RO-Verifier:

The ByStar Remote-Operations Invocations And Verifications Framework

Tools And Strategies For Generalized OpenAPI/Swagger Based Verification Of Web-Services

Mohsen BANAN
Email: http://mohsen.1.banan.byname.net/contact

PLPC-180057

Version 0.4
February 12, 2019

Available on-line at:
http://www.by-star.net/PLPC/180057

Contents

List of Figures

Part I

Summary And Overview

1 Summary

Web-Services, REST-APIs, microservices and services-oriented architecture (SOA) have become the dominant model for exposing Remote-Operations through https as web services.

The formal specification of web services have now converged on OpenAPI Specification (formerly Swagger Specification) format. Many web services now publish their interfaces as formal swagger specifications. Availability of the service definitions in a standardized form allows for development of strategies and tools for both general validation and security oriented validation of web-services.

We are proposing that a specific invocation and verification software package, called:

> **RO-Verifier: ByStar Remote-Operations Invocations And Verifications Framework**

be used for both general verification and security oriented verification of existing and new web services. The RO-Verifier is open-source and is subject to Affero General Public License (GPL).

The purpose and scope of this document is two fold:

1) A Strategy For Service Specification Centered Verification Of Web Services: This document outlines a specific strategy that can be adopted by any organization to methodically verify its web services comprehensively and also with a focus on security in a generalized fashion.

2) The RO-Verifier Framework: This document provides the needed information for obtaining, installing and using RO-Verifier and related software packages.

Scope and span of these web-services verification tools is the entirety of operations specified in the OpenAPI/Swagger file. In that broader context, these tools can be used by service developers for complete regression testing.

We consider our approach as "Generalized" because it only assumes the following from the service provider.

1. The Complete OpenAPI/Swagger Specification.
2. The Service Access Point. Provided either as a url or as service discovery methods.
3. Necessary credentials for accessing the service and for invoking operations.

In a proper digital ecosystem, these can be assumed to be available for all web services and as such the proposed tools and framework are applicable to all these web services.

Remote-operations/web-services are by nature network exposed. The presence of many web services exposes a large attack surface. Therefore, only a generalized validation approach which is applicable to all OpenAPI/Swagger web services is viable. Furthermore, the proposed generalized verification framework is inherently batch oriented and repeatable. So, the validation process conforms to the regression testing model and can be used re-verify the service as needed.

Since these tools view the performer as a blackbox and engage in invocation verification of the service based on its contract (the OpenAPI/Swagger specification), they are equally applicable to:

- in-house built web-services.
- public open-source web-services
- private (proprietary) closed-source web-services

For in-house developed and maintained web services, the RO-Verifier Framework can be deployed abinitio and used continuously as a quality assurance measure.

The RO-Verifier framework is based on ingesting the OpenAPI/Swagger specification and then mapping it to a set of corresponding functions and data structures in Python. The ingestion and mapping is performed by an open-source software package called: Bravado. Bravado creates its Python library dynamically without a code generator. RO-Verifier is a layer on top of Bravado where the Remote-Operations invocation abstractions are used to allow users to create scenarios for invocations and corresponding verifications. These customized scenarios can then be used to subject the perforomer (the web service) to regression testing and automated repeatable verification. RO-Verifier framework and the developed scenarios can be integrated with the web service development process to improve rfeliability and security.

There are three areas of security oriented blackbox verifications to which these tools apply:

1. Identify and test common security vulnerabilities that are manifested in the OpenAPI/Swagger specification
2. Validation of credentials, access control and correlations to IAM
3. Brute force attacks, that evaluate the services response to denial of service type attacks

In that context, in a sense our approach can be considered a form of penetration testing for web services. However, this framework allows for such tests to be considered generalized and become consistently applicable to all web services, instead of isolated penetration testing of web services one at a time.

Based on this approach, the proposed framework and these tools, with discipline and methodically we can continuously validate the entire web services foundation of the applications fabric of a digital ecosystem. The benefit is a more robust and more secure applications environment.

In Part II, we provide some relevant information about OpenAPI/Swagger specifications and invoker and performer realms of web services.

In Part III, we describe the problem space and the aspects of the problem that we wish to address in this document.

In Part IV, we describe the contours of our proposed solution and identify some benefits.

In Part V, we describe the proposed tools in broad terms and provide instructions for installation of the software and example usages.

In Part VI, we describe some of the various uses of RO-Verifier.

In Part VII, we describe the interplay between core web service related facilities, their reflection in OpenAPI/Swagger specifications. We then emphasize Identity and Access Management (IAM).

In Part IX, we identify some risks associated with adopting this approach based on a 5 year horizon.

2 Context

2.1 ByStar: The Broader Context

This is part of a bigger picture. The origin of RO-Verifier is the ByStar digital ecosystem.

Many of the architecture principles presented in the ByStar context are equally applicable in any large enterprise's context.

2.1.1 Public Presentation Form Of This Topic

A presentation form of this subject matter has been publicly available in:

> **RO-Verifier: ByStar Remote-Operations Invocations And Verifications Framework**
> **Tools And Strategies For Generalized OpenAPI/Swagger Based Verification Of Web-Services**
> http://www.by-star.net/PLPC/180057 — [1]

2.1.2 About ICM (Interactive Command Modules)

Interactive Command Modules (ICM) is a framework for development of Python and Bash modules that conform to particular structures which permit their functions to map to command-line.

Based on the tools and strategies described in this document, ICMs can become Remote-Operation Interactive Command Modules (RO-ICM) which can then be deployed as Web Services.

