# Custom Vulnerability Checks with QRDI

User Guide

April 23, 2021

Qualys, Inc.
919 E Hillsdale Blvd
4th Floor
Foster City, CA 94404
1 (650) 801 6100

# Table of Contents

# About this Guide

Thank you for your interest in the Qualys Cloud Platform and custom vulnerability checks! This guide tells you how to add custom vulnerabilities using Qualys Remote Detection Interface (QRDI) and execute them by launching Qualys Vulnerability Management scans.

## About Qualys

Qualys, Inc. (NASDAQ: QLYS) is a pioneer and leading provider of cloud-based security and compliance solutions. The Qualys Cloud Platform and its integrated apps help businesses simplify security operations and lower the cost of compliance by delivering critical security intelligence on demand and automating the full spectrum of auditing, compliance and protection for IT systems and web applications.

Founded in 1999, Qualys has established strategic partnerships with leading managed service providers and consulting organizations including Accenture, BT, Cognizant Technology Solutions, Deutsche Telekom, Fujitsu, HCL, HP Enterprise, IBM, Infosys, NTT, Optiv, SecureWorks, Tata Communications, Verizon and Wipro. The company is also a founding member of the Cloud Security Alliance (CSA). For more information, please visit www.qualys.com

## Qualys Support

Qualys is committed to providing you with the most thorough support. Through online documentation, telephone help, and direct email support, Qualys ensures that your questions will be answered in the fastest time possible. We support you 7 days a week, 24 hours a day. Access support information at www.qualys.com/support/

# Get Started

Now you can easily add custom vulnerabilities (QIDs) using Qualys Remote Detection Interface (QRDI) and execute them by launching Qualys Vulnerability Management (VM) scans via the Qualys Cloud Platform UI and API.

For each QRDI vulnerability you'll provide:

- vulnerability settings similar to Qualys provided vulnerability (i.e. title, QID, severity, threat, impact, solution, mappings), and

- QRDI definition, in valid JSON format, that describes the logic of the vulnerability detection. Simple HTTP and TCP requests are supported. Note that we do not follow HTTP redirects.

- Lua function library shared by all QRDI detections. This allows customers to write Lua functions which can be referenced by JSON documents to implement certain parts of a detection, e.g. to calculate buffer content to be sent, or to implement custom parsing rules.

QRDI detection results:

- The output of a QRDI vulnerability detection is similar to any Qualys provided vulnerability detection, i.e. QID instances appear in scan reports, API output, asset information etc. in the same way.

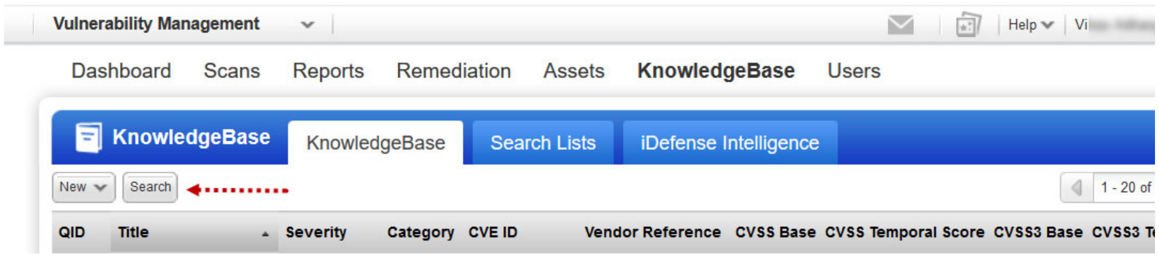
- When a debugging level is set for a QRDI QID, the detected Vulnerability/Information Gathered in the Scan Results is not processed and added to host history/trend info. The detection Result and Debug data will appear in the Scan Results report and Scan-based Scan report, but will not be present in Host-based Scan Report. Further, any ticketing rules for a QRDI QID (if they exist) are not triggered, when a QRDI has debugging enabled.

## What you'll need

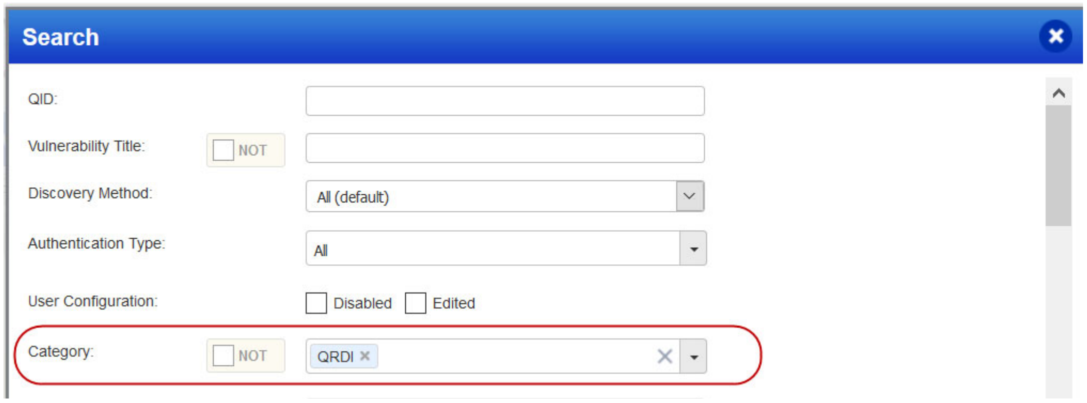
Qualys Cloud Platform account with Qualys Custom QRDI Checks enabled. Manager role is required to create and edit custom QRDI checks. Need some help with this? Just reach out to Qualys Support or your Qualys Account Manager.

## Let's have a look at QRDI vulnerabilities

Once you've added custom QRDI vulnerabilities they appear in the KnowledgeBase section in your account, just like Qualys provided vulnerabilities. Click the Search option to quickly find all your custom QRDI vulnerabilities.

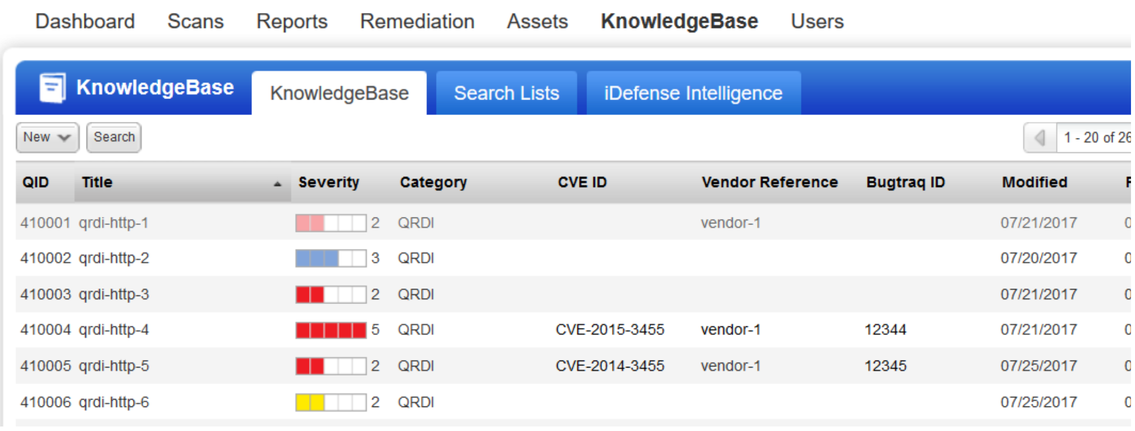


Choose the category QRDI for your search:



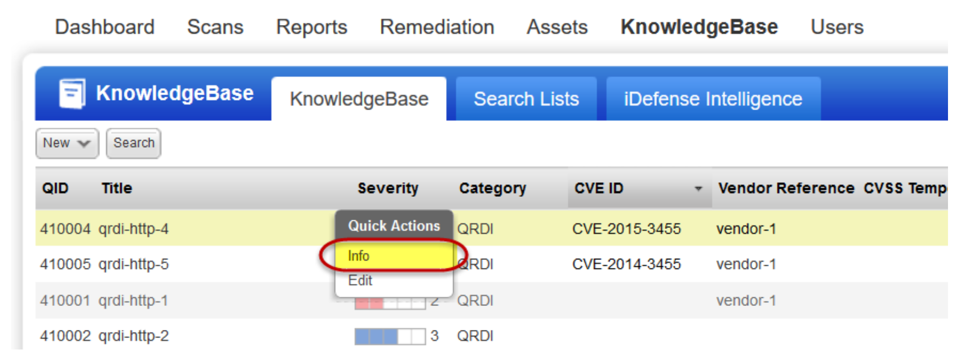
| | | |
|---|---|---|
| QID: | | |
| Vulnerability Title: | ☐ NOT | |
| Discovery Method: | | All (default) |
| Authentication Type: | | All |
| User Configuration: | | ☐ Disabled ☐ Edited |
| Category: | ☐ NOT | QRDI × |

Here's a sample list of QRDI vulnerabilities:



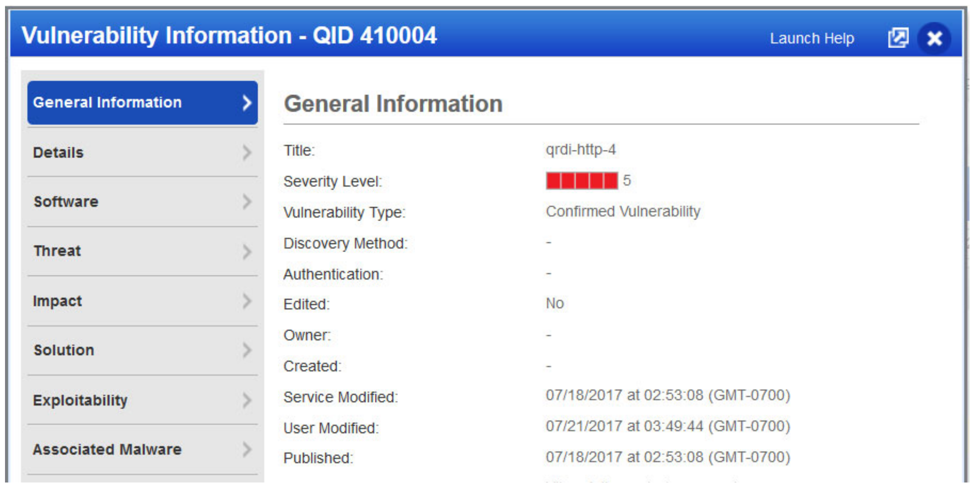
| QID | Title | Severity | Category | CVE ID | Vendor Reference | Bugtraq ID | Modified | F |
|---|---|---|---|---|---|---|---|---|
| 410001 | qrdi-http-1 | 2 | QRDI | | vendor-1 | | 07/21/2017 | 0 |
| 410002 | qrdi-http-2 | 3 | QRDI | | | | 07/20/2017 | 0 |
| 410003 | qrdi-http-3 | 2 | QRDI | | | | 07/21/2017 | 0 |
| 410004 | qrdi-http-4 | 5 | QRDI | CVE-2015-3455 | vendor-1 | 12344 | 07/21/2017 | 0 |
| 410005 | qrdi-http-5 | 2 | QRDI | CVE-2014-3455 | vendor-1 | 12345 | 07/25/2017 | 0 |
| 410006 | qrdi-http-6 | 2 | QRDI | | | | 07/25/2017 | 0 |

