING WITH CLOUDITOR 49
4.4.1 Basic communication between continuous certification configurator and Clouditor 49
4.4.2 ReSTfulL API Spec 50
4.4.3 Advanced protocol 50
4.5 INTERFACING WITH STARWATCH 50

LIST OF TABLES

N/A

LIST OF FIGURES

FIGURE 1: CLASS DIAGRAM OF THE CTP DATA MODEL 12
FIGURE 2: COMPONENTS OF THE CLOUDITOR TOOLBOX 14
FIGURE 3: OVERVIEW OF CLOUDITOR ENGINE MAIN COMPONENTS (WITH EXTERNAL TEST TOOL) 15
FIGURE 4: CLASS DIAGRAM OF THE CERTIFICATION OBJECTIVE DATA MODEL 19
FIGURE 5: CONTINUOUS CERTIFICATION CONFIGURATION 44

1 INTRODUCTION

1.1 OBJECTIVES OF THE DELIVERABLE

Any form of certification requires stakeholders to specify the scope of what is certified. In the context of continuous auditing based certification, we need to specify:

- **Who**: the scope of the information system that is scrutinized (e.g. a specific cloud service).
- **What**:
  - What is the list of security and privacy *requirements* (control objectives) that are applicable to the *subject*?
  - What are the *objectives* (SLOs and SQOs) related to each one of these *requirements* that can be evaluated on a continuous basis?
- **How**: what metrics are used to evaluate relevant security and privacy *objectives*.
- **When**: how often should each *objective* be assessed, in order to be sufficiently confidant that the *subject* satisfies the objectives over time while taking into account the constrains of the assessment tools and processes.

This deliverable proposes a data structure that specifies these elements, in a way that they can be used by various tools in the EU-SEC tool-chain. In addition, we also specify how this data structure, called a "certification objective" is transferred to the various tools in the EU-SEC tool-chain, which include CTPD, Clouditor and STARwatch.

Terminology is based on deliverable D1.4.

1.2 HIGH LEVEL EXAMPLE

Consider a PaaS called "WebMaker" which allows customers to build secure web services, to host applications such as a corporate client portal. A risk assessment conducted for "WebMaker" identified the need to implement the following control from CSA CCM (truncated here for simplicity):

EKM.04 "Platform and data-appropriate encryption (e.g., AES-256) in open/validated formats and standard algorithms shall be required. […]"

Note that in a real world scenario, "WebMaker" would have to implement several dozens of controls like EKM-04. We choose only one for simplicity.

This control is specific to the web frontend of "WebMaker" with public IP address 172.217.23.110, which must provide secure HTTP with an up-to-date version of TLS, using cryptographic algorithms that provide medium term security given the current state of art. One way to measure cryptographic strength is key length[1]. Unfortunately, a protocol like TLS employs several types of cryptographic mechanisms, notably symmetric encryption, asymmetric encryption and message authentication codes, each with different types of key lengths. A better metric for cryptographic strength is to rate the strength of the underlying cryptographic algorithms on a normalized scale or in terms of years of "brute force resistance". In this example we will say that the TLS connection should be rated at a security level of 6 or above according to the ECRYPT II scale [ECRYPT], which is based on academic consensus from cryptographic researchers. In the context of an audit, we could use an automated tool that would try all available cipher suits accepted by the end-point, and compute a measurement based on the lowest score obtained for any cryptographic algorithms used in the cipher suits.

The elements above could be part of a continuous certification process, which would aim to assess the TLS connection security on a daily basis (or even more frequently).

Let's take the story above and structure it into a description of a certification objective:

- The subject of the certification is the PaaS "WebMaker", owned by "WebMaker Inc.". It notably encompasses the cloud based PaaS platform hosting the service.
    - Assuming this is a self-assessment, the auditee and the auditor are both "WebMaker Inc.".
- One of the key requirements that apply to this service is the implementation of control EKM-04, from the CSA CCM version 3.0.1.
- This requirement is translated to the objective of achieving a security level of 6 or above on the TLS connection provided to customers.
    - The security attribute we want to measure is "cryptographic strength".
    - The measurement used to evaluate the security level is based on the metric defined by [ECRYPT], using the lowest score approach mentioned earlier.
    - The asset this measurement applies to is the TLS connection of the webserver with IP address 172.217.23.110 (port 443).

[1] In practice the security of a TLS connection depends on much more than key length, and notably depends on implementation quality and correct configuration. We only use key length here as an illustration.

- The frequency of assessment of this objective is: every 24 hours.

The data structure proposed in this document aims to convey all this information, by following a hierarchy that is quite similar to the one exposed in the example above.

1.3 ORGANISATION OF THIS WORK

The remainder of this document is divided in three main parts:

- Section 2 reviews existing data formats, relevant tools and prior research form partners that are relevant to the objectives of this work.
- Section 3 lays down our proposed data structure.
- Section 4 describes how this data structure will be used with the different tools in the EU-SEC continuous auditing toolchain.

2 STATE OF THE ART

This section reviews existing data formats that are used by project partners as part of their products or research activities, in relationship with the exchange of information related to security and privacy requirements and assessments.

The data formats and protocols defined in this deliverable aim to take into account existing tools by EU-SEC partners, most notably **CTPD**, **Clouditor** and **STARwtach** as defined in the EU-SEC toolchain (see [D3.3]). While these tools will surely need to be extended to take into account the requirements of the EU-SEC project, we attempt to build upon existing and established technologies wherever possible.

We will also briefly recall some prior research oriented projects, which have had an important influence on the tools and the underlying concepts and technologies we will use in this deliverable.

2.1 PRODUCT AND TOOLS

2.1.1 CSA CLOUD TRUST PROTOCOL AND CTPD

The Cloud Trust Protocol is a RESTful API specification [CTP] that is designed to allow cloud customers to query cloud providers about the current status of their security for the purpose of continuous monitoring. The current API represents the third generation of the CTP specification, which is based on work that was initiated in 2010 and which was recently "rebooted" with novel ideas that were notably borrowed from the EU-funded project CUMULUS [CUMULUS] and other follow-up activities.

As a proof of concept, CSA recently created a prototype of the RESTful API: the CTP daemon[2] or "CTPD", which was first used and tested in the EU funded SPECS project [SPECS]. In addition to the implementation of the "official" CTP API, **CTPD** also provides a "non-official" administrative/configuration API and an optional lightweight browser client that enables users to explore the API without installing software locally.

[2] https://github.com/cloudsecurityalliance/ctpd

The CTP specification defines data formats for the representation of information such as "security attributes", "objectives" or "metrics" that share many commonalities with the concepts used in the EU-SEC project. Conceptually, CTP defined a hierarchy of data elements, which are summarized on the following class diagram.