Both, the ByStar Web-Services Development Framework and the ByStar Web-Services Validation Framework are based on the Interactive Command Modules (ICM) model which is documented in:

> **Unified Python Interactive Command Modules (ICM) and ICM-Players**
> **A Framework For Development Of Expectations-Complete Commands**
> **A Model For GUI-Line User Experience**
> http://www.by-star.net/PLPC/180050 — [3]

2.1.3 About BISOS (ByStar Internet Services OS)

ByStar Internet Services OS (BISOS) is a universal Services OS layered on top of Linux. All ByStar services and all ByStar usage environments (devices) use BISOS.

Based on the ICM framework, in the ByStar context, BISOS is documented in:

> **The Universal BISOS: ByStar Internet Services Operating System**
> **Model, Terminology, Implementation And Usage**
> **A Framework For Cohesive Creation, Deployment and Management Of Internet Services**
> http://www.by-star.net/PLPC/180047 — [2]

2.1.4 About ByStar

The strategies and tools being proposed in this document have their origins in the ByStar digital ecosystem and are practiced in ByStar.

ByStar is an autonomy and privacy oriented digital ecosystem. For more information about ByStar, see `http://www.by-star.net`

The universal BISOS is then used for realization of the ByStar digital ecosystem, which is documented in:

> **The Libre-Halaal ByStar Digital Ecosystem**
> **A Unified and Non-Proprietary Model For Autonomous Internet Services**
> http://www.by-star.net/PLPC/180016

A podcast providing an overview of ByStar is available at:

> **An Overview Of The Libre-Halaal ByStar Digital Ecosystem**
> **With Pointers For Digging Deeper**
> http://www.by-star.net/PLPC/180054 — [4]

2.1.5 About Mohsen

Mohsen Banan, is the primary author of this document, the origin of ByStar, the primary architect of BISOS and the primary implementer of the ICM Framework.

For more information about Mohsen, see `http://mohsen.1.banan.byname.net`

Part II

Background

3 Web Services: Technology Background

The basic concept and model of Remote Operations (RO), with some variations on emphasis has morphed into various terms and key words. Some of these include:

- Web Services – Where use of http/https as the underlying protocol for remote operations is emphasized and where use of JSON for Arguments and Results encoding is emphasized.
- micorservices – Where building on other services, API-Gateways, service-discovery and containerization is emphasized.
- serverless – Where on demand realization of the performer is emphasized.
- SOA (Service Oriented Architecture) - also called service-based architecture – Where the notion of services being provided to the other components by application components is emphasized.
- RESTful APIs (Representational State Transfer) – Where stateless operations are emphasized.
- SOAP (Simple Object Access Protocol) – Where neutrality of transport is emphasized and where use of XML for Arguments and Results encoding is emphasized.
- Protocol Buffers – Where encoding efficiency is emphasized.
- ESRO (Efficient Short Remote Operations) – Where protocol efficiency is emphasized and constrained systems and IoT are addressed.
- RPC (Remote Procedure Call) – Where the notion of inter-process communication is emphasized and where use of XDR for Arguments and Results encoding is emphasized.
- ROSE (Remote Operation Service Element) – Where the formality of the ISO Reference Model is emphasized and where use of ASN.1 for Arguments and Results encoding is emphasized.

All of these are really just variations and trends on the basic model and concept of Remote-Operations.

Each of these flavors come with their own model and lingo. The models of terminology of these flavors are not consistent, resulting into confusion.

This confusion then results into mis-use of the terminology which then morphs and degenerates the underlying concept. For example in common use, RESTful API have become synonymous to Web Services even when REST is not relevant.

Furthermore, the technology behind some of these terms may have become obsolete or may have fallen out of fashion. These terms are often used without precision and out of place leading to more confusion.

So, it is an understatement when we say the current model and terminology of Web-Services is messy.

To address this we put forward our own model and terminology.

4 Overview Of Model And Terminology Of Remote-Operations/Web-Services

Our model and terminology for Remote-Operations/Web-Services is based on:

> **X.880: Remote Operations: Model, Notation and Service Definition**

CCITT/ITU and ISO/IEC in X.219 [?] and updated in X.880 and X.881.

Throughout this document and in the RO-Verifier code We use ROSE's model and terminology of Operations (with some augmentations), Invokers and Performers – not the ad hoc clients and servers terminology of web services folklore.

The basic model and concept is depicted in Figure 1.

Where;

- The Remote Operations Invoker, invokes a Remote Operation based on the Service Specification at the Service-Access-Point and provides its Argument as a Parameter.
- The Operation is described by its Argument, Result and Error.
- The Remote Operations Performer, performs a Remote Operation based on the Service Specification at the Service-Access-Point and provides a Result or an Error.