Drill down to vulnerability info. Hover over the QID of interest and select Info from the Quick Actions menu.



| QID | Title | Severity | Category | CVE ID | Vendor Reference | CVSS Temp |
|---|---|---|---|---|---|---|
| 410004 | qrdi-http-4 | | QRDI | CVE-2015-3455 | vendor-1 | |
| 410005 | qrdi-http-5 | | QRDI | CVE-2014-3455 | vendor-1 | |
| 410001 | qrdi-http-1 | 2 | QRDI | | vendor-1 | |
| 410002 | qrdi-http-2 | 3 | QRDI | | | |

View vulnerability info and edit settings as needed:



General Information

| | |
|---|---|
| Title: | qrdi-http-4 |
| Severity Level: | 5 |
| Vulnerability Type: | Confirmed Vulnerability |
| Discovery Method: | - |
| Authentication: | - |
| Edited: | No |
| Owner: | - |
| Created: | - |
| Service Modified: | 07/18/2017 at 02:53:08 (GMT-0700) |
| User Modified: | 07/21/2017 at 03:49:44 (GMT-0700) |
| Published: | 07/18/2017 at 02:53:08 (GMT-0700) |

### How many QRDI vulnerabilities can I add?

You can add a maximum of 20,000 QRDI vulnerabilities to your subscription.

### What about the vulnerability settings?

When you add a QRDI vulnerability, you’ll provide title, QID, severity, type, descriptions, optional mappings (CVE IDs, Bugtraq IDs, vendor references) and the QRDI definition in the form of a JSON document. The category is always QRDI. Tip - You can search the KnowledgeBase for category QRDI to find your custom QRDI vulnerabilities.

### Assigning QIDs to QRDI vulnerabilities

You’ll assign a QID within the range 410001-430000 to each QRDI vulnerability. When creating a new QRDI vulnerability, you’ll notice that the QID field is pre-populated with the next available QID. You can keep this or modify it as long as the new QID is within the allowed range and not already in use.

### Can I edit a QRDI vulnerability?

Sure you can edit the general information (title, severity, descriptions, mappings etc.) and the QRDI definition (JSON document) at any time. Note you can't edit these settings: the QID, the category QRDI, and the vulnerability type if it's Information Gathered.

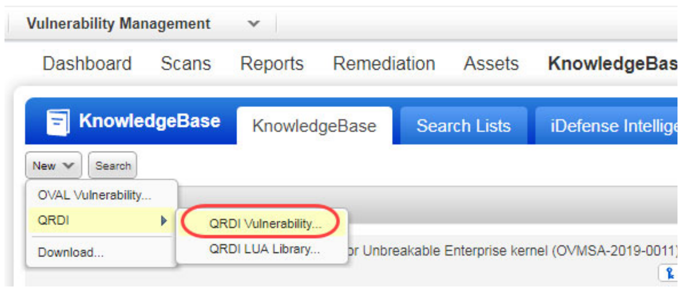
### Can I remove a QRDI vulnerability

It's not possible to remove/delete the QID for QRDI vulnerabilities once added to your subscription.

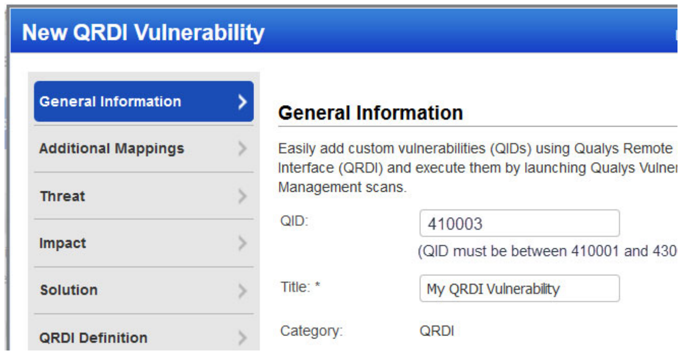
### Is there support to enable/disable QRDI vulnerabilities?

Yes, you can enable/disable a QRDI vulnerability just like a Qualys provided vulnerability by editing the vulnerability settings. Good to Know - When a vulnerability is disabled and it is included in the scan settings (i.e. option profile), it will be scanned like any other vulnerability and it will appear grayed out in scan results and reports.

# How to add QRDI vulnerabilities



Go to the KnowledgeBase and select New > QRDI > QRDI Vulnerability.



General Information

Easily add custom vulnerabilities (QIDs) using Qualys Remote Interface (QRDI) and execute them by launching Qualys Vulne Management scans.

QID: 410003
(QID must be between 410001 and 430

Title: * My QRDI Vulnerabilty

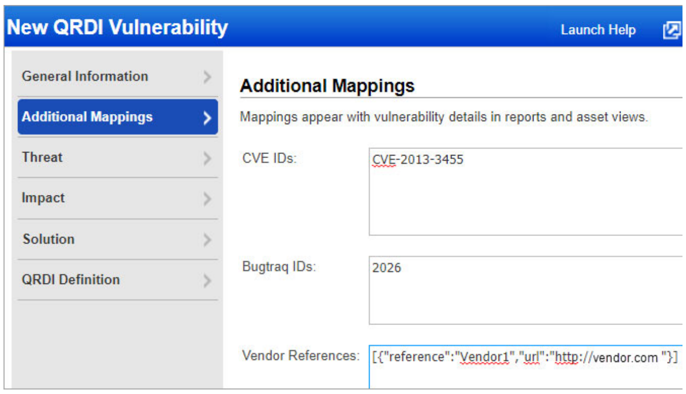
Category: QRDI

Provide vulnerability settings:

- general information (QID, title, type, severity)

- optional mappings (CVE IDs, Bugtraq IDs, vendor references)

- descriptions for threat, impact, solution



Additional Mappings

Mappings appear with vulnerability details in reports and asset views.

CVE IDs: CVE-2013-3455

Bugtraq IDs: 2026

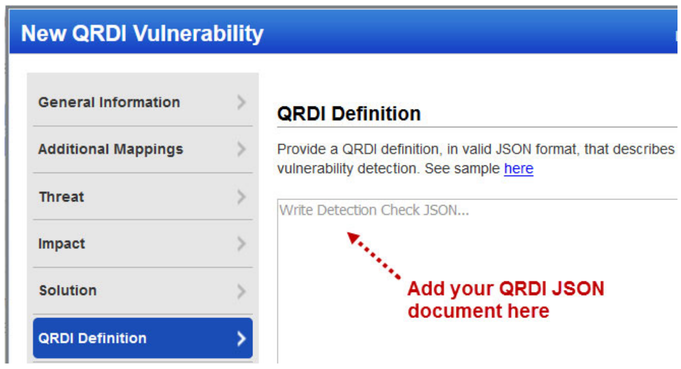
Vendor References: [{"reference":"Vendor1","url":"http://vendor.com "}]

- CVE ID associated with this vulnerability.

- Bugtraq ID number assigned to the vulnerability.

- Vendor reference ID released by the vendor. Provide vendor reference ID and URL link to the vendor's website. Example,

[{"reference":"Vendor1","url":"http://www.vendor.com"}]

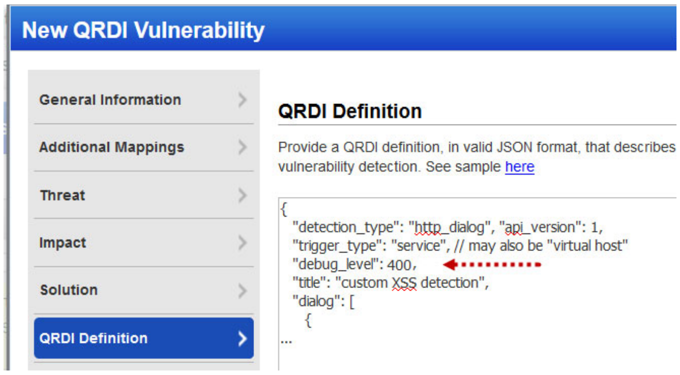


QRDI Definition

Provide a QRDI definition, in valid JSON format, that describes vulnerability detection. See sample here

Upload a JSON document that describes the logic of the detection, i.e. the conditions under which to run the detection, the sequence of data to be sent and received, any pattern matching rules, other data required to run the detection.

Learn more



QRDI Definition

Provide a QRDI definition, in valid JSON format, that describes vulnerability detection. See sample here

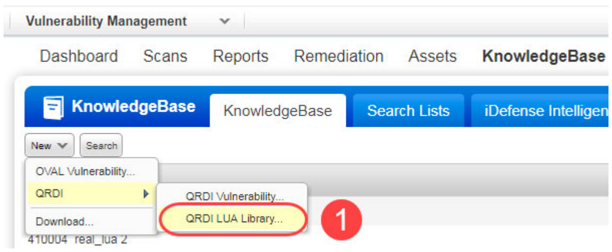
```
{
  "detection_type": "http_dialog", "api_version": 1,
  "trigger_type": "service", // may also be "virtual host"
  "debug_level": 400,
  "title": "custom XSS detection",
  "dialog": [
    {
...
```

We recommend debug mode is enabled for the QRDI vulnerability.

To enable debug mode, add "debug_level" in the top-level JSON object and a value that indicates verbosity of logging info (least info to most info): 100, 200, 300 or 400

## How to add QRDI Lua library

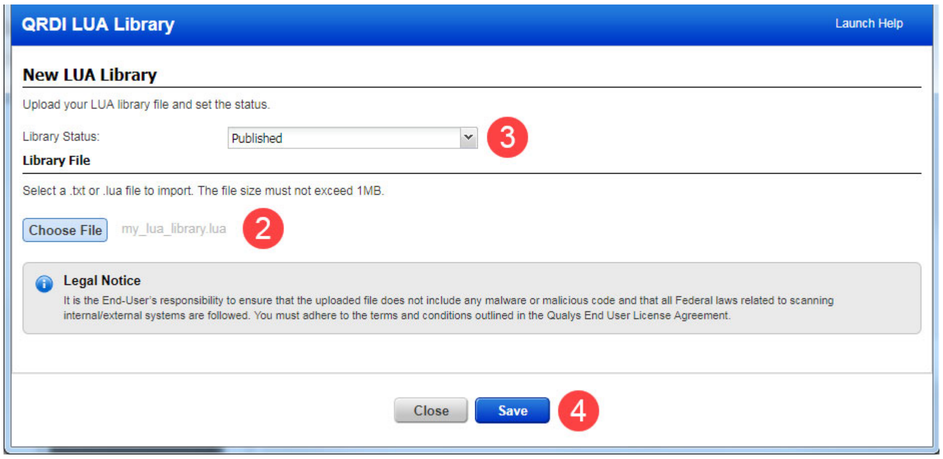
1) Go to KnowledgeBase > New > QRDI > QRDI LUA Library.