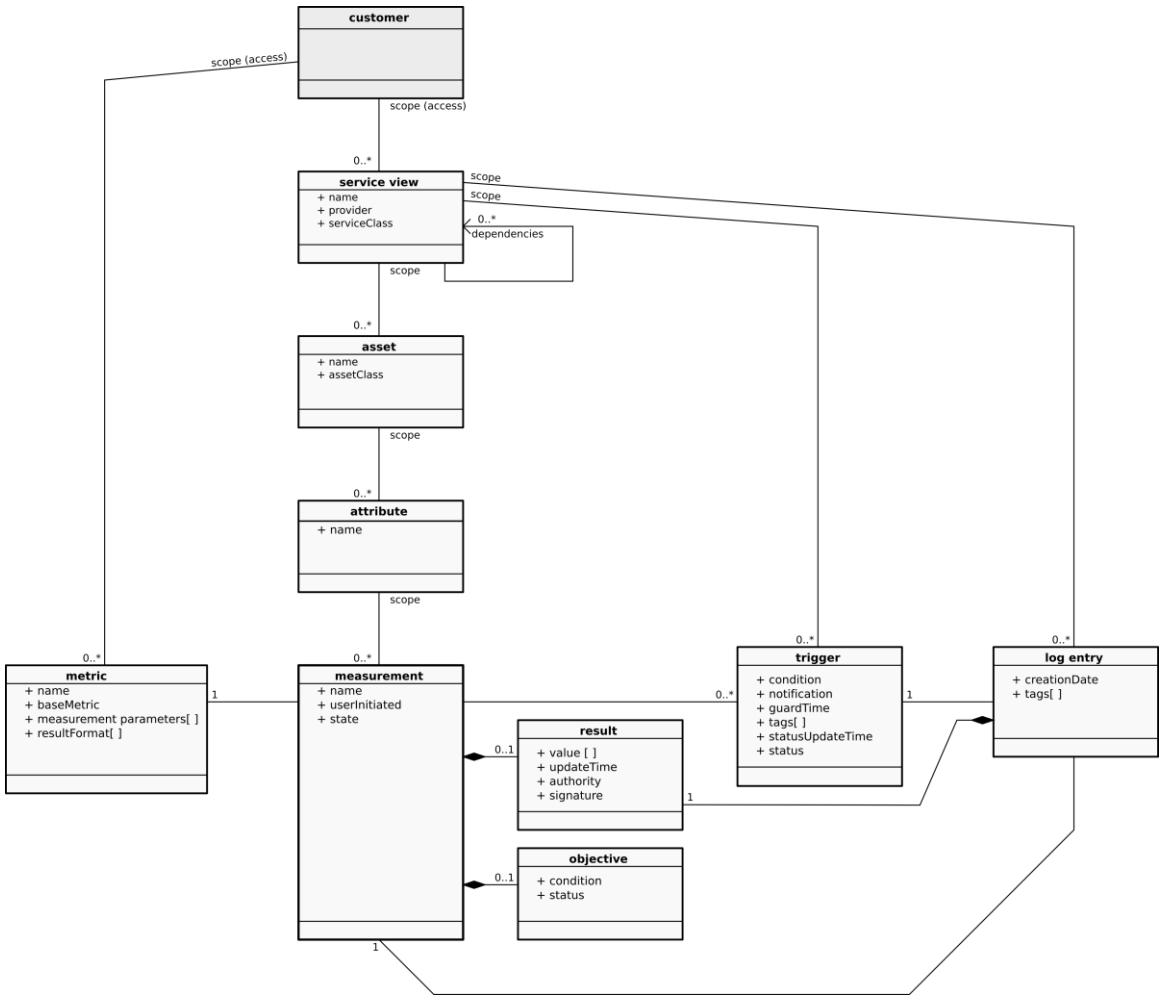

Figure 1: Class diagram of the CTP data model

A detailed review of this diagram is beyond the scope of this work. We can however summarise the core hierarchy of resources that serve as the foundation of the CTP data model:

- CTP defines a **service view** as the data element that represents a cloud service. Each service view is broken down into a set of "assets".
- An **asset** defines a subset of an information system that is subject to specific security measures (e.g. a virtual machine, a storage bucket, a database, etc.). Assets have one or more security or privacy "attributes".

- An **attribute** defines a measurable characteristic of an asset (e.g. "uptime", "cryptographic strength", etc.). Attributes are evaluated through one or more "measurements".
- A **measurement** defines of process for evaluating evidence and obtaining a "measurement result", which is a value (i.e. a string, number or Boolean) that can be later assessed against an "objective". A measurement references an external "metric".
- An **objective** represents a constraint on a measurement value: when this constraint is violated (when it's evaluated to "false") the objective is considered as failed. Objectives are expressed through a JavaScript expression.

Through this hierarchy, CTP offers the basic structure needed to represent the result of the automated assessment of controls.

As detailed in deliverable [D3.3], it is sensible to use **CTPD** in the EU-SEC project as a tool that collects input from other tools and evaluates objectives and provides feedback to cloud users, based on the CTP API and data model. As such, while CTP does not address non-automated objectives, we consider that it is useful to re-use some of the ideas behind the CTP data model in the data structure we define in this document. At a very minimum, we aim to make our data model compatible with CTP since it could be used to configure the **CTPD** tool itself.

2.1.2 CLOUDITOR

This section outlines the Clouditor-engine which is part of the Clouditor toolbox. The Clouditor toolbox is part of the background of the EU-Sec project, it consists of five main components – Engine, Explorer, Simulator, Evaluator & Dashboard – which are shown in *Figure 2*.[3] The engine is responsible for executing test-based measurement techniques, that is, continuous tests of cloud service security properties.

[3] For further information about the Clouditor toolbox see https://www.aisec.fraunhofer.de/de/fields-of-expertise/projekte/Clouditor.html.

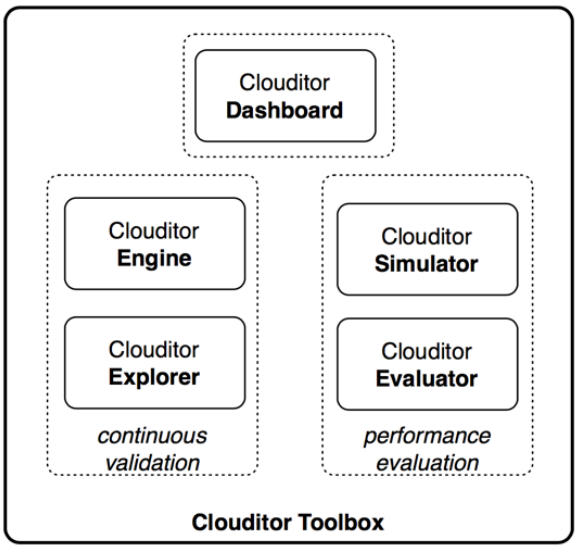

Figure 2: Components of the Clouditor toolbox

Figure 3 shows a high level architecture of the Clouditor Engine's components, including data and control flow. Test case are implemented using hooks to existing security tools such as Nmap[4], SQLMap[5], sslyze[6] etc, there reusing existing knowledge and tooling. Alternatively, test cases can be implemented natively and self-contained as part of the Engine.

[4] https://nmap.org/
[5] http://sqlmap.org/
[6] https://github.com/nabla-c0d3/sslyze

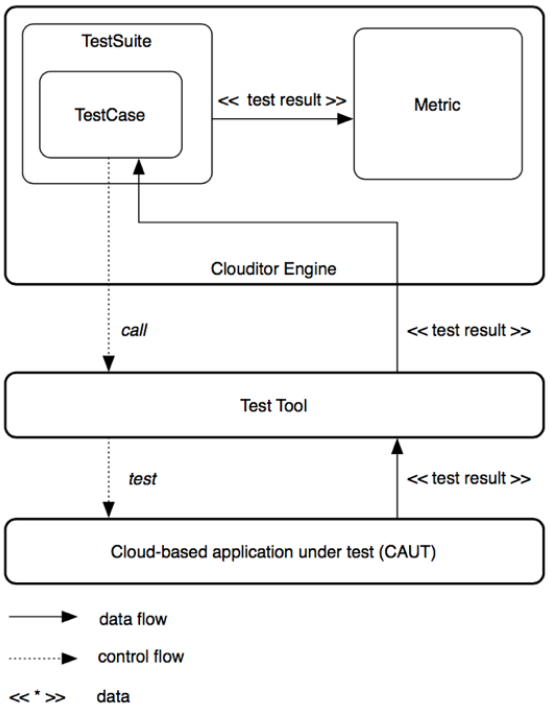

Figure 3: Overview of Clouditor Engine main components (with external test tool)

2.1.3 CSA STARWATCH

CSA STARwatch is a cloud assessment SaaS application based on the CSA CAIQ questionnaire, which itself is derived from the CSA CCM standard, used as a foundation in the EU-SEC project. The CAIQ simply translate each individual CCM control into 1 or more questions that can be answered "yes", "no" or "not applicable": see [CAIQ] for details. Before the creation of STARwatch, users would typically do self-assessments with plain Excel spreadsheets. With STARwatch, users perform assessments online and can benefit from smart support features such as track-changes or cross matching of controls across multiple standards. In the future months, STARwatch will likely be extended with other types of assessments, most notably the PLA v3 (see [PLA]), for compliance with data protection regulation as defined in the GDPR and Dir. 95/46/EC.

By construction, STARwatch is designed for human-based (non-automated) assessments. Each CAIQ question that is applicable to an information system is designed to be answerable by "yes" or "no". A CAIQ question can therefore be viewed as a crude expression of an SQO (Service Qualitative Objective). With this in mind, consider the following possible extension of STARwatch:

- A policy defines:

    - A scope as a list of applicable CAIQ questions.
    - An "update interval" for each CAIQ control (e.g. "a month") within the scope.
- Users performing assessments have their responses monitored by the STARwatch application.
    - If the user updates all CAIQ responses within the timeframe defined in the policy and if the responses are "yes", the assessment is marked as "SUCCESS".
    - If the user fails to update at least one response within the timeframe defined in the policy or if at least one response is "no", the assessment is marked as "FAILED".

While some details need further specification, this extension offers a primitive framework for continuous auditing for requirements that require human assessment. With this in mind, the data structure proposed in this document must therefore provide machine-readable information that can be used to propose such a framework. Despite the fact that some assessments require human intervention, their assessment can still benefit from machine-readable data specifying the policy we defined in our extension above.

2.2 RESEARCH

2.2.1 THE CUMULUS PROJECT

The CUMULUS project [CUMULUS] was designed to develop "*an integrated framework of models, processes and tools supporting the certification of security properties of infrastructure (IaaS), platform (PaaS) and software application layer (SaaS) services in cloud*". CSA was one of the partners in this project launched in 2012. CUMULUS proposed to define a framework for the continuous certification of cloud services. In many ways, CUMULUS pioneered some of the concepts and ideas behind "continuous auditing" in ways that have influenced several other EU-funded project including EU-SEC project.