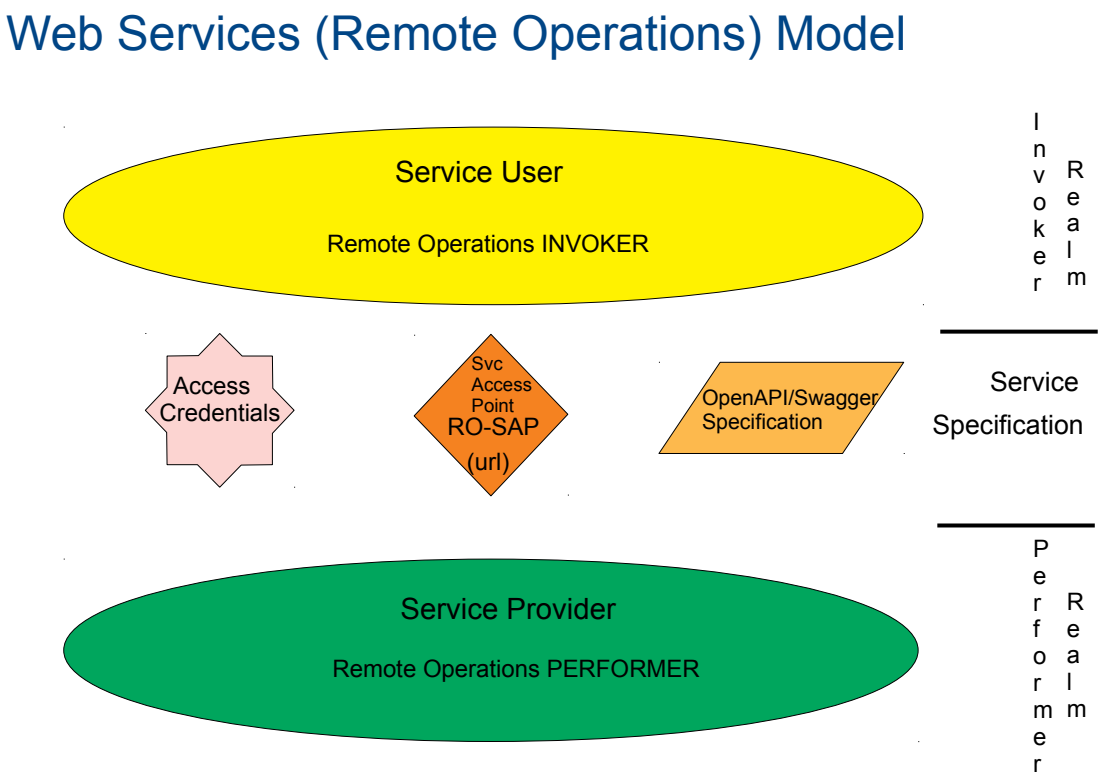

Figure 1: The Basic Model Of Web Services (Remote Operations)

This basic simple model is common across the above mentioned key words and terms.

In 2019, the industry has converged around the following practices for Remote Operations:

- Use of HTTP/HTTPS and its full set of verbs for the transport of synchronous remote operations.
- Use of JSON for encoding of Arguments, Results and Errors.
- Use of Swagger/OpenAPI for the formal specification of the Remote Operations Service.

- Use of RESTful model for making the service stateless and resilient
- Use of OAuth token for Authentication and Authorization in the context of IAM (Identity and Access Management)
- Use of containers (e.g, docker) and orchestration of containers (e.g., swarm, kubernetes) to allow for scalablity and resilience of performers.
- Use of front-end controller/proxies to load balance, rate limit and protect, in common, the providers.

Sometimes, various of these common and core needed features are bundled together as umbrella frameworks. Apigee is an example of such an umbrella framework. Existance of such containerization and front-endings do not impact our basic model.

With these specifities in place we now further refine our Web Service (Remote Operations) Model.

5 Web Services (Remote Operations) Model

As illustrated in Figure 1, our model consists of 3 realms.

1. The Performer Realm – server/services side
2. The Service Specification Realm – the API
3. The Invoker Realm – client/user/device side

Each of these realms have their own characteristics and environments. We expand on these below.

5.1 The Service Specification Realm

The services (operations) of the Performer are exposed to the Invoker in the service specification realm.

The service specification realm consists of three fundamental pieces.

- The Operations Specification – the Swagger/OpenAPI specification – The API
- The Remote Operations Service Access Point (RO-SAP) – the url at which
- Access Credentials

5.1.1 The Operations Specification – the Swagger/OpenAPI specification

OpenAPI Specification (formerly Swagger Specification) is an API description format for Remote Operations that the Performer supports. An OpenAPI specification allows you to describe your entire API, including:

- Available endpoints and operations on each endpoint
- Operation parameters Input and output for each operation
- Authentication methods
- Usage information and comments

OpenAPI/Swagger specification is an ad hoc standard. It is not a product of any formal standardization organization.

The OpenAPI/Swagger specification has facilitated the creation of a great deal of tools (code-generators, libraries, user-interfaces, editors, validators) and facilities. We provide an overview of some of these below.

5.1.1.1 Swagger Editor And Swagger Specification Validators

The Swagger Editor is an open source editor to design, define and document swagger specifications.

An instance of swagger editor is hosted at: https://editor.swagger.io/.

5.1.1.2 Swagger Code Generators

Swagger code generators that can create performer stubs and invoker interfaces exist for many languages and many frameworks.

The performer swagger code generators, generate a server stub for your API. The only thing left is to implement the server logic – and your API is ready to go live!

The invoker swagger code generators, generate client libraries for your API in over 40 languages.

See https://github.com/swagger-api/swagger-codegen#swagger-code-generator for more details.

5.1.1.3 Frameworks That Produce Swagger Specifications

Many rich web services frameworks support production and maintenance of swagger specifications. Java's Dropwizzard and SpringBoot are two such frameworks.

5.1.1.4 Swagger UI

The Swagger UI is an open source tool which allows you to visually render documentation for an API defined with the OpenAPI (Swagger) Specification. It allows you to generate interactive API documentation that lets your users try out the API calls directly in the browser.

An instance of the swagger ui is available online at: https://swagger.io/tools/swagger-ui/

5.1.1.5 The Pet Store Example

In order to provide a live example for swagger tools chain development an example application called the "Pet Store" has been created.

See https://petstore.swagger.io/ for more details.

5.1.1.6 Directory Of Publicly Available Swagger Specifications

A large collection of publicly available swagger specifications is collectively maintained.

See https://github.com/APIs-guru/openapi-directory for more details.

5.1.2 Remote Operations Service Access Point (RO-SAP)

The point at which the service is delivered.

Expand on service discovery.

5.1.3 Authentication Information And Access Credentials

The Operations Specification (OpenAPI/Swagger file) may include description of Security Requirement Object, Security Schemes and OAuth Flows Object which describe methods and formats.

To invoke operations, the invoker needs provide authentication information and access credentials that the performer expects.

5.2 Performer Realm

The performer is responsible for the realization of the services specified in the Service Specification Realm.

The performer needs to scale and be performant. There are one or few implementation of a performer typically in one or few programming-languages.