2) Click Choose File to browse and select the Lua library file (.lua or .txt) from your system.

3) Set the Library Status to Published to start using it. If you don't want to use the library (perhaps you're still working on it) set the status to Draft or Inactive.

4) Click Save.



New LUA Library

Upload your LUA library file and set the status.

Library Status: Published

Library File

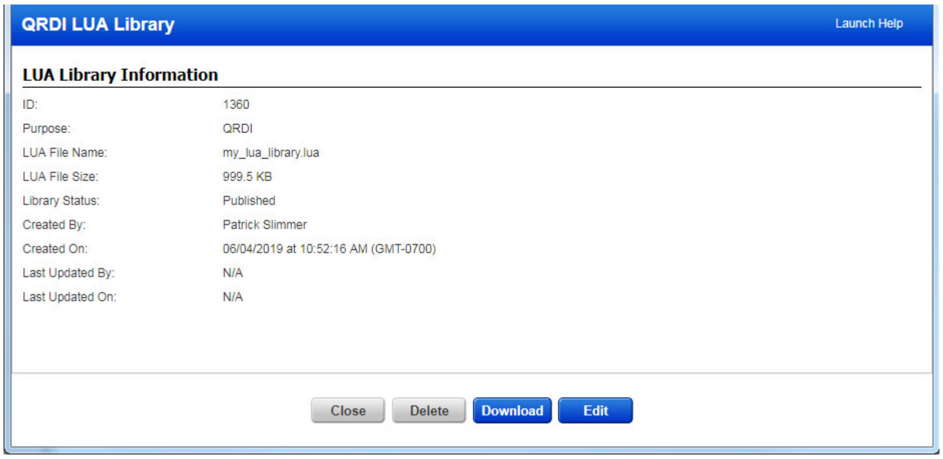
Select a .txt or .lua file to import. The file size must not exceed 1MB.

my_lua_library.lua

Legal Notice

It is the End-User's responsibility to ensure that the uploaded file does not include any malware or malicious code and that all Federal laws related to scanning internal/external systems are followed. You must adhere to the terms and conditions outlined in the Qualys End User License Agreement.

The LUA Library Information screen appears where you can view details.



QRDI LUA Library

LUA Library Information

| | |
|---|---|
| ID: | 1360 |
| Purpose: | QRDI |
| LUA File Name: | my_lua_library.lua |
| LUA File Size: | 999.5 KB |
| Library Status: | Published |
| Created By: | Patrick Slimmer |
| Created On: | 06/04/2019 at 10:52:16 AM (GMT-0700) |
| Last Updated By: | N/A |
| Last Updated On: | N/A |

Feel free to close this window. You can return to it at any time by going to KnowledgeBase > New > QRDI > QRDI LUA Library. From here, you can take these actions:

**Edit** - Click Edit to upload a new/revised library file (remember, there can only be one Lua library file in the subscription at a time) or change the library status.

**Download** - Click Download to download the last saved library file, perhaps to make changes.

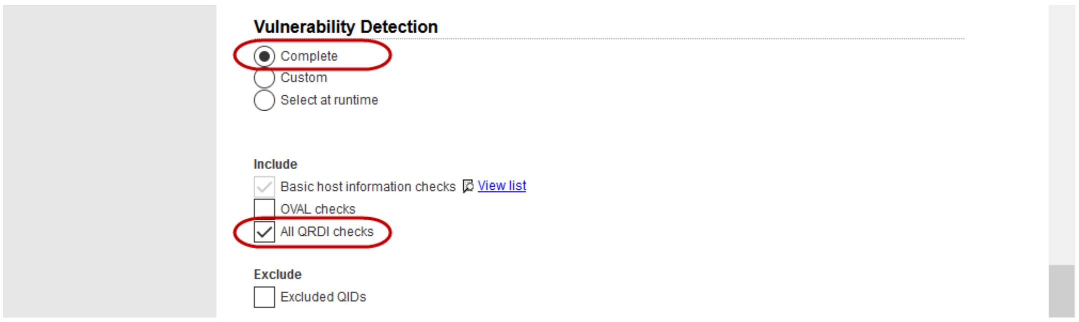
**Delete** - Click Delete to remove the Lua library file from the subscription.

Want to learn more about Lua scripting?

## How to configure scan settings

Customize the Vulnerability Detection section in the option profile you'll use for scanning. There's a few ways to do this.

**Option 1** - Scan for all vulnerabilities in your account. This includes Qualys provided vulnerabilities plus all QRDI vulnerabilities. Select "Complete" and "All QRDI" checks.



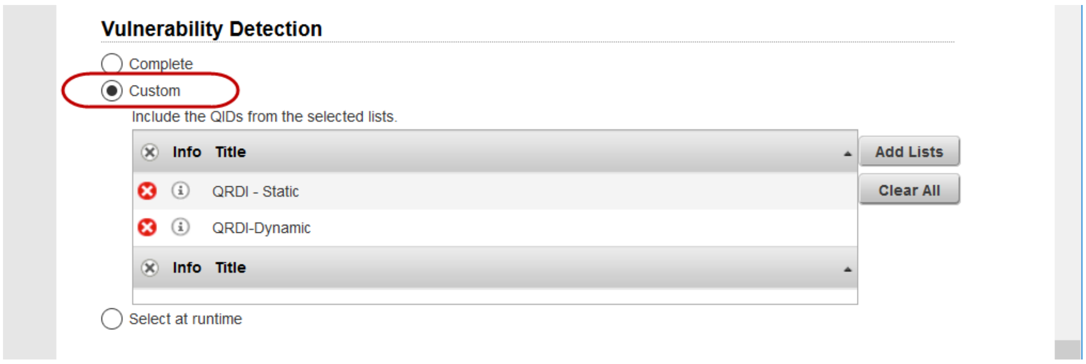
Vulnerability Detection

[✓ Complete] Custom Select at runtime

Include
✓ Basic host information checks
☐ OVAL checks
✓ All QRDI checks

Exclude
☐ Excluded QIDs

**Option 2** - Scan for selected QRDI vulnerabilities. Create search lists (static and/or dynamic) including only QRDI vulnerabilities, select "Custom" and add your custom lists. In this sample QRDI vulnerability checks matching a static search list and a dynamic search list are selected.

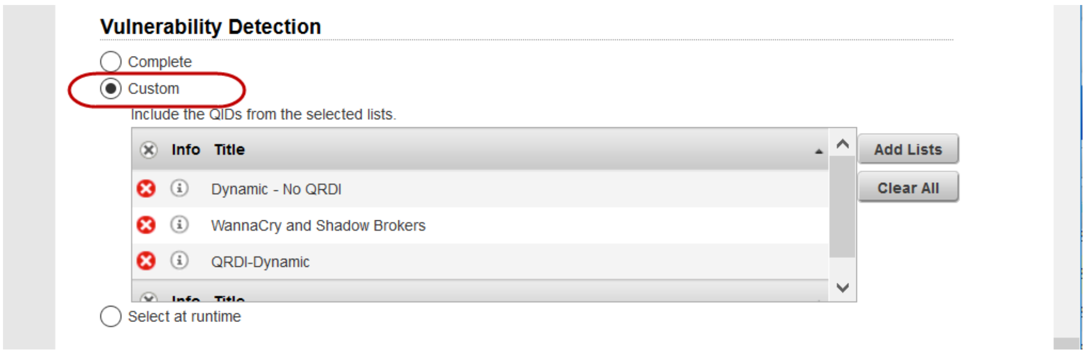

Vulnerability Detection

Complete [✓ Custom]

Include the QIDs from the selected lists.

| Info | Title |
|---|---|
| ⓘ | QRDI - Static |
| ⓘ | QRDI-Dynamic |

**Option 3** - Scan for selected vulnerabilities - Qualys provided and QRDI. Create search lists (static and/or dynamic) including QRDI vulnerabilities plus Qualys vulnerabilities, select "Custom" and add your search lists.


Vulnerability Detection

Complete [✓ Custom]

Include the QIDs from the selected lists.

| Info | Title |
|---|---|
| ⓘ | Dynamic - No QRDI |
| ⓘ | WannaCry and Shadow Brokers |
| ⓘ | QRDI-Dynamic |

## Recommended scan workflow

1) Add QRDI vulnerabilities in debug mode.

How do I do this? Add debug_level: to top level JSON object and set logging level to 100, 200, 300 or 400. Learn more

2) Launch scans on QRDI vulnerabilities (Debug mode enabled).

Scan processing will not be performed for vulnerabilities with Debug mode enabled, and related host based scan data will not be updated in your account.

3) Review scan results and confirm your vulnerability detections are performing as expected.

You'll notice details for QRDI vulnerabilities include additional sections for debug data and errors (see Scan Results). Use the debug information to analyze the results and resolve any issues you may have.

4) Edit QRDI vulnerabilities and disable debug mode.

5) Launch scans on QRDI vulnerabilities (Debug mode disabled).

Scan processing will be performed for vulnerabilities with Debug mode disabled, and related host based scan data will be updated in your account.

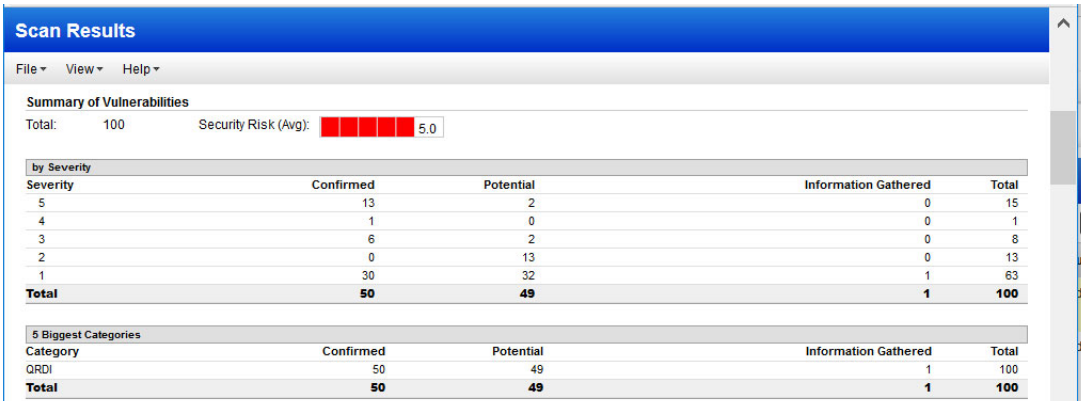
## Scan Results

Scan results return information on all vulnerability checks performed by the scan.