CUMULUS defined a certain number of data structures, including artefacts necessary for the representation of certificates, security properties, security SLAs, and assessment tool configuration. Most of these data structures can be found in [CUMULUS-D2.3]. CUMULUS also explored different solutions to express security "assertions" (i.e. SLOs and SQOs as defined EU-SEC) through the use of formal languages with temporal logic such as SLA* [SLA@SOI] and SecureSLA [CUMULUS-D2.3]. The sophisticated models and languages used in CUMULUS were quite ambitious but suffer from a strong level of complexity that might discourage adoption in

real-world scenarios. The EU-SEC goals are more pragmatic and notably aim to develop real world tools with defined Technology Readiness Levels (TLR). As a consequence, we chose not to re-use the data models of CUMULUS directly. However, many ideas initiated in CUMULUS have influenced the design of the data model we propose in this deliverable, notably because the CTP (see 2.1.1) itself was influenced by CUMULUS.

2.2.2 THE SPECS PROJECT

The SPECS project aimed "*at developing and implementing an open source framework to offer Security-as-a-Service, by relying on the notion of security parameters specified in Service Level Agreements (SLA), and also providing the techniques to systematically manage their life-cycle*". CSA was one of the partners in this project launched in 2013. Though this project was not about certification directly but about continuous monitoring Cloud SLAs in the context of security, it shares some common goals with the EU-SEC project.

This project was notably the opportunity to enhance and test **CTPD** (see 2.1.1) in an environment that mimicked a real cloud provider.

2.2.3 NEXT GENERATION CERTIFICATION (NGCERT)

NGCert is a research project funded by the by the Federal Ministry of Education and Research (BMBF) of Germany. Within NGCert, methods and tools have been developed which allow to continuously validate whether a cloud services adheres to a set of controls. NGCert is part of the "Secure Cloud Computing" program, which is derived from the so-called Hightech-Strategy by the German government.

Parts of the concepts which are implemented by the **Clouditor** have been developed within NGCert.

3 DATA FORMAT

3.1 OVERVIEW OF REQUIREMENTS

This deliverable aims to develop data structures needed to exchange information about "controls" between tools, in the context of continuous auditing. However "controls" do not exist in a limbo and must be considered within the context of continuous auditing-based certification: selected "controls" apply to a specific scope, they are translated into SLOs/SQOs and consumed by tools either for the purpose of automated assessments or for the purpose of human assisted assessments, involving auditees and auditors. In other words, the proposed data structure should describe the "what", "who", "how", and "when" we are certifying.

As a consequence, our data structure shall represent the scope of the continuous certification not only in terms of "controls" but also in terms of:

1) Information system perimeter.
2) Auditees and auditors.
3) Objectives (SLO/SQO), attributes and metrics.
4) Frequencies of assessment.

The data structure must be useful for both automated assessments as well as assessments that will require human intervention.

We aim wherever possible to make this data structure compatible with the existing tools that are used in the project in order to minimize integration/adaption efforts. Data structure elements should be machine-readable whenever possible, but should remain as simple as possible and in line with the capabilities of the tools that exploit them.

3.2 CLASS DIAGRAM

Based on the requirements previously described, we propose the following data structure, which is summarized in the following class diagram and then further detailed hereafter.

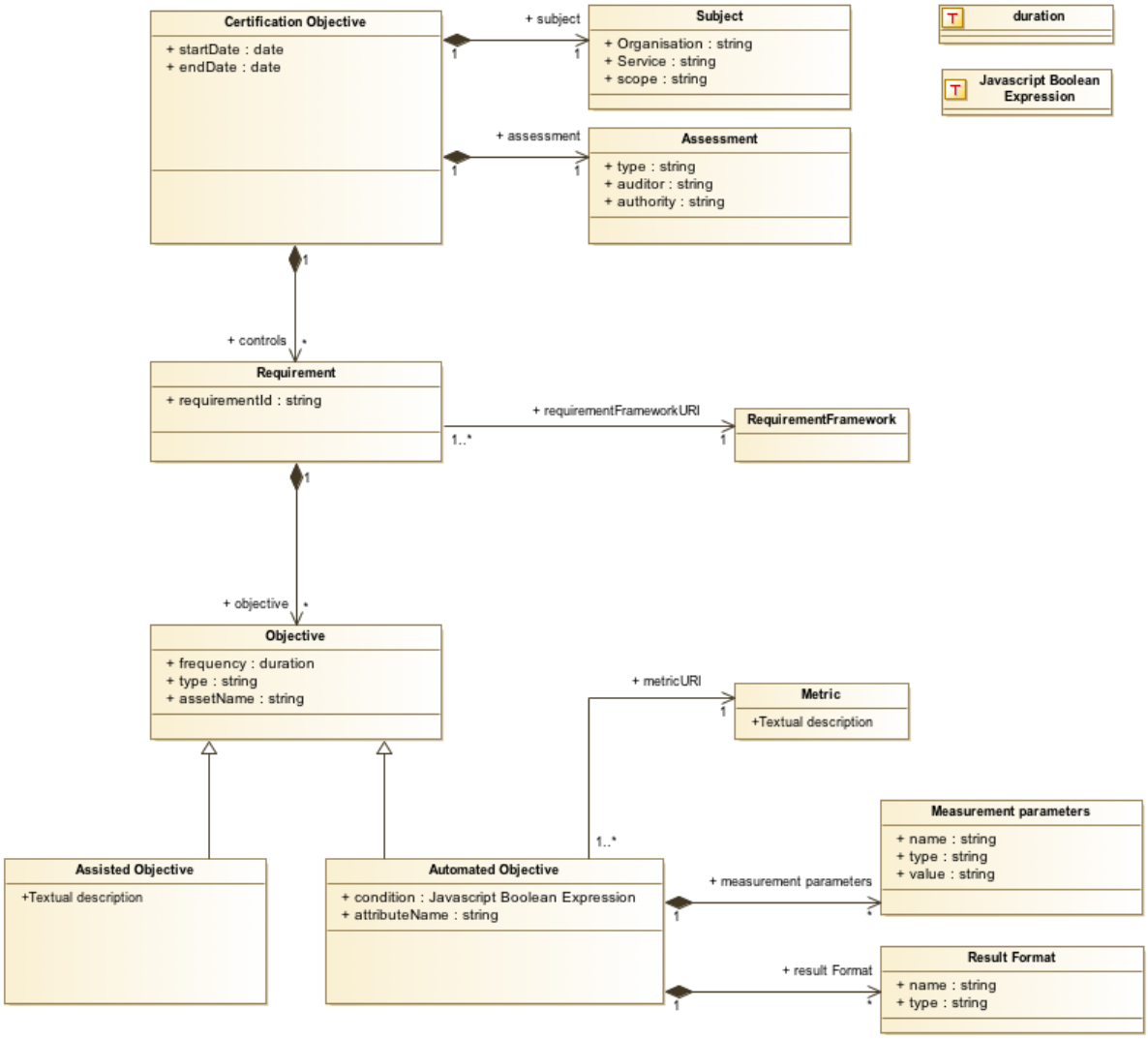

Figure 4: Class diagram of the certification objective data model

Let's walk through the different elements of this diagram.

Certification objectives

The "root" of our class diagram is the "certification objective", which is also the name we give to the whole structure represented in this diagram.

A certification objective specifies:

- The start and end date for the continuous certification process.
- The scope of the certification process (the subject).

- The type of the assessment (e.g. self-assessment) and parties involved (who is doing the continuous auditing and who decides if the certification is valid).

Each certification objective specifies a list of requirements.

Requirements

Requirements refer to the high-level statements that must be satisfied in order to consider that the subject is certified. Requirements are expected to be generic and technology independent. This is typically described as "control objectives" found in security standards such as ISO 27001 or CSA CCM[7], but it can also refer to legal texts (e.g. GDPR). A requirement will refer to:

- A requirement framework (ISO 27001, CSA CCM, …)
- A requirement id, referencing a requirement within the selected framework.

Each requirement is translated into a set of objectives.

Objectives

Objectives define qualitative or quantitative constraints that must be verified in order to consider that a requirement is correctly implemented. Whereas requirements are generic high-level statements, objectives are designed to concretely specify security or privacy constraints on an information system in ways that can be evaluated qualitatively or quantitatively (See "SLO" and "SQO" in [D1.4]). A single "high-level" requirement is likely broken down in a set of distinct objectives.

In the context of continuous auditing, objectives must be assessed according to a predefined frequency, which is defined as part of the objective.