5.2.1 Three Ways Of Specification Of Swagger-file By The Performer

1. Design and write the Service Specification in full in one place – e.g., with swagger-editor. Then use code-generators for both performer and invoker.
2. Design and write the Service Specification in pieces along with implementation (a la Dropwizard, SpringBoot). Then publish the aggregated swagger for code-generation of invokers. Performer is framework driven – no generated code.
3. Design and write the Service Specification in pieces along with implementation for single process usage. Then generate the aggregated Service Specification for use with code-generators for both performer and invoker.

Method (3) is that of Web Services ICM.

5.3 Invoker Realm

The operations specified in the Service Specification Realm can be invoked by users.

The Invoker Realm includes many users of different capabilities and implemented in many programming-languages. Typically based on a the Service Specification, code-generators create code for libraries that client code can use. For dynamic languages such as Python, tools such as Bravado do the equivalent of code-generation on the fly such that in-effect the Service Specification is mapped to the target programming language.

6 ByStar Remote-Operations Services Terminology

The RO-Verifier software and framework use a formal model and terminology.

ITU X.880 and X.881 which are harmonized with ISO/IEC 13712-1, provide a model, terminology and service definitions for Remote Operations.

Such a valuable formal model and terminology is absent in the Web Services world and the OpenAIP/Swagger world.

The RO-Verifier software exposes web services capabilities in the Remote Operations model which conform to the ROSE terminology.

A summary of X.880 and X.881 terminology is included below. These reflect the Python interface that RO-Verifier provides.

6.1 ROS Model

Remote operations (ROS) is a paradigm for interactive communication between objects. Objects whose interactions are described and specified using ROS are **ROS-objects**. The basic interaction involved is the invocation of an operation by one ROS-object (the invoker) and its performance by another (the performer).

Completion of the performance of the operation (whether successfully or unsuccessfully) may lead to the return, by the performer to the invoker, of a report of its outcome.

A report of the successful completion of an operation is a **result**; a report of unsuccessful completion an **error**.

During the performance of an operation, the performer can invoke **linked operations**, intended to be performed by the invoker of the original operation.

For correct interworking, certain properties of the operation must be known by both invoker and performer. The properties include:

- whether reports are to be returned, and if so, which ones;
- the types of the values, if any, to be conveyed with invocations of the operation or returns from it;
- which operations, if any, can be linked to it;
- the code value to be used to distinguish this operation from the others that might be invoked.

6.2 ROS Concept Definitions

argument: A data value accompanying the invocation of an operation.

contract: A set of requirements on one or more objects prescribing a collective behaviour.

error: A report of the unsuccessful performance of an operation.

linked operation: An operation invoked during the performance of another operation by the (latter's) performer and intended to be performed by the (latter's) invoker.

object: A model of (possibly a self-contained part of) a system, characterized by its initial state and its behaviour arising from external interactions over well-defined interfaces.

operation: A function that one object (the invoker) can request of another (the performer).

parameter (of an error): A data value which may accompany the report of an error.

result: A data value which may accompany the report of the successful performance of an operation.

ROS-object: An object whose interactions with other objects are described using ROS concepts.

synchronous: A characteristic of an operation that, once invoked, its invoker cannot invoke another synchronous operation (with the same intended performer) until the outcome has been reported.

6.3 ROS Service Definitions

RO-INVOKE: The RO-INVOKE service enables an invoking AE to request an operation to be performed by the performing AE.

RO-RESULT: The RO-RESULT service enables the performing AE to return the positive reply of a successfully performed operation to the invoking AE.

RO-ERROR: The RO-ERROR service enables the performing AE to return the negative reply of an unsuccessfully performed operation to the invoking AE.

RO-REJECT-U: The RO-REJECT-U service enables one AE to reject the request or reply of the other AE if the ROSE-user has detected a problem.

RO-REJECT-P: The RO-REJECT-P service enables the ROSE-user to be informed about a problem detected by the ROSEprovider.

Argument: This parameter is the argument of the invoked operation.

6.4 Adding REST To ROSE

Our use of the X.219 (Remote Operation Service Element – ROSE) model, then needs to be extended to include the RESTFul model.

The basic concepts of ROSE (Remote Operations Services Element) can easily be augmented by the basic concept of REST (Representational State Transfer).

In the Web Services context we can go from Remote Operations to REST's object, method model by introducing the notion of "RO-Sap-endpoint" (Remote Operation Service Access Point Endpoint).

6.5 Mapping Of ByStar ROS Terminology To OpenAPI/Swagger Terminology

In this section we will be presenting a mapping between the informal terminology of Web Services and OpenAPI/Swagger to the formal terminology presented here.

Part III

Contours Of The Problem

The Remote-Operations/Web-Services paradigm provides many advantages over the direct monolithic applications model. These include:

- Large applications can be split into many loosely coupled components.
- Each component can then be scaled independently.
- Based on their Service Specification, each component can be developed, maintained and evolved independently.
- The smaller code base of each components may lead to higher quality and more optimization.
- The loose coupling between components enables containment of outages.

The Remote-Operations/Web-Services paradigm has some disadvantages that can lead to a number of problems. We expand on risks and vulnerabilities of Remote-Operations/Web-Services in the following sections.

6.6 Confusion: Absence Of Reference Model And Common Terminology

The model and terminology of Remote-Operations/Web-Services has diverged over the past 30 years into many branches. Many concepts and variations of concepts are discussed with redundant and ambiguous terminology.

It is an understatement when we say the current model and terminology of web-services is messy.

6.7 Inherent Risk: Remote-Operations/Web-Servies Expose A Large Attack Surface

Remote-Operations/Web-Services are by nature network exposed. The presence of many web services exposes a large attack surface.