- All QRDI vulnerabilities, as defined in the option profile, are included in scan results

- Debug info is included in detailed results for QRDI vulnerabilities in debug mode

Summary shows vulnerabilities found:.



Summary of Vulnerabilities

Total: 100 Security Risk (Avg): 5.0

by Severity

| Severity | Confirmed | Potential | Information Gathered | Total |
|---|---|---|---|---|
| 5 | 13 | 2 | 0 | 15 |
| 4 | 1 | 0 | 0 | 1 |
| 3 | 6 | 2 | 0 | 8 |
| 2 | 0 | 13 | 0 | 13 |
| 1 | 30 | 32 | 1 | 63 |
| **Total** | **50** | **49** | **1** | **100** |

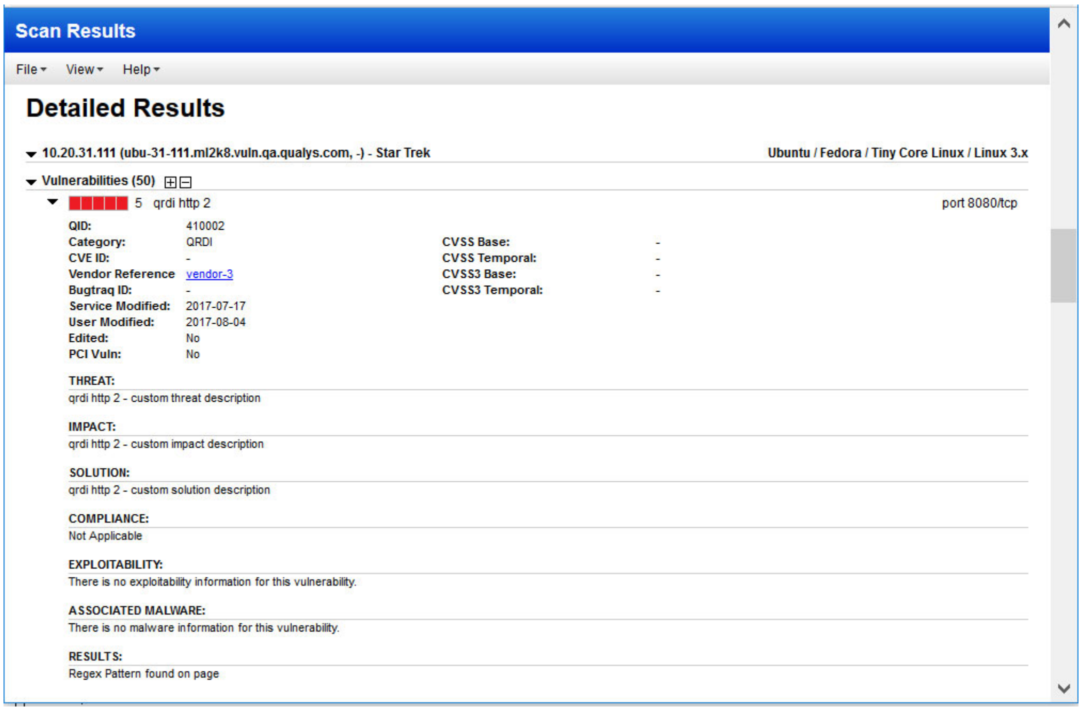
5 Biggest Categories

| Category | Confirmed | Potential | Information Gathered | Total |
|---|---|---|---|---|
| QRDI | 50 | 49 | 1 | 100 |
| **Total** | **50** | **49** | **1** | **100** |

**Debug info in scan results**

Additional data is passed from the scanning engine and shown in scan results for QRDI vulnerabilities in debug mode. In Detailed Results, RESULT DEBUG contains debug data (log messages), always posted with corresponding QID if trigger condition fulfilled. RESULT ERRORS is present if detection ended in error.

Sample QRDI vulnerability not in debug mode - details like Qualys provided QID:



**Detailed Results**

10.20.31.111 (ubu-31-111.ml2k8.vuln.qa.qualys.com, -) - Star Trek — Ubuntu / Fedora / Tiny Core Linux / Linux 3.x

Vulnerabilities (50)

5 qrdi http 2 — port 8080/tcp

| | | | |
|---|---|---|---|
| QID: | 410002 | | |
| Category: | QRDI | CVSS Base: | - |
| CVE ID: | - | CVSS Temporal: | - |
| Vendor Reference | vendor-3 | CVSS3 Base: | - |
| Bugtraq ID: | - | CVSS3 Temporal: | - |
| Service Modified: | 2017-07-17 | | |
| User Modified: | 2017-08-04 | | |
| Edited: | No | | |
| PCI Vuln: | No | | |

THREAT:
qrdi http 2 - custom threat description

IMPACT:
qrdi http 2 - custom impact description

SOLUTION:
qrdi http 2 - custom solution description

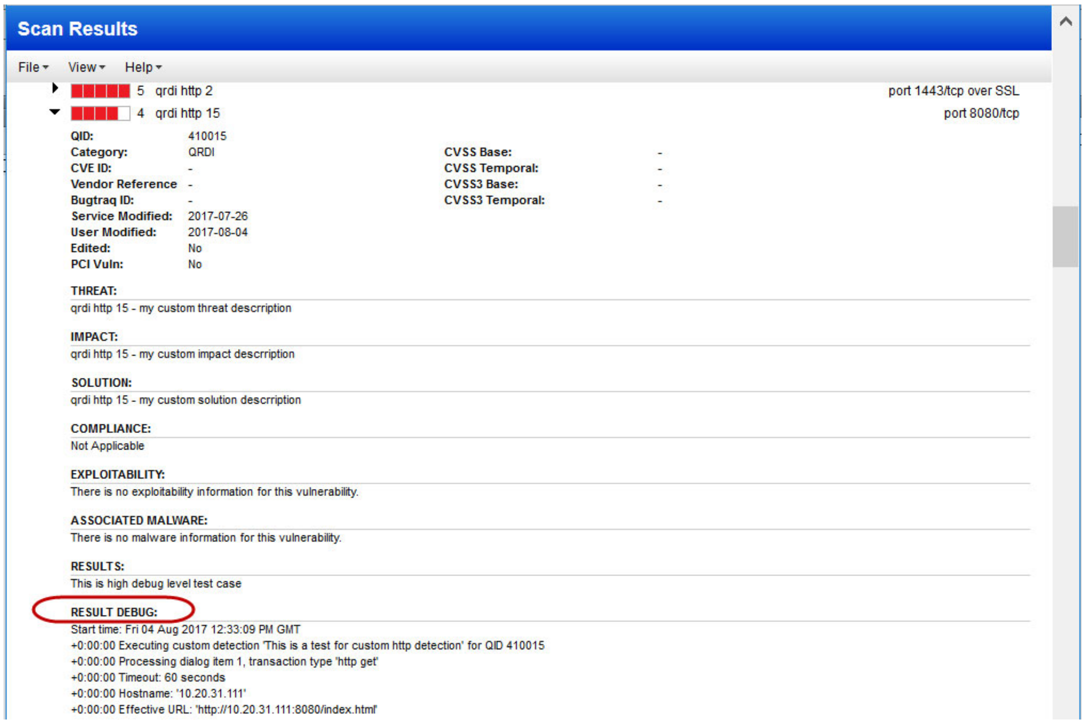
COMPLIANCE:
Not Applicable

EXPLOITABILITY:
There is no exploitability information for this vulnerability.

ASSOCIATED MALWARE:
There is no malware information for this vulnerability.

RESULTS:
Regex Pattern found on page

Sample QRDI vulnerability in debug mode - details include section RESULT DEBUG (logging info) and possibly RESULT ERROR if error encountered:



5 qrdi http 2 — port 1443/tcp over SSL

4 qrdi http 15 — port 8080/tcp

| | | | |
|---|---|---|---|
| QID: | 410015 | | |
| Category: | QRDI | CVSS Base: | - |
| CVE ID: | - | CVSS Temporal: | - |
| Vendor Reference | - | CVSS3 Base: | - |
| Bugtraq ID: | - | CVSS3 Temporal: | - |
| Service Modified: | 2017-07-26 | | |
| User Modified: | 2017-08-04 | | |
| Edited: | No | | |
| PCI Vuln: | No | | |

THREAT:
qrdi http 15 - my custom threat descrription

IMPACT:
qrdi http 15 - my custom impact descrription

SOLUTION:
qrdi http 15 - my custom solution descrription

COMPLIANCE:
Not Applicable

EXPLOITABILITY:
There is no exploitability information for this vulnerability.

ASSOCIATED MALWARE:
There is no malware information for this vulnerability.

RESULTS:
This is high debug level test case

RESULT DEBUG:

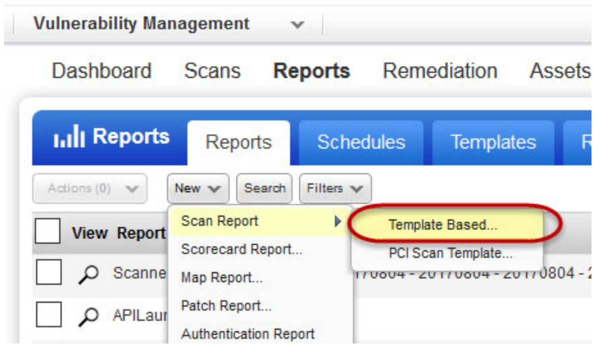
```
Start time: Fri 04 Aug 2017 12:33:09 PM GMT
+0:00:00 Executing custom detection 'This is a test for custom http detection' for QID 410015
+0:00:00 Processing dialog item 1, transaction type 'http get'
+0:00:00 Timeout: 60 seconds
+0:00:00 Hostname: '10.20.31.111'
+0:00:00 Effective URL: 'http://10.20.31.111:8080/index.html'
```

Sample QRDI vulnerability in debug mode with tag <RESULT_DEBUG> (XML format):

```
...
<VULN number="410015" severity="4">
        <TITLE><![CDATA[qrdi http 15]]></TITLE>
        <LAST_UPDATE><![CDATA[2017-08-
04T17:20:39Z]]></LAST_UPDATE>
        <CVSS_BASE></CVSS_BASE>
        <CVSS_TEMPORAL>-</CVSS_TEMPORAL>
        <CVSS3_BASE></CVSS3_BASE>
        <CVSS3_TEMPORAL>-</CVSS3_TEMPORAL>
        <PCI_FLAG>0</PCI_FLAG>
        <DIAGNOSIS><![CDATA[qrdi http 15 - my custom threat
descrription]]></DIAGNOSIS>
        <CONSEQUENCE><![CDATA[qrdi http 15 - my custom impact
descrription]]></CONSEQUENCE>
        <SOLUTION><![CDATA[qrdi http 15 - my custom solution
descrription]]></SOLUTION>
        <RESULT><![CDATA[This is high debug level test
case]]></RESULT>
        <RESULT_DEBUG><![CDATA[Start time: Fri 04 Aug 2017
12:33:09 PM GMT
+0:00:00 Executing custom detection 'This is a test for custom
http detection' for QID 410015
+0:00:00 Processing dialog item 1, transaction type 'http get'
+0:00:00 Timeout: 60 seconds
+0:00:00 Hostname: '10.20.31.111'
+0:00:00 Effective URL: 'http://10.20.31.111:8080/index.html'
+0:00:00 Returned data '\n'
+0:00:00 Returned data '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">\n'
...
+0:00:00 Dialog item 3 finished with action 'stop'
+0:00:00 Custom detection returned success]]></RESULT_DEBUG>
</VULN>
...
```

## Reporting

Latest QRDI vulnerability detections appear in reports and API output unless the QRDI vulnerabilities are in Debug mode.