High-level security and privacy requirements are typically defined to apply globally to an ISMS, or at least within the specified scope of the certification subject. Once requirements (control objectives) are translated into objectives, they become more specific and their scope will typically become narrower, targeting a specific component of an information system. For example, consider the specification of a certain level of cryptographic strength for TLS/SSL connections (as in 1.2): this constraint will likely be scoped to the webservers that are facing the Internet and may not apply to internal data transfers on a protected network. Depending on the context, an asset may be specified through identifiers such as IP addresses, virtual machine identifiers, storage volume URIs, or through broader categories such as compute, database, etc. Of course, some objectives (e.g. data related to incident response performance)

[7] CCM uses the term "control" to refer to a "control objective".

might still apply to the IMS as a whole without targeting a specific component of an information system.

In the certification objective defined in this document, objectives can be associated with an asset through the "asset name" property. When necessary, this is property is used to identify a relevant asset to which the objective applies. When this property is left blank, the objective is considered to apply globally to the information system that is defined as the subject of the certification.

Objectives come in two flavors: (1) Assisted objectives and (2) Automated objectives, which are both described hereafter.

Assisted objectives

An assisted objective is an objective that cannot be fully assessed by automated means and therefore requires a human assessment.

As such it does not have any machine-readable properties beyond the frequency and asset name, which are part of all objectives. It contains a description that is intended for human interpretation, describing the objective to the assessor.

Whilst assisted objectives are intended to be evaluated by humans, automated tools can still monitor their frequency of assessment. We exploit this fact in the EU-SEC toolset (see 4.5).

Automated objectives

An automated objective is an objective that will be evaluated automatically (e.g. with Clouditor and CTPD in EU-SEC). As a consequence, automated objectives are described with a more complex structure, which refers to "attributes", "metrics", "measurement results" and "measurement parameters".

The central element of an automated objective is its "condition": a JavaScript expression that is evaluated to either True or False. This JavaScript expression describes the objective by combining variables and operators (e.g. "uptime[0]>99.95"). The assessment tools populate the variables with values and then this expression is evaluated to establish whether the objective is met or not. Section 3.6 provides a definition of JavaScript expressions.

Automated assessments have a property called "attribute name" which enables to provide a name for the security or privacy attribute that is being evaluated as part of an SLO or SQO (e.g. "monthly uptime"). Tools that use this data structure can use this name to display or report information to end-users.

The process of evaluating an attribute results in the production a measurement result, as part of a measurement, as defined in a "metric" (i.e. a standard for measurement [D1.4]). Objectives refer to a metric through a URI, which typically points to a textual description of the metric, notably specifying rules for the collection of evidence, calculations, units, etc.

Ultimately, the measurement process defined in the metric specifies how to obtain a qualitative or quantitative "measurement result" (e.g. numbers, strings, Booleans, etc.). In order for the JavaScript expression to be used, we need to put those measurement results in a variable. As such, a measurement result is represented by a table where each column has a name and a type, and where each row has a number starting from 0 (see 3.6.8 for more details). The names and types of the columns of the table are defined with a list of "result formats", i.e.

- The first result format defines the name and type of the first column.
- The second result format defines the name and type of the second column.
- The n^{th} result format defines the name and type of the n^{th} column.

This may sound complicated but in the most basic cases, where a measurement result is just a single value, this table has just one cell (one row and one column). For example, if we measure "password minimal length", we might define a table with one column called "password_minimal_length" that has the type "number". Next the javascript expression describing the condition "passwords must be at least 8 characters" would simply be "password_minimal_length[0]>=8". In more complex cases, measurement results can hold multiple values, represented as key-value pairs, where each value can itself be a tuple (an array indexed from zero) if necessary.

In an ideal world, metrics would be directly applicable regardless of the underlying technical platform or architecture being used. In practice, the measurement process defined in the metric might need to be parameterized with system/platform/architecture specific values according to the operational environment. To allow this to happen, automated objectives may optionally refer to "measurement parameters", specifying a set of names, types and values that are used as input to the measurement process.

3.3 CONVENTIONS

In the following sub-sections, JSON objects will be described directly in a pseudo language that uses and extends JSON itself, and is mostly self-explanatory. This approach frequently

used in the industry (e.g. Google Cloud API[8]) was preferred over more formal approaches such as JSON schema[9] for readability and ease of use.

Types

A type is described by a name enclosed between "<" and ">". In addition to the standard JSON types <string>, <number> and <boolean>, we will also use the following base types:

- <long>: A integer number.
- <datetime>: A string representing a UTC timestamp as defined in ISO 8601, including the year, month, day, hour, minute and second, and ending with the 'Z' marker representing UTC time (e.g. 2016-09-29T13:11:43Z).
- <duration>: A duration as defined in ISO 8601, using the extended format P[YYYY]-[MM]-[DD]T[hh]:[mm]:[ss].
- <uri>: A string representing a URI (Uniform Resource Identifier), typically a URL.

Example:

```
{
  "name": <string>,
  "age": <long>
}
```

In addition to the primitive types above, this specification defines additional objects that are detailed each in their own sub-section. These objects are named with the same convention as above, with a type name enclosed between "<" and ">" (e.g. <assisted_assessment>).

Multiple choices

When an object can take several values or types, this choice is indicated with a pipe symbol "|" (e.g.

```
{
  "gender": "male" | "female"
}
```

Arrays

[8] https://cloud.google.com/storage/docs/json_api/
[9] http://json-schema.org/

When an array appears in a schema description, we only represent an example of the first element in the array, followed by an ellipsis ("…"). This means that the element may appear 0 or more times, unless otherwise specified in the description.

```
{
  "name": <string>,
  "phones": [
     {
        "type": <string>,
        "number": <string>
     },
     ...
  ]
}
```

3.4 JSON REPRESENTATION

This section defines the "certification objective" JSON data structure, which was initially represented as a UML class diagram in section 3.2. For clarity, the description of that data structure is broken down into 3 parts:

1) The "certification objective" object described in 3.4.1, which is the main data structure.
2) The "assisted assessment" object described in 3.4.2, which details the special case of representing objectives that cannot be automatically assessed.
3) The "automated assessment" object described in 3.4.3, which details the special case of representing objectives that are automatically assessed.

This three-part structure mirrors the structure of our UML diagram which has a common core, and two branches that join together through the class "Objective".

To be clear, despite being presented in 3 parts, the "certification objective" is designed to be used as a single JSON object that will be consumed by EU-SEC tools such as CloudAudit or StarWatch.

3.4.1 CERTIFICATION OBJECTIVE

JSON structure

```
{
    "certification_objective_id": <string>,
    "start_date": <datetime>,
    "end_date": <datetime>,
    "subject": {
        "organisation": <string>,
        "service": <string>,
        "scope”: <string>
    },
    "assessment":
        "type": <string>,
        "auditor": <string>,
        "authority": <string>
    }
    "requirements": [
        {
            "requirement_id": <string>,
            "requirement_framework": <uri>,
            "objectives": [
                <assisted_assessment> | <automated_assessment>,
                …
            ]
        },
        …
    ]
}
```

Details

Property	Description
certification_objective_id	Unique identifier of the certification objective. It may be left blank.
start_date	Start date of the continuous certification process. This date notably serves as a starting point from which the frequencies of the assessment of objectives are computed. (i.e. the “frequency” property in 3.4.2 and 3.4.3)
end_date	End date for the continuous certification process. A review of the scope and content and certification objective by the *subject* is recommended at this point. The end_date is typically expected to be set one year or more after start_date.
subject	-
subject.organisation	The organisation responsible for the service that is being audited.
subject.service	The service that is being audited.

subject.scope	Describes the effective scope of the continuous assessment in more detail, in terms of information system components and governance.
assessment	-
assessment.type	A string describing the assessment type: • "SelfAssessment": … • "ThirdParty": …
assessment.auditor	The identity of the organisation performing the assessments (it can be the same as the subject if it's a self-assessment).
assessment.authority	The identity of the organisation that publishes the status of the certificate based on the results of the continuous certification process, i.e. whether or not the subject is certified.
requirements[]	A list of requirements.
requirements[].requirement_id	An identifier that uniquely identifies a requirement with a requirement framework.
requirements[].requirement_framework	A pointer that uniquely identifies a requirement framework down to the version/revision level (e.g. an URL that points to the official specification of CCM version 3.0.1).
requirements[].objectives[]	This list consists of objectives that a requirement is translated to, in the context of continuous auditing. Each object in this list can either be: 1) An assisted_assessment described in 3.4.2. 2) An automated_assessment described in 3.4.3

3.4.2 ASSISTED ASSESSMENT

JSON structure

```
{
    "objective_id": <string>,
    "frequency": <duration>,
    "type": "assisted",
    "asset_name": <string>,
    "description": <string>
}
```

Details

Property	Description
objective_id	Unique identifier of the objective, within a certification objective scope. It may be left blank.
frequency	A duration describing how frequently an objective must be assessed (e.g. every month). When the assessment is not conducted within this predefined frequency, the objective is considered as (temporarily) failed. The starting reference point for calculating assessment deadline is the start_date property of the certification objective (see 3.4.1).
type	This property always has the value "assisted", and is used to distinguish assisted assessments from automated assessments.
asset_name	The name of the particular asset this objective applies to (within the scope of the subject of the certification). When this property is left blank, the full subject defined in the certification objective is considered as the evaluated asset.
description	A textual description of the assessment.