6.8 Absence Of Formal Reflection Of IAM In Service Specifications Inhibits Effective Use Of Security Mechanisms

In practice, the existing OpenAPI/Swagger specification capabilities do not properly support reflections of authentication and authorization tokens and their association with specified operations.

6.9 Difficult To Monitor And Manage – Diverging Dynamics

Use of uncoordinated web-services frameworks lead to incoherence and insecure components.

6.10 Ever Evolving: Underlying Implementations Of Remote-Operations Change Continuously

A component may have to call the latest version of another component for some clients and call the previous version of the same component for another set of clients (i.e., version management). Running an integration test is more difficult since a test environment is needed wherein all components must be working and communicating with each other.

6.11 Many Points Of Failure

The presence of multiple components creates the availability problem since any component may cease functioning at any time.

Well known machineries exist to address this problem. For example, use of containers (e.g, docker) and orchestration of containers (e.g., swarm, kubernetes) allows for scalablity and resilience of performers. Addressing this problem is not a focus of this document.

Part IV

Contours Of A Solution And Benefits

7 Mapping Of Identified Problems To Proposed Solutions

In order to address the difficulties described in Section 6.10 , we propose a model of automated native penetration testing centered around the OpenAPI/Swagger specification.

7.1 Reference Model And Common Terminology

In order to address the difficulties described in Section 6.6 we have provided a starting point for a formal reference model and terminology as it applies to our specific approach and environment in Section II

7.2 Regression Oriented Verification Of Exposed Attack Surfaces

A human review of the service specification may point to potential security vulnerabilities for which we can easily develop scenarios that can exercise those areas. An example is common use of unbounded and unqualified strings in swagger specifications.

Verification of access control to each and every operation within the service specification can well be implemented using RO-Verifier.

7.3 Reflection Of IAM In Service Specifications

In order to address the difficulties described in Section 6.8 we have provided a starting point in Section 17

8 Benefits Of A Generalized OpenApi/Swagger Strategy For Web Services Validation

A generalized OpenApi/Swagger strategy for web services validation offers many benefits. Some of these we describe in the sections below.

8.1 Benefits And Advantages Of The Generalized Swagger Centered Invocation Model

The primary benefit of the proposed approach is enduring improvements in security and reliability of all OpenAPI/Swagger based web services.

The generalized validation framework of (invoke, invoke-verification and invoke-reporting) permits for disciplined complete external blackbox validation of web-services based on their service specifications.

8.2 Responsibilities For Methodic Validation Of Web Services And For Securing Their Attack Surface

Web services are the foundation of application fabrics in modern digital ecosystems.

From an architecture perspective, not addressing security of this foundation amounts to negligence.

Any approach to this problem which is not generalized and reproducible and repeatable is not scalable and is temporary (not enduring). Such approaches are incomplete.

Any generalized and reproducible approach to web services need to be centered around their service specifications.

So, the only remaining question is: “What service specification centered tools should be used to validate web-services?”

If not the proposed RO-Verifier, then which framework and which tools?

Part V

RO-Verifier: Generalized Web-Services Validation Tools

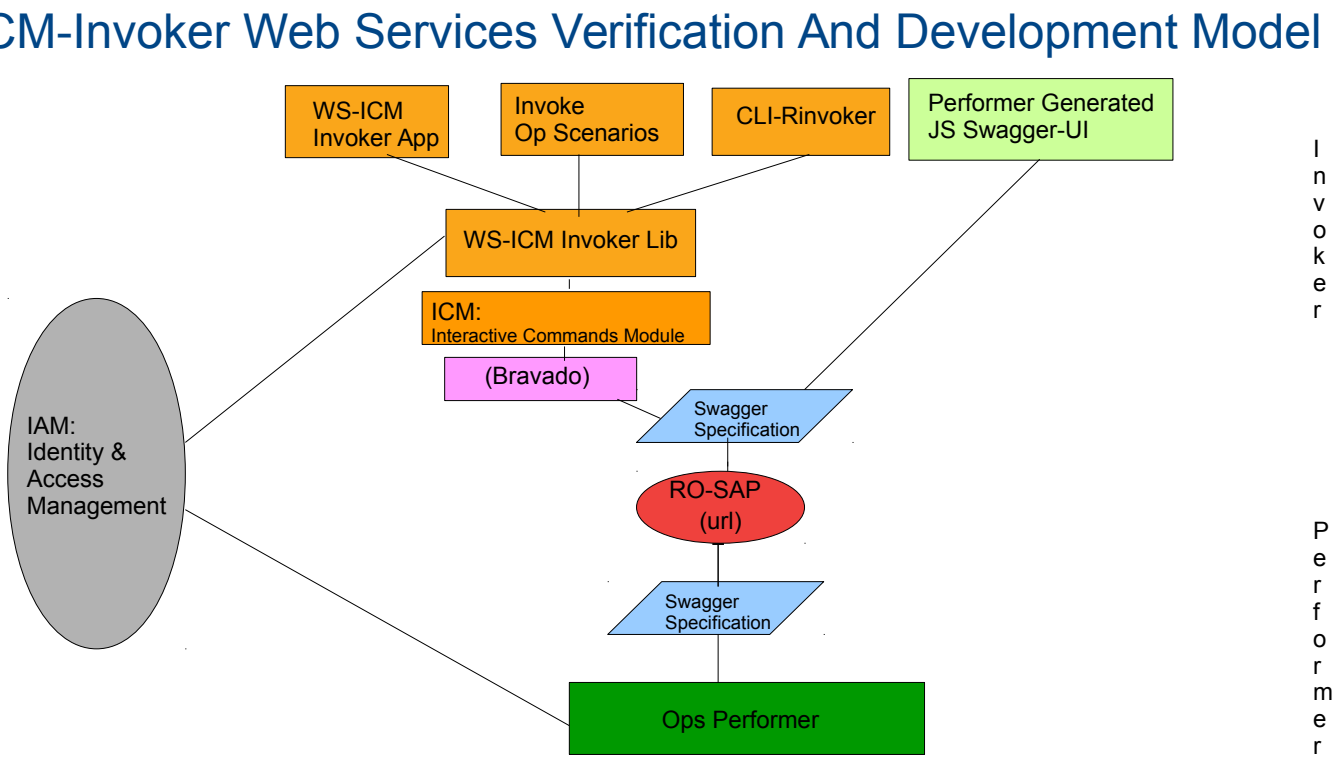

Figure 2: Swagger Based ICM Web Services Invoker Model

9 Overview Of The Web-Services Validation Tools

The Web-Services Validation Tools assume that a correct Swagger Specification is available. These tools do not focus on validating the service specification – there are a variety of tools available for validation of the swagger specifications, that is a different topic. These tools focus on validating the service based on its assumed correct specification.