# QRDI Vulnerability Definition

A QRDI definition, in valid JSON format, describes the logic of the vulnerability detection. Simple HTTP and TCP requests are supported. Note that we do not follow HTTP redirects.

This section is the official reference of the JSON schema supported by QRDI.

QRDI detections are based on these third-party technologies:

The JSON document standard (www. json.org) to formally describe structured data

The Lua programming language (www.lua.org), currently version Lua 5.3

PCRE - Perl Compatible Regular Expressions (www.pcre.org)

## JSON document overview

The JSON standard supported by QRDI is compatible with ECMA-404. In addition, QRDI permits the use of JavaScript-style comments which start with "//" and extend to the end of the line. The top-level item of any JSON document supported by QRDI is a JSON “object”, with some mandatory and some optional fields. HTTP detection type is supported at this time.

**Example 1 - JSON document describing a valid HTTP detection**

```
{
    "detection_type": "http dialog",
    "api_version": 1,
    "trigger_type": "service", // may also be "virtual host"
    "title": "custom XSS detection",
    "dialog": [
        {
            "transaction": "http get",
            "object": "/cgi-
bin/no5_such3_file7.pl?\"><script>alert(73541);</script>"
        },
        {
            "transaction": "process",
            "mode": "regexp",
            "match": "\"><script>alert\\(73541\\);</script>"
        },
        {
            "transaction": "report",
            "result": "XSS found"
        }
    ]
}
```

#### Example 2 - JSON document describing a valid HTTP detection using "http_header"

An "http_header" field is a permitted field of "http get" and "http post" transactions. This field can contain a single string with linefeed-separated header lines, which are added to HTTP requests. The string can be calculated dynamically using eval-expressions.

```
{
    "detection_type": "http dialog",
    "api_version": 1,
    "trigger_type": "service",
    "title": "Custom http header test detection",
    "ports": 8080,
    dialog": [
        {
            "transaction": "http get",
            "http_header": "HEADER_TEST: HeaderTest\nCOMPANY:
Company_name\nUser-Agent: UserAgentTest",
            "object": "index.html",
            "label" : "label_to_start_transaction"
        },
        {
            "transaction": "process",
            "label" : "label_to_process_transaction",
            "mode": "luapattern",
            "match": "xterm"
        },
        {
            "transaction": "report",
            "label" : "label_to_report_transaction",
            "result": "Luapattern is detected"
        }
    ]
}
```

## Common concepts

### Variables

QRDI supports the notion of “variables”. A variable is a storage slot used to store temporary data during the execution of a detection, e.g. a variable might be used to store part of the result of one query which can then be used in the formatting of a second query, as part of the same detection, or to store data in preparation for reporting. The scope of each variable is a single QRDI detection, i.e. values of variables are not preserved across different QRDI detections.

There are two types of variables: user variables and system variables. System variables have read-only permissions from the user's point of view. They are set to a defined value by the QRDI run-time, and can be read by the JSON detection, but cannot be modified.

Example: With HTTP detections, after every “http get” or “http put” transaction the HTTP body of the result returned from the web server is available in the system variable “body”.

User variables may be freely created, read and written from a JSON detection.

## Eval-expressions

Several places in the QRDI JSON schema allow values to be dynamically calculated by using an “eval-expression”. Currently eval-expressions allow the use of constants and string concatenation.

The following items are permitted anywhere eval-expressions are allowed:

String constant, e.g.

```
"abc"
```

Numeric constant, e.g.

```
123
```

```
45.6
```

User variable reference, e.g.

```
{"user": quoted_variablename}
```

System variable reference, e.g.

```
{"system": quoted_variablename}
```

String concatenation of items in an array, e.g.

```
{ "concat": [eval-expression  ...]}
```

A more complex concatenation example:

```
{"concat": "abc", {"system": "body"}, 123, {"user": "myvarl"}]}
```

The result of this eval-expression is a string constructed by concatenating the following items:

"abc"

The value of the system variable body

"123" (automatically converted from an integer to a string)

The value of the user  variable myvarl

## Dialogs and transactions

Every JSON document describing a QRDI detection has to contain a "dialog" field that describes the dialog underlying the detection, as shown in the example above. The value of a dialog field is a JSON array consisting of one or more transactions. Each “transaction” describes one step of the QRDI detection. Transactions are usually executed serially, in the order specified, but that behavior can be modified by certain “return actions”, as described below. Each transaction can have the following fields, plus any fields specific to the transaction type:

**`"transaction"`**: mandatory field that contains the transaction type as a string constant. The supported transaction types depend on the type of detection.

**`"label"`**: optional field that assigns a label (name) to a transaction. This is used in combination with the "goto" return action described below, to allow the transaction to be the target of a "goto" jump. Labels have to be unique within a QRDI detection.

The only transaction type shared by all types of detections is the "report" transaction, which has the following format:

```
{
"transaction": "report",
"result":  eval-expression
}
```

A **`"report"`** transaction is always the last transaction run in a detection. It calculates the result from its eval-expression and then posts that result as the cleartext result of the QID associated with the detection. Processing of the detection stops after that.

Processing of the transaction array of a detection can end in one of the following ways:

- By reaching a **`"report"`** transaction. This causes the QID to be posted with the cleartext result provided in the **`"result"`** field of the report transaction.

- By reaching the end of the transaction array. This causes the QID not to be posted.

- By receiving a **`"stop"`** return action (described below). This causes the QID not to be posted.

- By encountering an error, or an **`"error"`** return action (described below). This causes the QID to be posted with an error message in the cleartext section of the QID result and in the /INFO/ERROR XML tag of the XML section of the QID result.

## Return actions

“Return actions” are used in several places of the JSON schema. A return action defines the result of a transaction and tells the QRDI run-time what to do next, e.g. to abort the detection, continue with the next transaction, continue with a different transaction, or throw an error. The following return actions are currently supported. Simple return actions, which do not take a parameter, can be specified as constant strings. Return actions that require a parameter are specified using a JSON object.

**`"continue"`** : this is the default and means that the run-time will continue with the next transaction in the transaction array. If the end of the transaction array is reached then the processing of the detection ends, and the QID is not posted.

**`"stop"`**: this tells the run-time to stop processing the current detection. The QID is not posted in this case.

**`"report"`**: this tells the run-time to skip to the next transaction, in sequential order, which is of type **`"report"`**. If no such transaction is found then this behavior is equivalent to **`"stop".`**

**`{"action": "goto", "label": quoted_targetlabel}`**: this tells the run-time to continue with the transaction that has the specified label. If such a transaction does not exist then an error is thrown, and the detection processing ends with an error condition.

**`{"action": "error", "message": quoted_errormsg}`**: this causes the processing of the detection to stop and for an error to be thrown. This causes the QID to be posted with an error, as described above.

## JSON syntax common to all detections

The following fields are supported in the top-level JSON object and are common to all types of remote detections:

**`"detection_type"`**: required field and describes the type of the QRDI detection. Currently we support types **`"tcp dialog"`** and **`"http dialog"`**.

**`"api_version"`**: required field and describes the QRDI version for which this detection was written. At this time the only supported value is the integer **`1`**.

**`"trigger_type"`**: required field which indicates the condition or event that should trigger the QRDI detection. For supported values see HTTP dialog detections.

**`"debug_level"`**: optional field which indicates the verbosity of debug information (logging) being returned for each instance of a QRDI detection. The following values are permitted:

-- **`0`**: no debug information is generated. This is the default.

-- **`100`**: Only the start and end of a QRDI detection are logged

-- **`200`**: In addition to the information from 100, the start and end of each transaction and run-time errors are logged

-- **`300`**: In addition to the information from 200, the values of variables and temporary data is logged. However string values are only logged if they contain valid UTF-8 strings without special characters, and if the string value has fewer than 1024 characters.

-- **`400`**: Similar to 300, but strings which do not contain valid UTF-8 or which are longer than 1024 characters are logged as well, in a combined binary/hex dump if necessary. For strings longer than 8kB only the first 4kB and the last 4kB are logged.

**`"dialog"`**: required field that contains the transactions which are run for the detection. The format of supported transactions is described in HTTP dialog detections.

**"timeout"**: optional field that contains the overall timeout in milliseconds. This is the maximum total time spent running the detection. The default is 60000 (60 seconds). The maximum permitted value is 180000 (180 seconds).

**"os"**: optional field that contains a PCRE-compatible regular expression that needs to match the scan target's operating system in order for the detection to be run. The type of OS fingerprint that this field is compared against is described in HTTP dialog detections.

**"not_os"**: optional field that contains a PCRE-compatible regular expression that needs to NOT match the scan target's operating system in order for the the detection to be run. The type of OS fingerprint that this field is compared against is described in HTTP dialog detections.

**"ports"**: optional field that restricts the detection to only run on certain ports. The value can be a single integer or a JSON array of integers. Each integer represents a single port number. The default is to not restrict the ports that the detection is run on, i.e. to run the detection on any port for which the trigger condition or event applies.

**"title"**: optional field that contains a title (description) of the detection. If present the value has to be a string constant. The value has no functional impact on the detection and is only used in some log messages.

## Output format

If a QID is posted then the output of the QID can contain the following sections:

cleartext section: For detections that ended with a **"report"** transaction this section will contain the value of the **"result"** eval-expression calculated in the **"report"** transaction. For detections that ended in an error this section will contain the error message.

**<RESULT_ERRORS>** XML tag in scan results: this tag is only present for detections that ended in an error. In that case the tag contains the error message.

**<RESULT_DEBUG>** XML tag in scan results: this tag will contain the debug data (log messages) generated by the detection during its processing, wrapped into CDATA, if the debug_level was greater than zero. This tag is produced regardless of the end status of the detection, i.e. even if the detection ended in a way that would otherwise not have caused the QID to be posted (e.g. with a **"stop"** return action). This also means that with a **"debug_level"** greater than zero a detection will always post its corresponding QID if the trigger condition was fulfilled, regardless of its end status.