3.4.3 AUTOMATED ASSESSMENT

JSON structure

```
{
    "objective_id": <string>,
    "frequency": <duration>,
    "type": "automated",
    "asset_name": <string>,
    "metric": <uri>,
    "attribute_name": "string",
    "measurement_parameters": [
        {
            "name": <string>,
            "type": "number" | "long" | "boolean" | "string",
            "value": <number> | <long> | <string> | <boolean>
        },
        ...
    ],
    "result_format": [
        {
            "name": <string>,
            "type": "number" | "long" | "boolean" | "string"
        },
        ...
    ],
    "assertion": <string>
}
```

Details

objective_id	Unique identifier of the objective, within a certification objective scope. It may be left blank.
frequency	A duration describing how frequently an objective must be assessed (e.g. every month). When the assessment is not conducted within this predefined frequency, the objective is considered as (temporarily) failed. The starting reference point for calculating assessment deadline is the start_date property of the certification objective (see 3.4.1).
type	This property always has the value "automated", and is used to distinguish automated assessments from assisted assessments.
asset_name	The name of the particular asset this objective applies to (within the scope of the subject of the certification).

	When this property is left blank, the full subject defined in the certification objective is considered as the evaluated asset.
attribute_name	A unique name given to the attribute that is being assessed (e.g. “Monthly Uptime”)
metric	A URL uniquely identifying the metric that is used to assess/measure an attribute, in order to prove a measurement result.
measurement_parameters[]	A list of measurement parameters. Often this list should be empty.
measurement_parameters[].name	A string identifying a measurement parameter name.
measurement_parameters[].type	A type, which can be either "number", "long", "boolean", or "string".
measurement_parameters[].value	A value of type measurement_parameters[].type
result_format[]	A list of items defining the name and type of the produced measurement results.
result_format[].name	The name of a variable created as part of a measurement result, which can then be used in the Javascript limited Boolean expression below.
result_format[].type	The type of a variable created as part of a measurement result, which can then be used in the Javascript limited Boolean expression below. The type can be "number", "boolean" or "string".
assertion	A Javascript Limited Boolean expression, which expresses an objective in terms of constrains on a result. See 3.6 for details.

3.5 DEFINING AUTOMATED OBJECTIVES

As we saw, assisted objectives are described with a very simple data structure and are ultimately defined with a plain text human description. On the other hand, automated objectives require a greater level of machine-readable information in order to be exploited by tools such as Clouditor or CTPD. This section takes a deeper look into the structure of automated objectives and in particular the expression of "conditions" in these objectives, which relies on a subset of JavaScript.

One of the central elements of Automated Objectives is the "assertion" property, which is used to describe an objective as a Javascript Expression, which we call "Javascript Limited Boolean Expression" (JSLBE). These conditions are used as follows:

1) An auditing tool analyses evidence and produces a measurement result.
2) This result is assigned to a variable (or a set of variables), which are arrays named and typed according to the **result_format** property of the automated objective.
3) The assertion property of the automated objective will be evaluated: each time the assertion contains a variable name, it is substituted with the corresponding value assigned in step 2).
4) The evaluation of the assertion produces a result that is ultimately converted to a Boolean (true or false), which defines whether the objective is satisfied or not.

Example:

Consider a service that defines a "MonthlyUptime" attribute and promises that this value will be above 99.95% (a typical SLO).

The corresponding automated assessment object would typically contain the following properties:

```
{
   ...
   "result_format": [
        {
            "name": "uptime",
            "type": "number"
        }
    ],
    "assertion": "uptime[0]>=99.95"
}
```

Now if we assume that the monitoring tools collect evidence and measure a monthly availability of 99.978%, this value will be used to populate a single-value array named “uptime”, which could be defined as follows in pseudo-code:

```
uptime[0] = 99.978
```

The condition "uptime[0]>=99.95" can now be evaluated and since “uptime[0]” is equal to 99.978, the assertion is true: the objective is verified.

The use of a Javascript subset provides a lot of flexibility and allows more complex objectives. For example, we could imagine defining “BusinessHourUptime” and “NonBusinessHourUptime” attributes and write conditions such as:

```
"business_hours_uptime[0]>=99.95 &&
non_business_hours_uptime[0]>=95.0"
```

More examples are provided in section 3.6.

3.6 JAVASCRIPT LIMITED BOOLEAN EXPRESSIONS

A large part of this section is adapted from the CTP Data Model and API [CTP] with minor changes.

Conditions used objectives and triggers are described with Java Script Limited Boolean Expressions (JSLBE): a language that is modeled after JavaScript expressions, with many simplifications that are designed to facilitate implementation. We highlight in particular that JSLBE is limited to statements representing expressions and does not include any other language construct (such as assignments, control structures, declarations, prototypes, etc.). One of the key benefits of JSLBE is to offer language semantics that are familiar to the tech industry. We note however that the simple predicate logic offered by JSLBE cannot directly express temporal logic, to the contrary of more complex languages described in relevant research (e.g. see 2.2.1). In practice, this does not constitute a real disadvantage since temporal semantics can be contained in measurement results as shown in the example distinguishing “business hours” form “non-business hours” at the end of section 3.5.

JSLBE expressions operate on the table variables defined as part of a measurement result (see 3.4.3). In addition to these tables, a variable "updateTime" which is set to the UTC time when the measurement result was created or updated, can be used in JSLBE expressions.

The following are further examples of JSLBE Expressions that could be provided in objectives.

```
uptime[0] >= 99.0

(timeUTC("now")-toTimeUTC(updateTime)<3600)

matchRegex("UK", select(result, "country"))
```

3.6.1 BASIC GRAMMAR

The following EBNF grammar describes JSLBE, with some further details provided in the following sub-sections. A JSLBE expression is defined by the **Expr** token in the grammar below. A JSLBE expression **Expr** is considered true if `toBoolean(Expr)` returns true, and false otherwise, taking the definition of toBoolean described in 3.6.4.

```
Expr =
    Literal
    | Identifier
    | Expr "&&" Expr
    | Expr "||" Expr
    | Expr "<" Expr
    | Expr "<=" Expr
    | Expr ">" Expr
    | Expr ">=" Expr
    | Expr "!=" Expr
    | Expr "==" Expr
    | Expr "+" Expr
    | Expr "-" Expr
    | Expr "/" Expr
    | Expr "*" Expr
    | Expr "%" Expr
    | "-" Expr
    | "!" Expr
    | "(" Expr ")"
    | FunctionCall
    ;

Literal =
    StringLiteral
    | NumericLiteral
    | ArrayLiteral
    | ObjectLiteral
    | "true"
    | "false"
    | "null"
    ;

Identifier =
    IdentifierName
    | Identifier "[" Expr "]"
    | Identifier "." IdentifierName
    ;

FunctionCall =
    Identifier "(" ")"
    | Identifier "(" ParameterList ")"
    ;

ParameterList =
    Expr
    | ParameterList "," Expr
    ;

ObjectLiteral =
    "{" "}"
    | "{" ObjectElements "}"
    ;

ArrayLiteral =
    "[" "]"
    | "[" ArrayElements "]"
    ;

ArrayElements =
    Expr
    | ArrayElements "," Expr
    ;

ObjectElements =
    ObjectElementItem
```

```
    | ObjectElements, ObjectElementItem
    ;

ObjectElementItem =
    IdentifierName ":" Expr
    | StringLiteral ":" Expr
    ;
```

3.6.2 DEFINITIONS BORROWED FROM OTHER STANDARDS

The JSLBE grammar above borrow a few definitions from existing freely available standards, in particular [ECMA_262] version 5 (i.e. JavaScript).

`StringLiteral`

A UTF8 string quoted with simple or double quotes as defined in ECMA 262 v5, in clause 7.8.4.

Note that JSLBE represents strings in UTF-8 format, not UTF-16 as in JavaScript.

`NumericLiteral`

A double precision 64-bit format IEEE 754 floating point as defined in ECMA 262 v5, in clause 7.8.3.

`IdentifierName`

An **IdentifierName** as defined in ECMA 262 v5, in clause 7.6.

3.6.3 TYPES

THE 7 TYPES OF JSLBE

There are 7 types defined in JSLBE:

- **string**: A UTF8 character string.
- **number**: A double precision 64-bit format IEEE 754 floating point number, which is used also to represent integers.
- **boolean**: A type that represents either true or false.
- **null**: A type that represents a null or undefined value.