These tools are invocation tools. They operate purely as the client side and other than verification and use of the service their scope does not extent to the server side (performers).

These tools are purely python based and they are purely Open Source (FOSS, Libre-Halaal). The python packages are available at PyPi and the complete source code is available at github.

You can think of these tools as a layer on top of
Bravado – `https://bravado.readthedocs.io/en/stable/`, `https://github.com/Yelp/bravado`.

Bravado ingests the swagger service specification in json or yaml and maps it to python on the fly. Bravado is a complete replacement to swagger codegen – the traditional code generation phase is eliminated. The Web-Services Validation Tools then create a higher level of convenience for invoking the operations specified in the service specification.

The Web-Services Validation Tools fall into two broad categories of:

1. Command Line Service Invocation Tools
2. Python Scenarios Service Invocation Tools

9.1 Command-Line Remote Invocations – rinvoker.py

Based on a given a swagger specification, rinvoker.py maps the json/yaml specification to python (using Bravado) and then python functions corresponding to remote-operation invocations are exposed as command-line using the ICM package (Interactive Command Modules).

All of this happens on the fly. Given a service-specification and a base service url, all operations become available for invocation at command line.

Pointers to examples and additional details are provided below.

9.2 Python Operations Scenarios – opScnSvc.py

In many situations the command-line interface may not be adequate for operations invocations as the parameters syntax may be complex, and as the results syntax may complex and as operation invocations sequences may be chained or interdependent. For such situation, a high level python interface called Operation-Scenarios (opScn) is provided.

You can then customize remote operation invocations as concrete scenarios (opScn) specifications.

Pointers to examples and additional details are provided below.

10 Installation Of The Web-Services Validation Tools

If you don't have Python 2.7 already installed, execute the following steps.

- Install Python 2.7
- Install pip and upgrade to the latest
- Optionally, if you are familiar with virtualenv, create a virtualenv.

With Python and pip in place, you can now install the unisos.mmwsIcm package.

- pip install unisos.mmwsIcm

All needed dependencies will be installed by just doing that.

10.1 Run The Web-Services Validation Tools Against The Canonical Petstore

Locate where the python scripts are installed. They would be in the bin directory of where Python 2.7 was installed. On Windows locate the default bin directory by running: "where rinvoker.py". On Linux locate the default bin directory by running: "which -a rinvoker.py".

Now, verify that you can run the example commands against the petstore.

In https://pypi.org/project/unisos.mmwsIcm/, go through the instruction in the sections titled:
"Binaries And Command-Line Examples"
"Remote Invoker (rinvoker-svc.py) Examples"
"Operation Scenario (opScn-svc.py) Examples"

Just run the mentioned commands and verify that you are seeing the mentioned expected outputs.

If these all work right, then you know that the Web-Services Validation Framework has been properly installed and is operational.

10.2 Source Code And Packages Repositories

Complete sources are at:
https://github.com/bisos-pip/mmwsIcm

The PYPI page is at:
https://pypi.org/project/unisos.mmwsIcm

10.3 Applying This Services Validation Framework To Your Own Swagger Specifications

When you have a formal swagger service specification in place, use of these service validation tools is very convenient.

Simply follow the instructions that were provided for the canonical petstore and replace petstore's service specification with your own.

Follow the documentation to build your own scenarios for service specification and use the provided framework to combine multiple scenarios to form regression tests.

11 Use Of Command Line Remote Invocation (rinvoker) Validation Tools

11.1 rinvoker Seed Features – Commands – Paramters – Arguments

11.1.1 rinvoker.py Seed Features – Commands

- Cmnd: -i svcOpsList

```
svcOpsList command digests the Service Specification (swagger-file)
specified on command line as --svcSpec= parameter and produces a
complete list of ALL remotely invokable commands with their corresponding
--resource, --opName and url or body arguments.

Applicable options, parameters and arguments are:

  * Parameter (Mandatory) : --svcSpec=
  * Parameter (Optional)  : --perfSap=  --headers=
```

- Cmnd: -i rinvoker

```
rinvoker command invokes the "opName" operation at "resource" with
specified arguments.

Applicable options, parameters and arguments are:

  * Parameter (Mandatory) : --svcSpec=  --resource=  --opName=
  * Parameter (Optional)  : --perfSap=  --headers=

  * Arguments             : name=value  bodyStr=jsonStr
```

11.1.2 rinvoker.py Seed Features – Parameters

- Parameter: –svcSpec= (url, or swagger-file)

  The swagger file as a url or as a json/yaml file is specified with the –svcSpec= parameter.