# HTTP dialog detections

Qualys supports HTTP type detections using QRDI. For this type of detection the **`"detection_type"`** in the top-level JSON object has to be set to **`"http dialog"`**.

An HTTP dialog detection can be triggered in one of the following two ways:

**With "trigger_type" set to "service"**

If the scanner engine finds an HTTP or HTTPS service running on a TCP port on the target during TCP service discovery then any HTTP dialog detection with this trigger type is executed on that port. The **`"Host:"`** field of any HTTP request is set to the IP address of the target.

**With "trigger_type" set to "virtual host"**

With this trigger type the detection will be run later in the VM scan. After HTTP/HTTPS service discovery the scanner engine performs web server fingerprinting and then attempts to validate the customer-configured list of virtual hosts for the current IP address and port. HTTP dialog detections with this trigger type will then be run for each validated virtual host. The **`"Host:"`** field of any HTTP request is set to the virtual host name. With this trigger type it is possible for an HTTP dialog detection to be run multiple times on a single IP address and port, once for each virtual host. QID instances generated by these detections are reported and tracked separately on the Qualys Cloud Platform.

The operating system used for matching the **`"os"`** and **`"not_os"`** fields in the top-level JSON object is the operating system found during TCP fingerprinting, i.e. the operating system reported in QID 45017 with the **`"Technique"`** column set to **`"TCP/IP Fingerprint"`**. That fingerprint may not be accurate if the target is behind a firewall.

HTTP dialog detections support four types of transactions: **`"http get"`**, **`"http post"`**, **`"process"`** and **`"report"`**. **`"http get"`** and **`"http post"`** perform an HTTP transaction on the target, i.e. they send an HTTP GET or POST request to the target and wait for the response. All data encoding and decoding is done by the run-time. **`"process"`** is used to process the results of the most recent **`"http get"`** or **`"http put"`** transactions. **`"report"`** processing works as described earlier.

## "http get" and "http post" transactions

Supported fields:

**`"object"`**: *eval-expression*. This is a required field that evaluates to the part of the URL that follows the host name and the " : ". It usually starts with a "/" and contains the path, file name and, for GET requests, any encoded CGI values, as required by the target web application. No URL encoding is performed by the run-time, i.e. the value in this field already needs to be in URL-encoded form.

**`"data"`**: *eval-expression*. This field is required for **`"http post"`** transactions and ignored for **`"http get"`** transactions. It evaluates to the data sent in the body of the HTTP POST request.

**"on_error"**: *return action*. This optional field changes the return action in the case of a network error. The default return action is **"error"** with the **"message"** field set to a meaningful value, which results in the detection processing to stop in case of an error. If an error encountered during a particular transaction should not result in detection processing to stop then the return action could, e.g., be set to **"continue"** or **"goto"** with an appropriate target label. Only network errors are affected by this field. HTTP error codes returned by the web server do not trigger this return action and can be handled separately, using **"http_ status_map"**, described below.

**"http_status_map"** : By default only network errors result in a return action of **"error"**. HTTP transactions which are successful at the network level but result in an HTTP status code different from 200 ("OK") are still considered successful and result in a return action of **"continue"**. This optional field allows the return action to be customized based on the HTTP status code. This can, e.g., be used to treat certain types of HTTP status codes (e.g. 404) as errors. The value of this field is a JSON array, and each member of the array is a JSON object with two fields: **"status"** and **"action"**. The **"status"** field can contain a single status code or an array of status codes. The **"action"** field contains the corresponding return action. Status codes can be integers, for an exact match, or strings, for a PCRE regular expression match.

Example:

```
{
        "http_status_map": [{
                        "status": [403, 404],
                        "action": {
                                "action": "error",
                                "message": "http error"
                        }
                },
                {
                        "status": "3[0 .. 9] [0 .. 9]",
                        "action": {
                                "action": "goto",
                                "label": "process_3xx"
                        }
                }
        ]
}
```

With this example HTTP status codes 403 and 404 are treated as errors, causing the detection to stop, and HTTP status codes 3xx cause processing to continue with the transaction labeled **"process_3xx"**.

**"timeout"** : Optional field that contains the timeout in milliseconds for the HTTP data transfer. The default is 60000 (60 seconds) and only applies to the HTTP data transfer, not the TCP connection establishment, which has a different fixed timeout of 15 seconds. In addition the overall HTTP transfer is subject to the global time window defined in the top-level JSON object.

**"http_header"** : *eval-expression*. This is an optional field with a single string containing line feed-separate header lines that are added to the HTTP header when sending the request. Even though the HTTP standard requires "\r\n" to be used as a separator only a single line feed ("\n") is needed as a separator here. If a line starting with "User-Agent:" is included as one of the header lines then that line replaces the user agent value usually sent by the scanner engine.

Example:

```
"http_header": "Header1: value1\nUser-Agent: foo\n"
```

After the transaction completes, the following system variables are set by the run-time, for use by subsequent transactions.

**"body"**: Content of the HTTP response body (payload).

**"network_status"**: A short text string describing the type of network error, or "OK" if no error has occurred.

**"http_status"**: An integer with the HTTP status code of the response (0 in the case of a network error).

## "process" transactions

Supported fields:

**"mode"**: Optional field, which can take one of the following values. The default is **"substring"**. This field determines the type of pattern matching that should be performed on the input.

-- **"substring"**: the search expression has to be a case-sensitive substring of the source.

-- **"luapattern"**: the search expression is interpreted as a Lua pattern, as described in the Lua documentation in the chapter labeled "Patterns".

-- **"regexp"**: the search expression is interpreted as a PCRE-compatible regular expression, executed with the PCRE options DOTALL and UTF8.

**"match"**: *eval-expression*. This is a required field which provides the search expression as a string. The exact meaning depends on the **"mode"**, as described above. In regexp terms this is called the "needle".

**"source"**: *eval-expression*. This is an optional field which provides the data to search. The default is the content of the system variable **"body"**. In regexp terms this is called the "haystack".

**"extract"**: For transactions with a **"mode"** of **"luapattern"** or **"regexp"** it is possible to extract sections from the matched source string and copy them into user variables. This is indicated in the match string by surrounding the corresponding sub-pattern in " () ", as described in the Lua and PCRE documentations. The value of the **"extract"** field is a JSON array that contains one JSON object for each variable to be extracted. Each object needs to have the format **{"var": quoted_varname}**.

Variables are mapped to sub-patterns and extracted in the order in which they appear in the array. The first item in the array always matches the complete matched string (index 0 in PCRE). All subsequent items in the array match actual sub-patterns.

Example:

```
"mode": "regexp",
"source": "abcdefghijkl",
"match":  "b(c.*f)g(h.*j)",
"extract":  [{"var": "v1"}, {"var": "v2"}, {"var": "v3"}]
```

This causes the following user variable assignments:

```
"v1 = "bcdefghij"
"v2 = "cdef"
"v3 = "hij"
```

**`"on_found"`**: *return action*. This optional field changes the return action in the case that the pattern has matched. The default return action is **`"continue"`**.

**`"on_missing"`**: *return action*. This optional field changes the return action in the case that the pattern has not matched. The default return action is **`"stop"`**.

# TCP dialog detections

Qualys supports TCP type detections using QRDI. For this type of detection the **`"detection_type"`** in the top-level JSON object has to be set to **`"tcp dialog"`**.

A TCP dialog detection can be triggered in one of the following two ways:

### With "trigger_type" set to "service"

With this setting the detection is triggered if the scanner engine detects a TCP service running on a target listening port, and the name of that service matches one of the names listed in the **`"services"`** field. **`"services"`** is a mandatory field that contains a Json array of strings that represent service names. The names are the same names that appear in the **`"Service Detected"`** column of QID 82023 ("Open TCP Services List").

### With "trigger_type" set to "port"

With this setting the detection is triggered if the scanner engine detects an open TCP listening port on the target, and the port number is listed in the **`"ports"`** field. **`"ports"`** is a mandatory field that contains a Json array of TCP port numbers.

The operating system used for matching the **`"os"`** and **`"not_os"`** fields in the top-level JSON object is the operating system found during TCP fingerprinting, i.e. the operating system reported in QID 45017 with the **`"Technique"`** column set to **`"TCP/IP Fingerprint"`**. That fingerprint may not be accurate if the target is behind a firewall.

The scanner engine automatically detects whether a service is running over SSL/TLS, and also, for some services, whether the service supports upgrading the connection to SSL/TLS via a STARTTLS mechanism. STARTTLS is currently supported for the following services:

-- telnet

-- smtp

-- imap

-- pop3

-- ldap

-- ftp

-- PostgreSQL

By default a detection is triggered regardless of whether SSL/TLS is negotiated or not, and alway runs at the highest available security level, i.e. if SSL/TLS is not available then the detection executes over plain TCP, but if SSL/TLS is available (either natively on the port or dynamically via STARTTLS) then SSL/TLS is first negotiated, as part of the connection process, and the detection then executes over SSL/TLS. It is possible to override this behavior, i.e. to limit a detection to only run if SSL/TLS is available or not available, and to control whether STARTTLS and/or SSL/TLS are negotiated before a detection is run, using the optional field **`"tls_modes"`**. This field, if present, has to be Json array of strings.

Each string has the following format:

```
      prefix ':' suffix
```

The acts as a filter for the detected SSL/TLS state, and the suffix tells the run-time what kind of negotiation should be performed. For each prefix only one string may be listed. The possible values are:

-- **`"plain:plain"`**. If the service supports plain text only (no STARTTLS or SSL/TLS) then run the detection in plain text.

-- **`"tls:tls"`**. If the service runs over SSL/TLS natively on the TCP port (i.e. without using STARTTLS) then negotiate SSL/TLS first and run the detection over SSL/TLS.

-- **`"tls:plain"`**. If the service runs over SSL/TLS natively on the TCP port (i.e. without using STARTTLS), then do not negotiate SSL/TLS first, but run the detection over TCP. In most cases this means that the detection itself needs to send SSL/TLS packets manually.

-- **`"starttls:tls"`**. Same as **`"tls:tls"`**, but for services that use STARTTLS to negotiate SSL/TLS. The run-time will negotiate STARTTLS and then SSL/TLS before passing control to the detection.

-- **`"starttls:plain"`**. Same as **`"tls:plain"`**, but for services that use STARTTLS to negotiate SSL/TLS. This allows access to the service in its cleartext form, without STARTTLS, even if the service supports STARTTLS.

-- **`"starttls:starttls"`**. If the service supports SSL/TLS natively on the TCP port (i.e. without using STARTTLS), then negotiate STARTTLS first, but not SSL/TLS, then run the detection over TCP. In most cases this means that the detection itself needs to send SSL/TLS packets manually.