- **objects**: A type that represents a list of fields as (key, value) pairs (a simplified Javascript object).
- **arrays**: A type that represents a list of values indexed by a integral number (a simplified Javascript array). JSLBE arrays have one property *length*, which represents the highest existing index in the array, plus one.
- **function**: A type that represents a function.

We call **integral number** the subset of numbers that can be represented as an integer in the range -2^{31} and $2^{31}-1$, inclusive. This type does not explicitly exist in JSLBE but is useful to define some elements in this specification.

Note that JSLBE does not provide the **undefined** type that exists in javascript.

THE TYPEOF() PSEUDO-FUNCTION

For any value **a** represented in JSLBE, we define a pseudo-function `typeOf(a)` which returns a string representing the type of **a**(e.g. "string", "number", "boolean", "null", "object", "array" or "function"). The implementation of the pseudo-function `typeOf()` is not required: it is only defined to facilitate the description of this specification.

THE GETFIELD() PSEUDO-FUNCTION.

For the purpose of this specification, we define the pseudo-function `getField(a, prop)` which takes as first argument an array or object **a**, and as second argument a string or an integral number **prop**, and returns the value associated with the key **prop** in the array or object if it exists or **null** otherwise.

The implementation of the pseudo-function `getField()` is not required: it is only defined to facilitate the description of this specification.

3.6.4 FUNCTIONS

JSLBE only defines 8 functions or methods, which are detailed hereafter.

Note that all Arrays also define the special property length as specified in "The 7 types of JSLBE" (see 3.6.3).

toString(a): string

The function `toString` converts its argument **a** to a string as follows:

1. If **a** is a string return **a**
2. If **a** is a number return a string representation of a as would be returned by the "%e" specifier in the C standard function `sprintf()` (as defined in ISO C99).
3. If **a** is a Boolean return "true" if **a** is true, "false" otherwise.
4. If **a** is null or an empty array, return an empty string.
5. If **a** is an object, return the string "[Object Undefined]".
6. If **a** is an (non-empty) array:
    1. Let r be initialized with the value of `toString(GetField(a,0))`
    2. For i = 1 to GetField(a,"length")-1 do
        1. Concatenate the string "," to r
        2. Concatenate the string toString(GetField(a,i)) to r
        3. Return the string r.
7. If **a** is a function return the string "`function <id>() { [Native code] }`", where **<id>** is replaced by the name of the function.

toBoolean(a): Boolean

The function `toBoolean` converts its argument **a** to a Boolean as follows:

1. If **a** is a string return false if **a** is the empty string, and true otherwise.
2. If **a** is a number return false if **a** is -0, +0, or NaN, and true otherwise.
3. If **a** is a Boolean return **a**.
4. If **a** is null, return false.
5. If **a** is an object or an array return true.
6. If **a** is a function return true.

TONUMBER(A): NUMBER

The function `toNumber` converts its argument **a** to a number as follows:

1. If **a** is a string return the number that would be returned by applying the standard C function atof() to the string **a** (as defined in ISO C99).
2. If **a** is a number return **a**.
3. If **a** is a Boolean return 1 if **a** is true, 0 otherwise.
4. If **a** is null, return 0.
5. If **a** is an object or an array return NaN.
6. If **a** is a function return NaN.

ARRAY.MIN(): (ANY TYPE)

The method `min` returns the smallest value in an Array object as follows:

1. Let **t** be the array to which we apply the min method.
2. If **t** is empty return null.
3. Let **t[i]** define any value that verifies **t[i] <= t[j]** for any **i ≠ j** (using the comparison operator **<=** defined in 3.6.6.
4. If there is only one index **i** that verifies the property in step 3, return **t[i].**
5. If there are several indices **i** that verify the property in step 3, return the value **t[i]** with the smallest index **i** that verifies step 3.

ARRAY.MAX(): (ANY TYPE)

The method max returns the largest value in an Array object as follows:

1. Let **t** be the array to which we apply the max method.
2. If **t** is empty return null.
3. Let **t[i]** define any value that verifies **t[i]** >= **t[j]** for any **i ≠ j** (using the comparison operator **>=** defined in 3.6.6.

4. If there is only one index **i** that verifies the property in step 3, return **t[i]**.
5. If there are several indices **i** that verify the property in step 3, return the value **t[i]** with the biggest index **i** that verifies step 3.

MATCHREGEXP(R, V): BOOLEAN

The function `matchRegexp` verifies that the value **v** matches the regular expression **r**, which must be a string representing a POSIX Extended Regular Expression[10] (POSIX ERE). The value **v** can either be a string or an array of strings. The function works as follows:

1. If typeOf(r) is not "string" then abort raising an exception.
2. If **r** is not a correctly formed POSIX ERE, then abort raising an exception.
3. If typeOf(a) is not "array" then goto step 5.
4. If all integral number keys **i** of **a** verify matchRegexp(r, GetField(a, i)) then return true, otherwise return false.
5. If typeOf(a) is not "string" then abort raising a/n exception.
6. If **s** matches **r** according to POSIX ERE matching rules then return true, otherwise return false.

SELECT(S, A): TABLE

The function select iterates through an array **a** of objects and returns an array that is created by selecting the key **s** in each one of the objects of **a**. The function operates as follows:

1. If typeOf(a) is not "array" then abort raising an exception.
2. Let **t** be an array. For each index **i** representing an integral number key in **a**, define **t[i]** as getField(getField(a, i), s).
3. Return the array **t**.

[10] http://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html

TIMEUTC(S): NUMBER

The function timeUTC takes a string s and converts it to a number representing the number of seconds since 00:00:00 UTC, January 1, 1970, which is referred as the "epoch".

The string **s** must be either:

- An Internet Date/Time format as defined in RFC 3339 (e.g. "2015-07-20T12:34:56Z")
- The string "now".

The function operates as follows:

1. If typeOf(s) is not a string then abort raising an exception.
2. If **s** is the string "now", return the current UTC time expressed as the number of seconds elapsed since the "epoch".
3. If **s** is an RFC 3339 date/time string representing a UTC date before the "epoch", return the corresponding UTC time expressed as a negative number of seconds before the "epoch".
4. If **s** is an RFC 3339 date/time string representing a UTC date equal or greater than the "epoch", return the corresponding UTC time expressed as a number of seconds elapsed since the "epoch".
5. In all other cases, abort raising an exception.

3.6.5 BOOLEAN OPERATORS ||, &&

The expression a || b is evaluated as follows:

- If toBoolean(a) = true then return **a** else return **b**.

The expression a && b is evaluated as follows:

- If toBoolean(a) = false then return **a** else return **b**.

> **Note**: this approach is similar to the one used in Javascript (ECMAScript) but is different from more traditional languages such as C or Java.

3.6.6 COMPARISON OPERATORS <, <=, >, >=, ==, !=

Let compareString(a,b) be the pseudo-function that compares two UTF-8 strings in lexicographic order and returns (-1) if **a**<**b**, (1) if **a**>**b** and (0) if **a** and **b** are equal. Comparison is done on the basis of the UTF code unit values (not bytes).

The expression **a < b** is evaluated as follows:

1. If typeOf(a) is "string" and typeOf(b) is "string" return true if compareString(a,b) equals -1
2. Let **an** = toNumber(a) and **bn**= toNumber(b)
3. If **an** is **NaN** or **bn** is **NaN** return false.
4. Taking into account the canonical ordering of numbers, return true if **an < bn**, otherwise return false.

The expression **a == b** is evaluated as follows:

1. If typeOf(a) is "string" and typeOf(b) is "string" return true if compareString(a,b) equals 0
2. Let **an** = toNumber(a) and **bn**= toNumber(b)
3. If **an** is **NaN** or **bn** is **NaN** return false.
4. Taking into account the canonical ordering of numbers, return true if **an == bn**, otherwise return false.

The expression **a <= b** is evaluated as follows:

1. Return the result of evaluating (a<b || a==b)

The expression **a >= b** is evaluated as follows:

1. Return the result of evaluating (a>b || a==b)

The expression **a != b** is evaluated as follows:

1. Let **r** be the evaluation of a == b
2. If **r** is true return false, otherwise return true.

The expression **a > b** is evaluated as follows:

1. Let **r** be the evaluation of a < b
2. If **r** is true return false, otherwise return true.

*3.6.7 MULTIPLICATIVE AND ADDITIVE BINARY OPERATORS: +, -, *, /, %*

The expression **a + b** is evaluated as follows:

1. If typeOf(a) is "string" and typeOf(b) is "string", return the string formed by appending the string **b** to the end of **a**.
2. If typeOf(a) is number and typeOf(b) is "number", return the numerical sum **a**+**b**, according to IEEE 754.
3. In all other cases, return NaN.