- Parameter: –perfSap= (url)
  The Performer Service Access Point Address (perfSap) is specified as a URL with the –perfSap= parameter.
- Parameter: –header= (file)
  Additional headers (e.g., a token) can be included with the –svcSpec= parameter.
- Parameter: –resource= (string, corresponding to SvcSpec)
  The resource to be invoked should be specified with the –resource= parameter
- Parameter: –opName= (string, corresponding to SvcSpec)
  The operation name to be invoked should be specified with the –opName= parameter

11.1.3 rinvoker.py Seed Features – Arguments

- Argument: name=value (string=string corresponding to SvcSpec's URL Params)
- Argument: bodyStr=jsonStr (bodyStr=string corresponding to SvcSpec's Body)

11.2 rinvokerPetstore.py Example

Allows you to list all possible invocations based on a service specification (swagger file).

```
rinvoker.py --svcSpec="http://petstore.swagger.io/v2/swagger.json" -i svcOpsList
```

Allows you to fully specify an invocation on command line. Example:

```
rinvoker.py  --svcSpec="http://petstore.swagger.io/v2/swagger.json"
      --resource="user" --opName="createUser"  -i rinvoke
      bodyStr="{...}"
```

12 Scenarios Invokation Validation Tools

12.1 Invoke-Specifications, Invoke-Verification, Invoke-Reporting

12.1.1 Model Of Invoke – Specification, Verification And Reporting – Scenarios

- Invoke Scenarios Are pure python specification of sequence of invocations.
- Invoke-Expect Scenarios Are pure python specification of sequence of invocations subject to preparations and post-invoke verification and reporting.
- opInvoke class allows for complete invoke specification and complete results to be fully captured.

12.1.2 Scenario Specification For Sequences Of Invocations

In pure python specify invocation of each operation, for example:

```
thisRo = ro.Ro_Op(
    svcSpec=petstoreSvcSpec,
    perfSap=petstoreSvcPerfSap,
    resource="pet",
    opName="getPetById",
    roParams=ro.Ro_Params(
        headerParams=None,
        urlParams={ "petId": 1},
        bodyParams=None,
        ),
    roResults=None,
    )
rosList.opAppend(thisRo)
```

Validation And Reporting Of Invokations

Building on the previously mentioned Operation Specification, in pure python you can the specify Operation Expectations, for example:

```
thisExpectation = ro.Ro_OpExpectation(
    roOp=thisRo,
    preInvokeCallables=[sleep1Sec],
    postInvokeCallables=[ verify_petstoreSvcCommonRo, ],
    expectedResults=None,
    )
roExpectationsList.opExpectationAppend(thisExpectation)
```

preInvokeCallables(ro.Ro_OpExpectation) can include a function that initializes the DB or sleepFor1Sec.

postInvokeCallables(ro.Ro_OpExpectation) can include a function that verifies the result was as expected and then reports success or failure.

12.2 opScn-Seed (Remote Operation Scenarios) – Commands – Paramters – Arguments

12.2.1 opScn Seed Features – Commands

opScn-seed provides the following commands and parameters:

- Cmnd: -i roListInv

  ```
  roListInv command serially invokes the list of ro.Ro_Op() opersations
  specified in the loaded scenario files.

  roListInv displays the invokation and its results. But does not do any verifications.

  Applicable options, parameters and arguments are:

    * Parameter (Mandatory) : --load=
  ```

- Cmnd: -i roListExpectations

```
roListExpectations command serially invokes the list of ro.Ro_OpExpectation()
specified in the loaded scenario files.

roListExpectations displays the invokation and its results and
additionally runs the list of preInvokeCallables and
postInvokeCallables.

postInvokeCallables can include functions that verify the results of the invocation
were as expected.

Applicable options, parameters and arguments are:

  * Parameter (Mandatory) : --load=
```

12.2.2 OpScn Outputs And Reportings

The output format is:

```
* ->:: Invoke Request
* <-:: Invoke Response
* ==:: Invoke Validation (SUCCESS or FAILURE)
```

Additional information for each is include with "**" tags.

This output format can then be used in outline or org-mode.

13 Complete Invoker-Applications Development

13.1 Invoker-Apps Development Model

13.1.1 Invoker-Apps Can Easily Build On unisos.mmwsIcm Capabilities

- Bravado does invoker code-generation on the fly.
- unisos.mmwsIcm.opInvoke – Abstracts invoke-specifications
- unisos.mmwsIcm.wsInvoker – Allows for invokation and verification of opInvoke

With these in place, building Invoke-Apps becomes very simple.

Part VI

Purpose Oriented Uses Of RO-Verifier

14 Uses Of RO-Verifier

RO-Verifier is a general purpose tool. It can be sued for different purposes. It makes it simple and convenient for you to create different invocation scenrios for various purposes.

The RO-Verifier tools view the performer as a blackbox and engage in invocation validation of the service based on its contract (the OpenAPI/Swagger specification). As such, these tools are equally applicable to:

- in-house built web-services.
- public open-source web-services
- private (proprietary) closed-source web-services

There are several distinct purposes for which RO-Verifer can be used. We enumerate some such broad purposes below.

1. Functionality Regression Testing And Verification By Performer Developers
2. Security Oriented Blackbox Repeatable Verification
3. Penetration Testing
4. Complete End-User Oriented Python Applications Development

These purposes often overlap and evolve and morph. For example the scenarios and tools developed for security oriented verifications can be shared with performer developers and then jointly further developed.

15 Security Oriented Use Of RO-Verifier Framework

The scope of RO-Verifier Framework is not inherently just security oriented. However, in this section we emphasize the security oriented use of these tools.

These generalized security oriented validation uses of RO-Verifier fall in the following categories.

1. Validation of the web service's authentication, authorization and access control based on its service specification.
2. Verification of common security vulnerabilities that are manifested in the service specification.

Part VII

Reflections Of Core Features In OpenAPI/Swagger

16 About Core Features And Their Reflections In OpenAPI/Swagger

The criticality of the communication infrastructure in a web-services-based application environment calls for several sophisticated capabilities to be provided as core features.