If one of the supported prefixes does not appear in any of the provided strings then this indicates that the detection should not be run at all for a service with those properties. Some examples:

-- **`tls_modes: ["tls:tls", "starttls:tls"]`**: run the detection only if encryption is available, and use encryption for the detection

-- **`tls_modes: ["plain:plain", "starttls:plain"]`**: run the detection only if an unencrypted service is available, and run it without encryption

-- **`tls_modes: ["tls:plain", "starttls:starttls"]`**: run the detection only if SSL/TLS is available, and then run the detection at the SSL/TLS protocol level, not above it.

TCP dialog detections support five types of transactions: **`"send"`**, **`"receive"`**, **`"reconnect"`**, **`"process"`** and "**`report"`**. **`"send"`** is used to send streaming data to the target, **`"receive"`** is used to wait for data to arrive from the target and receive it, and to continue until a condition has been met. **`"reconnect"`** is used to close the current TCP connection to the target and connect to the same port again, using the new connection for all subsequent transactions. **`"process"`** is used to process the results of the most recent **`"receive"`** transaction. **`"report"`** processing works as described earlier.

**"send"**, **"receive"** and **"reconnect"** transactions set the system variable **"network_status"** to one of **"OK"**, **"Timeout reached"** or "Error xxx" with a Unix error code. In the case of an error a **"error"** return action is generated, unless an **"on_error"** field overrides that return action. The **"on_error"** field has the following format:

-- **"on_error"**: *return action*. This optional field changes the return action in the case of a network error. The default return action is **"error"** with the **"message"** field set to a meaningful value, which results in the detection processing to stop in case of an error. If an error encountered during a particular transaction should not result in detection processing to stop then the return action could, e.g., be set to **"continue"** or **"goto"** with an appropriate target label.

## "send" transactions

**"send"** transactions send data to the target over the current connection. The return action is always **"continue"** if no error occurred during the data transmission.

Supported fields:

**"data"**: *eval-expression*. This is a required field that evaluates to the data to be sent to the target.

**"on_error"**: as described above

## "receive" transactions

**"receive"** transactions wait for data to arrive from the target and, as soon as data is available, process it in a manner similar to how **"process"** transactions work, i.e. data can be matched using different matching rules, or a Lua function can be called for matching. The purpose of that matching process is to determine if sufficient data has arrived from the target to continue the detection. If not then the return action **"wait"** should be used to tell the run-time to stay in the current **"receive"** transaction and wait for more data to arrive. If sufficient data has arrived then a **"continue"** return action should be used to advance to the next transaction. Note that the primary purpose of the matching process in **"receive"** transactions is to determine if sufficient data has arrived. Verifying whether the data matches certain criteria (e.g. to indicate a vulnerability, or make other decisions in the transaction) should usually be done in a separate **"process"** transaction following the **"receive"** transaction.

Two system variables are updated by **"receive"** transactions:

-- **"input"** contains the data received from the target during the current **"receive"** transaction. If **"wait"** return actions are used to stay in the current transaction then additional data is appended to the content of **"input"** every time new data is received, i.e. **"input"** contains all accumulated data received during the current **"receive"** transaction.

-- **"all_input"** contains all data received from the target during the current detection, accumulated over all **"receive"** transactions.

The supported fields are identical to those of the **`"process"`** transaction, with one addition:

-- **`"timeout"`**: Optional field that contains the timeout in milliseconds for the overall transaction, across all **`"wait"`** return actions. This overrides the value set at the detection level.

## "reconnect" transactions

**`"reconnect"`** transactions close the current TCP connection and connect again, to the same port, using the new TCP connection for all subsequent transactions.

Supported fields:

-- **`"timeout"`**: Optional field that contains the timeout in milliseconds for the overall connection process, including StartTLS negotiation and SSL/TLS negotiation, where applicable. This overrides the value set at the detection level.

-- **`"tls_modes"`**: Optional field with the same meaning as described above at the detection level. The default is to inherit the value defined at the detection level. If a different value is specified here then the new connection will use the value specified here.

## "tcp dialog" example 1

This example read the number of recent and total messages from an IMAP account, using fixed credentials. It uses regular matching only, without any Lua functions.

```
{
   "detection_type": "tcp dialog",
   "api_version": 1,
   "trigger_type": "service",
   "services": ["imap", "imaps"],
   "title": "test check 1",
   "dialog": [
     {
       "transaction": "send",
       "data": "a001 LOGIN myuser mypassword\n"
     },
     {
       "transaction": "receive",
       "mode": "luapattern",
       "match": "\na001 [^\n]*\n"
     },
     {
       "transaction": "process",
       "mode": "luapattern",
       "match": "\na001 OK[^\n]*\n"
     },
          {
            "transaction": "send",
            "data": "a002 SELECT INBOX\n"
          },
          {
            "transaction": "receive",
            "mode": "luapattern",
            "match": "\na002 [^\n]*\n"
          },
          {
            "transaction": "process",
            "mode": "luapattern",
            "match": "\na002 OK[^\n]*\n"
          },
          {
            "transaction": "process",
            "mode": "luapattern",
            "match": "\n[*] (%d+) RECENT[^\n]*\n",
            "extract": [{}, {"var": "recent"}]
          },
          {
            "transaction": "process",
            "mode": "luapattern",
            "match": "\n[*] (%d+) EXISTS[^\n]*\n",
            "extract": [{}, {"var": "exists"}]
          },
          {
            "transaction": "report",
            "result": {"concat": ["number of recent messages=",
{"user": "recent"}, "\n", "total number of messages=", {"user":
"exists"}, "\n"]}
          }
        ]
      }
```

## "tcp dialog" example 2

This example tries to find the highest SMB protocol version supported by a target. Since SMB is a binary protocol the example uses Lua functions for packet generation and parsing.

Json portion:

```
{
  "detection_type": "tcp dialog",
  "api_version": 1,
          "trigger_type": "service",
          "services": ["microsoft-ds"],
          "title": "test check 2",
          "dialog": [
            {
              "transaction": "send",
              "data": {"call": {"name":
"qrdiuser_smb_create_v1_negotiate"}}
            },
            {
              "transaction": "receive",
              "mode": "call",
              "name": "qrdiuser_smb_check"
            },
            {
              "transaction": "process",
              "mode": "call",
              "name": "qrdiuser_smb_process_packet"
            },
            {
              "transaction": "send",
              "data": {"call": {"name":
"qrdiuser_smb_create_v2_negotiate"}}
            },
            {
              "transaction": "receive",
              "mode": "call",
              "name": "qrdiuser_smb_check"
            },
            {
              "transaction": "process",
              "mode": "call",
              "name": "qrdiuser_smb_process_packet"
            },
            // never reached via fall through
            {
              "transaction": "report",
              "mode": "luapattern",
              "result": {"user": "result"}
            }
          ]
        }
```

Lua portion (functions that have to be included in the shared Lua function library):

```
function qrdiuser_smb_check(ctx)
  local len, tmp1, tmp2, tmp3;

  if #ctx.system.input < 4 then
    return "wait";
  end
  if 0 ~= string.byte(ctx.system.input, 1) then
    return {action="error", message="Not an SMB packet"}
  end
  tmp1, tmp2, tmp3 = string.byte(ctx.system.input, 2, 4)
  len = (((tmp1 * 256) + tmp2) * 256) + tmp3
  if len < 4  then
    return {action="error", message="Not an SMB packet"}
  end
  if #ctx.system.input < 8 then
    return "wait"
  end
  if string.sub(ctx.system.input, 6, 8) ~= "SMB" then
    return {action="error", message="Not an SMB packet"}
  end
  tmp1 = string.byte(ctx.system.input, 5)
  if tmp1 ~= 254 and tmp1 ~= 255 then
    return {action="error", message="Not an SMB packet"}
  end
  if #ctx.system.input < 4 + len then
    return "wait"
  end
  qrdisystem_unget(ctx, #ctx.system.input - (4 + len))
  ctx.user.packet = string.sub(ctx.system.input, -len)
  return "continue"
end

function qrdiuser_smb_create_packet(ctx, data)
  local len = #data

  return string.char(0, len >> 16 , (len >> 8) % 256, len % 256) ..
data
end

function qrdiuser_smb_create_v1_packet(ctx, hdr, words, bytes)
  local numb, numw, str, _, v

  numb = #bytes
  numw = #words
  str = hdr .. string.char(numw)
  for _, v in ipairs(words) do
    str = str .. string.char(v % 256, v >> 8)
  end
  str = str .. string.char(numb % 256, numb >> 8) .. bytes
  return qrdiuser_smb_create_packet(ctx, str)
end

function qrdiuser_smb_create_v1_negotiate(ctx)
  return qrdiuser_smb_create_v1_packet(ctx,
"\xffSMB\x72\x00\x00\x00\x00\x00\x03\x40\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00", {},
"\x02SMB 2.???\x00\x02SMB 2.002\x00\x02NT LM 0.12\x00")
end

function qrdiuser_smb_create_v2_packet(ctx, cmd, data)
  local msgid

  msgid = ctx.user.smbv2_nextmsgid or 1
  ctx.user.smbv2_nextmsgid = msgid + 1
  return qrdiuser_smb_create_packet(ctx,
"\xfeSMB\x40\x00\x00\x00\x00\x00\x00\x00" .. string.char(cmd %
256, cmd >> 8) .. "\x81\x00\x00\x00\x00\x00\x00\x00\x00\x00" ..
string.char(msgid % 256, msgid >> 8) ..
"\x00\x00\x00\x00\x00\x00\xff\xfe\x00\x00" .. string.rep("\x00",
28) .. data)
end

function qrdiuser_smb_create_v2_negotiate(ctx)
  local data, dialects, dlen, negoff, plen, negdata, neglen,
negcnt, _, v

  negdata =
"\x01\x00\x26\x00\x00\x00\x00\x00\x01\x00\x20\x00\x01\x00" ..

"\x00\x11\x22\x33\x44\x55\x66\x77\x88\x99\xaa\xbb\xcc\xdd\xee\xff"
..

"\x00\x11\x22\x33\x44\x55\x66\x77\x88\x99\xaa\xbb\xcc\xdd\xee\xff"
..
  "\x00\x00" ..
  "\x02\x00\x06\x00\x00\x00\x00\x00\x02\x00\x02\x00\x01\x00"
  neglen = #negdata
  negcnt = 2
  dialects = {0x0311, 0x0302, 0x0300, 0x0210, 0x0202}
  dlen = #dialects
  data = "\x24\x00" .. string.char(dlen % 256, dlen >> 8) ..
"\x01\x00\x00\x00\x44\x00\x00\x00\x01\x23\x45\x67\x89\xab\xcd\xef\
x01\x23\x45\x67\x89\xab\xcd\xef"
  negoff = #data + 8 + (2 * dlen) + 7
  negoff = negoff - (negoff % 8)
  plen = negoff - (#data + 8 + (2 * dlen))
  negoff = negoff + 64
  data = data .. string.char(negoff % 256, negoff >> 8, 0, 0,
negcnt % 256, negcnt >> 8, 0, 0)
  for _, v in ipairs(dialects) do
    data = data .. string.char(v % 256, v >> 8)
  end
  data = data .. string.rep("\x00", plen) .. negdata
  return qrdiuser_smb_create_v2_packet(ctx, 0, data)
end

function qrdiuser_smb_process_v1_packet(ctx)
  local tmp

  if #ctx.user.packet < 33 then
    return {action="error", message="Not an SMBv1 packet"}
  end
  tmp = string.byte(ctx.user.packet, 10)
  if (tmp & 0x80) ~= 0x80 then
    return {action="error", message="Stream loopback"}
  end
  tmp = string.byte(ctx.user.packet, 5)
  if tmp ~= 0x72 then
    return {action="error", message="Unexpected SMBv1 packet"}
  end
  if string.sub(ctx.user.packet, 6, 9) ~= "\x00\x00\x00\x00" then
    return {action="error", message="SMBv1 negotiation failed"}
  end
  -- No need to parse further. The only SMBv1 dialect we offer is
"NT LM 0.12".
  -- All other dialects would get reported in an SMBv2 packet
  ctx.user.result = "Negotiated SMBv1 NT LM 0.12"
  return "report"
end

function qrdiuser_smb_process_v2_packet(ctx)
  local tmp1, tmp2, ver

  if #ctx.user.packet < 64 then
    return {action="error", message="Not an SMBv2 packet"}
  end
  tmp1 = string.byte(ctx.user.packet, 17)
  if (tmp1 & 1) ~= 1 then
    return {action="error", message="Stream loopback"}
  end
  tmp1, tmp2 = string.byte(ctx.user.packet, 13, 14)
  if tmp1 + (256 * tmp2) ~= 0 then
    return {action="error", message="Unexpected SMBv2 packet"}
  end
  if string.sub(ctx.user.packet, 9, 12) ~= "\x00\x00\x00\x00" then
    return {action="error", message="SMBv2 negotiation failed"}
  end
  if #ctx.user.packet < 72 then
    return {action="error", message="Invalid SMBv2 negotiate
response"}
  end
  tmp1, tmp2 = string.byte(ctx.user.packet, 69, 70)
  ver = tmp1 + (256 * tmp2)
  if ver == 0x02ff then
    if ctx.user.v2sent then
      return {action="error", message="SMBv2 negotiation is
looping"}
    else
      ctx.user.v2sent = true
      return "continue"
    end
  elseif ver == 0x0202 then
    ctx.user.result = "Negotiated SMBv2.0.2"
  elseif ver == 0x0210 then
    ctx.user.result = "Negotiated SMBv2.1"
  elseif ver == 0x0300 then
    ctx.user.result = "Negotiated SMBv3.0"
  elseif ver == 0x0302 then
    ctx.user.result = "Negotiated SMBv3.0.2"
  elseif ver == 0x0311 then
    ctx.user.result = "Negotiated SMBv3.1.1"
  else
    return {action="error", message="Invalid SMBv2 dialect"}
  end
  return "report"
end

function qrdiuser_smb_process_packet(ctx)
  local type = string.byte(ctx.user.packet, 1)

  if type == 254 then
    return qrdiuser_smb_process_v2_packet(ctx)
  else
    return qrdiuser_smb_process_v1_packet(ctx)
  end
end
```