The expression **a - b** is evaluated as follows:

1. If typeOf(a) is "number" and typeOf(b) is "number", return the numerical subtraction **a**-**b**, according to IEEE 754.
2. In all other cases, return NaN.

The expression **a * b** is evaluated as follows:

1. If typeOf(a) is "number" and typeOf(b) is "number", return the numerical multiplication **a** * **b**, according to IEEE 754.
2. In all other cases, return NaN.

The expression **a / b** is evaluated as follows:

1. If typeOf(a) is "number" and typeOf(b) is "number", return the numerical division **a**/**b**, according to IEEE 754.
2. In all other cases, return NaN.

> **Note:** Divisions by 0 return –Inf, +Inf, or NaN depending on the case, as specified in IEEE 754.

The expression **a % b** is evaluated as follows:

1. If typeOf(a) is "number" and typeOf(b) is "number", return a result equivalent to the computation of fmod(a,b) where fmod is a standard C function, which computes the floating-point remainder of dividing a by b (as defined in ISO C99).

2. In all other cases, return NaN.

3.6.8 IDENTIFIERS

The JSLBE defines Array identifiers corresponding to the measurement results defined in the objective (see 3.4.3). For each "result format" defined in an automated objective, an Array is created where:

- The name of the array is the value of the **name** property of the "result format".
- The elements of the array are of the type specified in the **type** property of the "result format".

In addition to these array identifiers, JSLBE defines the `updateTime` identifier and sets it to a string representing the UTC time when the measurement results was created or updated. The value of updateTime can be used in JSLBE expression to evaluate the date of the last measurement of a security attribute, notably in combination with the JSLBE function timeUTC.

To illustrate the link between the result format property of an automated objective and the identifiers that can be used in a JSLBE expression, consider the following measurement result definition:

```
...
"result_format": [
     {
         "name": "country",
         "type": "string"
     },
     {
         "name": "percentage",
         "type": "number"
     }
 ]
 ...
```

The JSLBE will recognize the following variables:

- **country,** an Array of strings.
- **`percentage`**, an Array of numbers.
- **`updateTime`**, a string representing UTC time.

3.6.9 EVALUATING JSLBE EXPRESSIONS

The result of the evaluation of a JSLBE expression is always one of the three following values: false, true or error.

The evaluation MUST generate the value error if any of the following condition occurs:

- The JSLBE contains a syntax error.
- The JSLBE calls a function that does not exist or attempts to access a field in a null variable.
- A function used in the expression generates an exception.

In all other cases, the function `toBoolean`() is applied to the result providing a final value that is equal to either **true** or **false**.

4 DATA FLOWS AND APIS

This section solely focuses on data flows and APIs related to the exchange of "certification objectives" between different entities in the EU-SEC framework. For a broader view of the toolset we refer the reader to deliverable D3.3.

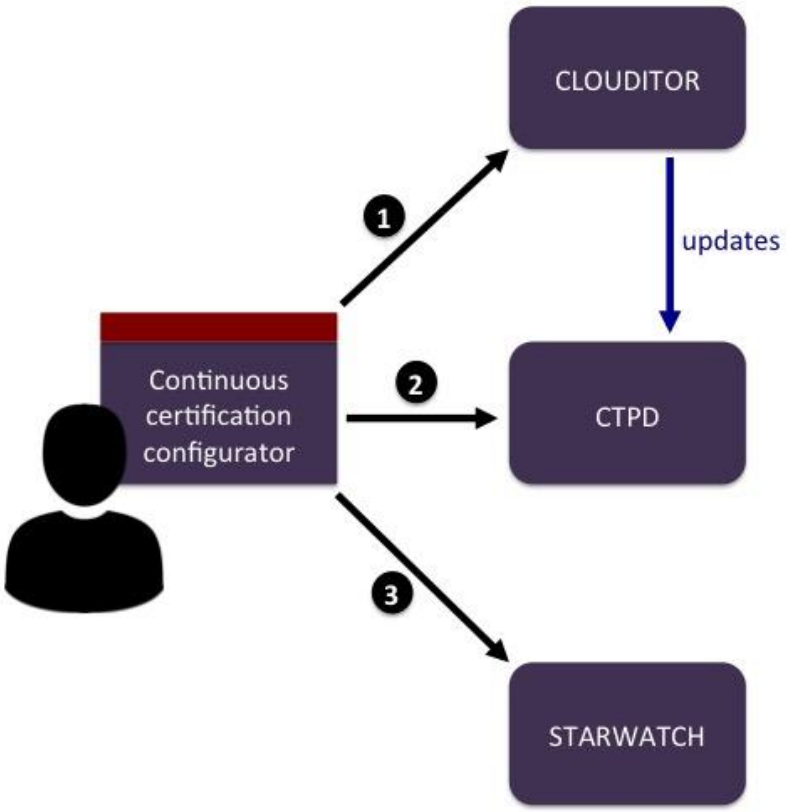

Figure 5: Continuous Certification Configuration

4.1 OVERVIEW

The above illustration shows the general use of the "certification objective" JSON data object defined in this document. An auditor will use a "continuous certification configurator" tool to create a **certification objective**, which will be then used to configure different tools in the EU-SEC continuous auditing toolchain that will be used to perform continuous certification, addressing both controls that can be automatically evaluated and controls that require human assessment (see [D3.3] for details). As illustrated by the three arrows in the illustration, the output of that tool — a JSON object — is then used as:

1) Input to Clouditor, to configure the automated audit tests that will be executed on the target cloud service. Clouditor will produce measurement results as the output of these tests.

2) As input to CTPD, to configure the automated objectives that CTPD will evaluate. CTPD will receive inputs from Clouditor and compare them with the defined objectives in the "certification objective".

3) As input to STARwatch, to configure the assisted objectives that will be monitored. STARwatch will check that the auditor further provides updates on the status of certain controls in timely manner in line with the frequencies specified in the "certification objective".

4.2 CONTINUOUS CERTIFICATION CONFIGURATOR

In its simplest form, the continuous certification configurator is simply a text editor that allows an auditor to edit a JSON file representing a "certification objective" data structure, as defined in 3.4.

A more advanced version would provide an UI that would enable the auditor to define the different components of a "certification objective" in a more visual way, taking advantage of the mostly hierarchical structure of the data structure. Such a tool would naturally provide consistency and validity checks on the produced data structure. With the availability of adequate APIs, this UI would also enable the auditor to send the produced "certification objective" to the tools defined in the auditing toolset (Clouditor, CTPD and perhaps STARwatch).

In upcoming EU-SEC pilot, we will start with the simplest approach and add a UI once the approach is qualified, taking into account feedback from early toolchain tests.

4.3 INTERFACING WITH CTPD

In the context of this project, CTPD will be extended with an API method that allows uploading a "certification objective", which will be then used by CTPD as a self-configuration mechanism. CTPD already provides means to configure a hierarchy of resources representing the audit of a cloud service, but this requires each resource in that hierarchy to be created separately, with a distinct API call to CTPD. The new proposed RESTful API methods will allow this configuration to take place at once, with one operation (POST).

This API extension will added to the next release of "The CTP prototype back office API.

4.3.1 CONVENTIONS USED IN THE API

The notations used in the following sections are the same as those adopted in the CTP API[11]. In particular, resource identifiers are noted as **`{id}`** in URL paths (e.g. the path /views/123 maps to **`/view/{id}`** where **`{id}`** equals to 123).

In CTPD, a **certification objective** will be called a "configuration" since it essentially allows configuring CTPD in the context of continuous auditing.

To remain consistent with the CTP API, the certification objective JSON data format described in section 3.4 is extended with two extra properties that will be shown in responses provided by the CTPD server:

1. "**self**": a URL that self-references the CTP resource being accessed. A GET query to this URL refreshes the client's representation of the CTP API resource. In a response to a POST request, this URL represents the newly created CTP API resource URL.
2. "**scope**": a URL pointing to the resource that is hierarchically above the current current. In the case of a **certification objective**, this is the URL of the CTP service view that represents the information system being audited.

As a consequence, the JSON data structure described in 3.4 will have the following properties in CTPD responses:

```
{
    "self": <url>,
    "scope": <url>,
    "certification_objective_id": <string>,
    "start_date": <datetime>,
    "end_date": <datetime>,
    "subject": {
        "organisation": <string>,
        "service": <string>,
        "scope": <string>
    },
...
```

Note that these extensions only apply to responses: data submitted to the CTPD server follows the format defined in 3.4.

[11] http://htmlpreview.github.io/?https://github.com/cloudsecurityalliance/ctpd/blob/master/client/CTP-Data-Model-And-API.html

4.3.2 UPLOADING A CERTIFICATION OBJECTIVE

```
POST /views/{id}/configurations
```

Upload a certification objective as part of the CTP service-view identified by {id}.