Web services core features include:

1. Identity and Access Management (IAM)
2. Service discovery
3. Secure communication protocols (confidentiality and integrity)
4. Resiliency or availability improvement techniques

(1) needs to be properly reflected in the OpenAPI/Swagger specification. We'll discuss this in Section 17.

(2) and (3) may be reflected in the OpenAPI/Swagger specification and are subject of Best Current Practices.

(4) is not reflected in OpenAPI/Swagger specification. But its availability (or absence of) may be verified by the proposed tools.

17 Reflections Of IAM In OpenAPI/Swagger

The initial form of authentication to web services involves the use of cryptographic keys. Authentication tokens encoded based on OAuth 2.0 framework provide an option for enhancing security. Additionally, a centralized architecture for provisioning and enforcement of access policies governing access to all web services is required due to the sheer number of services.

A standardized, platform-neutral method for conveying authorization decisions through a standardized token (e.g., JSON web tokens (JWT), which are OAuth 2.0 access tokens encoded in JSON format) is also required.

We call this expression of standardized, platform-neutral method for conveying authorization decisions in the OpenAPI/Swagger specification: "Reflections Of IAM In OpenAPI/Swagger".

Proper security oriented verification of a web service demands proper reflection of reflections of IAM in OpenAPI/Swagger specification.

Swagger 2.0 facilities are incomplete in this regard but OpenAPI 3.0 is richer in this regard.

Based on conventions and local standardization it is possible to reflect a given IAM in OpenAPI/Swagger specification to the extent that automated verification of authentication and access control is possible.

Part VIII

Startegies And Tools For Consistent Continuous Development Of Web Services

Best Current Practices (BCP) for development of OpenAPI/Swagger based web services based on the chosen framework (Dropwizard, Spring-Boot, etc.) should be developed and followed.

All such BCPs needs to have rich scenarios oriented invokers that can be used as regression testers as the web service evolves. It is best to integrate these web services verification framework with the web services development framework, abinitio.

In the ByStar digital ecosystem, the web services development framework, Remote Operations Interactive Command Modules (RO-ICMs) are inherently integrated with the testing framework as they are all based on Direct Operations Interactive Command Modules (DO-ICMs). This model of fully integrating DO-ICMs, with RO-ICMs and the validation framework which are then augmented by player user interfaces is documented in:

> **Unified Python Interactive Command Modules (ICM) and ICM-Players**
> **A Framework For Development Of Expectations-Complete Commands**
> **A Model For GUI-Line User Experience**
> http://www.by-star.net/PLPC/180050 — [3]

Part IX

Alternatives – Risks Associated With This Strategy And Approach – Future

The strategy and approach that we are proposing here heavily counts on industry convergences on:

1. OpenAPI/Swagger as an industry convergence point for web services specifications
2. Python as an industry convergence point for convenient development of scenarios for invocation of web services

Given a 5 year time horizon, there is hardly any risk with (2). However there are some risks with (1).

While it is clearly the case that OpenAPI/Swagger is currently the industry convergence point for web services specifications, that can change.

18 Short Comings Of OpenAPI/Swagger

At this time OpenAPI/Swagger ecosystem is rich with a large number of code-generators and interpreters and is growing. But, there are some fundamental flaws with OpenAPI/Swagger approach, these include:

- Swagger is un-necessarily married to https/http
- Swagger does not support multiple remote operations protocols
- Swagger does not well express the remote operations model and terminology
- Swagger does not well abstract encoding rules beyond json and xml
- Swagger can not play well with IoT

These deficiencies question long term viability of OpenAPI/Swagger.

Competitors such as Protocol Buffers have already appeared. Better remote operations specification standards will likely emerge within this 5 year horizon.

Nevertheless, OpenAPI/Swagger is clearly the right choice right now.

And adapting this very same basic approach to other points of convergence of remote operations specification standards is possible.

Our choice of abstracting to Remote-Operations services at the scenarios layers assists with preservation of investments when underlying providers change.

19 Alternatives To RO-Verifier

Here we mention availability of various tools related to Web Services API Verification.

Flexibility of pure python scenarios creation makes RO-Verifier the most convivial tool. Simplicity, elegance, open-source-ness and power of RO-Verifier framework distinguishes it from other tools in this space.

19.1 postman

Postman https://www.getpostman.com/ is one of the most used tools—if not the most used—when it comes to REST API troubleshooting. Postman is a Chrome app tool used, in its simplest implementation, for executing requests and validating responses. The popularity of Postman is well deserved, as it delivers simple to complex features for everyday users to quickly test HTTP based requests.

Postman is primartily UI based – not batch oriented.

There are some Swagger plug-ins for postman. The end result becomes too heavy and convoluted.

19.2 Assertible

https://assertible.com/ Introduces itself as: "Quality Assurance for the Web". Assertible tests and monitors the executions of your web requests and allows for assertions using JavaScript.

It is primarily a web based tool and not open-source.

References

[1] " Mohsen BANAN ". " a generalized swagger (openapi) centered web services testing and invocations framework ". Permanent Libre Published Content "180057", Autonomously Self-Published, "December" 2018. http://www.by-star.net/PLPC/180057.

[2] " Neda Communications Inc". " bystar internet services operating system (bisos) model, terminology, implementation and usage a framework for cohesive creation, deployment and management of internet services ". Permanent Libre Published Content "180047", Autonomously Self-Published, "July" 2017. http://www.by-star.net/PLPC/180047.

[3] " Neda Communications Inc". " interactive command modules (icm) and players a framework for cohesive generalized scripting a model for gui-line user experience ". Permanent Libre Published Content "180050", Autonomously Self-Published, "July" 2017. http://www.by-star.net/PLPC/180050.

[4] " Neda Communications Inc". " an overview of the libre-halaal bystar digital ecosystem with pointers for digging deeper ". Permanent Libre Published Content "180054", Autonomously Self-Published, "March" 2018. http://www.by-star.net/PLPC/180054.