Method characteristics:

- **Request body**: A JSON object representing a certification objective as described in 3.4.
- **Query string**: none
- **Response body**: A configuration resource (a JSON certification objective), with the added "self" and "scope" properties.
- **Success status code**: 201

If provided, the properties self scope are ignored in the request body and are replaced by values constructed by the ctpd server.

Request example

```
POST /serviceViews/VYkcvsy-5zuVAAAD/configuration
```

The request body is a configuration objective.

```
{
    "certification_objective_id": "012930",
    "start_date": "2017-12-21T15:13:31Z",
    "end_date": "2018-12-21T15:13:30Z",
    "subject": {
        "organisation": "Cloud Security Alliance",
        "service": "STARwatch",
        "scope": "...
```

Response example

The response body is a fully defined asset resource, which includes a self-property with the URL of the newly created object.

```
{
    "self": "https://example.com/ctpd/configuration/akdcvsy45zuV7770",
    "scope": "https://example.com/ctpd/serviceViews/VYkcvsy-5zuVAAAD",
    "certification_objective_id": "012930",
    "start_date": "2017-12-21T15:13:31Z",
    "end_date": "2018-12-21T15:13:30Z",
    "subject": {
        "organisation": "Cloud Security Alliance",
        "service": "STARwatch",
        "scope": "...
```

4.3.3 GETTING INFORMATION ABOUT A CERTIFICATION OBJECTIVE

```
GET /configurations/{id}
```

Description

For any configuration (certification objective) identified by a {id}, A GET request to /confirguations/{id} provides information about the configuration (certification objective) identified by a {id}.

CTP servers SHALL only provide access to a configuration when it is part of a service-view that the CTP client is authorized to access.

Method characteristics:

- **Request query string:** None.
- **Request body:** None.
- **Response body:** Returns a JSON encoded certification objective with 2 additional properties ("self" and "scope") as defined in 4.3.1.
- **Success status code:** 200.

4.3.4 REMOVING A CERTIFICATION OBJECTIVE

```
DELETE /configurations/{id}
```

Deletes the configuration (certification objective) identified by {id}.

Method characteristics

- **Request body**: none
- **Query string**: none
- **Response body**: none
- **Success status code**: 204

Request example

```
DELETE /configurations/akdcvsy45zuV7770
```

There is no query body.

There is no response body.

4.4 INTERFACING WITH CLOUDITOR

4.4.1 BASIC COMMUNICATION BETWEEN CONTINUOUS CERTIFICATION CONFIGURATOR AND CLOUDITOR

The *continuous certification configurator* defines a certification objective which is supplied to the Clouditor using its RESTful API. The Clouditor then parses the certification objective and extracts the *metric* field as well as the *measurement parameters* field. Based on this information, the measurement setup (i.e., the configurations of required continuous tests) is derived and the corresponding continuous tests are launched. Upon successful deployment, the Clouditor notifies the *continuous certification configurator*. The measurement results which are produced by the Clouditor's continuous tests are forwarded to the CTPD, using its RESTful API.

In case an already deployed certification objective is altered or deleted, *continuous certification configurator* posts a changed certification objective to Clouditor or deletes it using its unique Id. Clouditor then takes the respective action that is stopping and starting continuous tests according to the modified certification objective or simply stopping any continuous test bound to a deleted certification objective.

4.4.2 RESTFULL API SPEC

The Clouditor follows the same API spec as the CTPD (see Section 4.3) which allows to configure continuous tests according to a certification objective. This means that, e.g., the *continuous certification configurator* supplies the certification objective using the same API call which is used to post the certification objective to the CTPD.

4.4.3 ADVANCED PROTOCOL

The above protocol description has to be understood as a starting point and will be improved/extended in the course of the pilot in WP5. One challenging question, for example, is how to handle certification objectives which can only partly be satisfied by Clouditor, that is, for which Clouditor cannot provide certain measurement results. Such incomplete certification objective satisfaction may occur during initialization as well as in case of changes to the certification objective. Among others, whether the Clouditor can provide all necessary measurement results depends on the privileges Clouditor has to perform continuous security audits of a concrete cloud service which is subject to continuous certification. This, in turn, depends on the selected integration strategy of Clouditor whose variants will be described as part of Deliverable 3.4. However, the communication protocol between the *continuous certification configurator* and Clouditor in case of incomplete certification objection satisfaction has to be extended accordingly.

4.5 INTERFACING WITH STARWATCH

As opposed to CTPD or Clouditor, STARwatch is designed to take care of security objectives that require human intervention for their assessment. As such, an operator will "manually" upload the certification objective produced by the continuous certification configurator, as detailed in the following paragraphs.

Currently, STARwatch is designed for assessments that are conducted on a yearly basis at best and works as follows:

1) The user completes or updates a CAIQ self-assessment (133 controls, 195 questions), using the STARwatch SaaS tool.

2) The user exports the finalized self-assessment in EXCEL format, or clicks on the "publish to the registry" button[12] in STARwatch.
3) The resulting self-assessment is vetted by CSA and published in the registry as a "self-assessment", becoming publicly visible.
4) When the user feels like publishing a new entry, he goes back to step 1.

To take into account continuous auditing, the previously described STARwatch workflow will be extended as follows:

1) The user completes or updates a CAIQ self-assessment (133 controls, 195 questions), using the STARwatch SaaS tool.
2) The user clicks on "continuous auditing" and selects a JSON file containing a "certification objective". This certification objective must contain references to the CSA CAIQ self-assessment as its main standard.
3) The resulting self-assessment is vetted by CSA and published in the registry as a "continuous self-assessment". The state of the assessment becomes public in the registry.
4) The user is periodically notified by STARwatch to confirm that certain controls are in place, in accordance with the frequencies defined in the "certification objective".
    a. If the user fails to update relevant responses in a timely manner, STARwatch marks the continuous self-assessment as "suspended" in the publicly visible registry.
    b. If the user updates update relevant responses in a timely manner, STARwatch marks the self-assessment as "active" in the publicly visible registry.
5) The process halts when the user requests it or when the continuous self-assessment has expired, according to the duration defined in the certification objective.

[12] The "publish to the registry" button is a feature in currently being implemented and will be in the production version of STARwatch in 2018.

5 CONCLUSION

This deliverable defined a universal data format that can be used to configure tools in the EU-SEC tool chain for the purpose of continuous auditing, taking into account both (1) security objectives that can be automatically assessed and (2) security objectives which require human evaluation.

Many elements in this deliverable are new and will require further validation during the pilot phase. Some elements specified in this deliverable will likely be modified in this process. We will use the opportunity provided by the final deliverable in Work Package 3 to update this specification where necessary.

REFERENCES

[CAIQ] Cloud Security Alliance, Consensus Assessments Initiative Questionnaire. Online: https://cloudsecurityalliance.org/group/consensus-assessments/#_overview

[CTP] Cloud Security Alliance, CTP Data Model and API, rev. 2.14. November 2015. Online: http://htmlpreview.github.io/?https://github.com/cloudsecurityalliance/ctpd/blob/master/client/CTP-Data-Model-And-API.html

[CUMULUS] The CUMULUS project. Seventh Framework Program. Online: http://www.cumulus-project.eu/

[CUMULUS-D2.1] Alain Pannetrat, Giles Hogben, Spyros Katopodis, George Spanoudakis, Carlos Sánchez Cazorla, "D2.1 - Security-aware SLA specification language and cloud security dependency model". CUMULUS project. September 2013.

[CUMULUS-D2.3] Ernesto Damiani, George Spanoudakis, Spyros Katopodis, Khaled Mahbub, Maria Krotsiani, Stelvio Cimato, Marco Anisetti), Claudio Ardagna, Francesco Zavatarelli, Maria Rosa Vieira Alvarez, Renato Menicocci, Alessandro Riccardi, Vittorio Bagini, Javier Espinar, Antonio Muñoz, Antonio Maña, Hristo Koshutanski, "D2.3 - Certification models v.2". CUMULUS project. May 2014.

[D1.4] EU-SEC

[D3.3] EU-SEC

[ECMA_262] ECMA International. *ECMAScript Language Specification, 5.1 edition*. June 2011. Available at http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[PLA] Cloud Security Alliance, Privacy Level Agreement Version 3. Online: https://gdpr.cloudsecurityalliance.org/

[SLA@SOI] Empowering the Service Economy with SLA-aware Infrastructures. Seventh Framework Program. Online: http://cordis.europa.eu/project/rcn/87600_en.html

[SPECS] Secure Provisioning of Cloud Services Based on SLA Management. Seventh Framework Program. Online: http://www.specs-project.eu/

[RFC_3339] G. Klyne, C. Newman, *RFC 3339: Date and Time on the Internet: Timestamps* Internet Engineering Task Force (IETF), July 2002.