# LUA scripting

QRDI supports the concept of a library of Lua functions shared by all detections which can be used to implement complex data generation or parsing rules, in particular for binary protocols, for which substrings and regular expressions do not provide enough functionality. Lua functions can be used for two purposes:

-- In an eval_expression, to dynamically calculate a data item. In that case the Lua function needs to return the data value.

-- In a **`"process"`** or similar transaction (e.g. **`"receive"`**), to process incoming data. In that case the Lua function needs to return a return action.

Lua functions called by the run-time are always called with at least one argument: a table representing a context that can be used to access user variables and system variables. Additional arguments can be specified by the transaction.

We currently use Lua version 5.3, with minor modifications made to the interpreter as described below in the section “The Lua run-time environment”.

## Shared Lua function library

The shared function library consists of a series of Lua function definitions, with each definition having the following format:

```
function qrdiuser_my_function_1(ctx, additional_args)
  lua_function_body
end
```

The names of all functions defined in the library must start with the prefix **`"qrdiuser_"`**.

Every function that should be callable directly from a detection must have at least one argument to accept the current context.

Additional arguments are permitted. Functions that only have to be callable by other Lua functions have no restrictions on arguments.

Only function definitions formatted as shown above are permitted in the shared Lua function library. No other Lua code is permitted, in particular no global variables or Lua code outside of a function definition is permitted.

## Lua run-time environment

The **`ctx`** argument passed to each function is a Lua table that contains two fields: **`ctx.user`** and **`ctx.system`**. Those fields are, themselves, Lua tables and contain all user and system variables, respectively. The **`ctx`** base table and **`ctx.system`** table are read-only and may not be modified. The **`ctx.user`** table may be freely modified.

When data needs to be exchanged between Json and Lua, types are mapped in a natural manner: all simple types in one environment map to the corresponding simple type in the other environment. Json arrays map to Lua tables indexed by integers. Json objects map to Lua tables indexed by strings.

User-defined Lua functions run inside of a sandbox environment that is more restrictive than regular Lua, in order to ensure that detections can execute within reasonable time and memory constraints, and in order to minimize the risk of future Lua version upgrades causing incompatibilities.

Only a subset of Lua globals is accessible from user-defined Lua functions, and those globals may only be read (or, in the case of functions, called). Globals may not be written to, no new globals may be created, and no globals other than those listed below may be accessed at all. If a set of functions wants to preserve state across function calls within a single detection then user variables (in **`ctx.user`**) should be used, not globals. The following globals are readable/callable:

-- **`assert`**

-- **`error`**

-- **`ipairs`**

-- **`math`** (all members)

-- **`next`**

-- **`pairs`**

-- **`select`**

-- **`string`** (all members except for **`string.dump`**)

-- **`table`** (all members)

-- **`tonumber`**

-- **`tostring`**

-- **`type`**

-- **`utf8`** (all members)

-- **`_VERSION`**

In addition all user-defined functions (i.e. functions starting with **`"qrdiuser_"`**) and all functions provided by the QRDI run-time (functions starting with **`"qrdisystem_"`**) are callable.

The following run-time resource limits apply:

-- The size of the Lua function library is limited to 1 MB (source code length)

-- The total amount of memory allocated by the Lua interpreter during the execution of detections (for any purpose, i.e. for code, data etc.) is limited to 4 MB.

-- The total number of Lua opcodes (which roughly map to Lua instructions) that may be executed by a single detection is limited to 100,000. The purpose of this limit is to encourage good programming practices that avoid using excessive CPU resources. For example, Lua **`string.*`** functions should be used whenever possible, instead of manually iterating over bytes in a buffer. Also, very complex operations, such as compression, encryption or hashing, may not be implemented in Lua, where they would use excessive CPU resources. If such functionality is needed then feature requests should be filed so the functionality can be provided as part of the QRDI run-time environment.

The above restrictions and limits are enforced by the QRDI run-time, and violations usually result in run-time errors that cause the offending detection to abort with an error.

## Lua system functions

The QRDI run-time exports several Lua functions that can be called from user-defined Lua functions. The name of all of those functions starts with **`"qrdisystem_"`**.

### qrdisystem_log

**`qrdisystem_log(ctx, log_level, … )`**

This function is available for all detection types.

**`"qrdisystem_log"`** is used to generate a log message that is added to the **`/INFO/DEBUGDATA`** section of the QID output as described in the section “Output format” above.

**`"ctx"`** is the context which all Lua functions receive as their first argument.

**`"log_level"`** is an integer indicating the log (debug) level. A message appears in the debug log if **`"log_level"`** is less than or equal to the value of **`"debug_level"`**, as explained in “Output format”. Arguments following **`"log_level"`** in the function call are converted to strings and concatenated to form the message to be logged. Only simple types are permitted, i.e. numbers, strings, booleans and nil.

### qrdisystem_unget

**`qrdisystem_unget(ctx, num_bytes)`**

This function is available for detections of type **`"tcp_dialog"`**.

**`"qrdisystem_unget"`** removes **`"num_bytes"`** from the end of the current input buffer, i.e. from the system variable "input", and appends them to the end of an internal unget buffer.

**`"ctx"`** is the context which all Lua functions receive as their first argument. The function is intended to be used by Lua functions called during **`"receive"`** transactions to indicate how many bytes at the end of the input buffer are not part of the current protocol message and therefore should not be processed in subsequent transactions. Any bytes removed from the input buffer in this manner will automatically be inserted at the beginning of the input buffer during the next **`"receive"`** transaction, and that transaction will then immediately trigger its matching logic, without blocking first.

Example:

-- A **"receive"** transaction starts and blocks, the receives 200 bytes in its first TCP packet. It unblocks and calls its Lua matching function.

-- The matching function determines that only 128 bytes of those 200 bytes belong to the first protocol message and should be parsed. Before returning its **"continue"** return action the Lua matching function calls **qrdisystem_unget(ctx, 72)** to postpone the excess 72 bytes.

-- The input buffer now contains only the first 128 bytes, ready to be processed by subsequent **"process"** transactions.

-- When the next **"receive"** transaction starts, the excess 72 bytes are placed at the beginning of the input buffer, and the Lua matching function is called immediately. It can now accept the data with a **"continue"** return action (and, again, call **"qrdisystem_unget"** if there are still too many bytes in the buffer), or wait for more data with a **"wait"** return action.

## qrdisystem_is_tcp_port_open

**qrdisystem_is_tcp_port_open(ctx, port_num)**

This function is available for detections of type **"tcp_dialog"**.

**"qrdisystem_is_tcp_port_open"** checks whether the TCP port indicates by **"port_num"** is open on the target.

**"ctx"** is the context which all Lua functions receive as their first argument. The return value boolean indicating whether the port is open. For a port to be detected as open the port must have been included in the list of TCP ports to be scanned in the current scan. Port numbers not included in that list will usually be reported as closed, even if they are open on the target.

# Qualys API support

Qualys APIs support QRDI vulnerabilities like Qualys provided ones. We've made no changes to DTDs for XML output. Here's some sample API calls you can make to manage QRDI vulnerabilities in your account.

API v2 support | API v1 support

## API v2 support

API v2 calls support scanning and reporting on QRDI vulnerabilities. Several examples provided below.

### Launch scan on QRDI vulnerability checks

Launch scan using Scan API v2. The option profile "Initial Options" is configured with all vulnerability checks in the user's account including QRDI vulnerabilities and Qualys provided ones.

API request:

```
curl -u "USERNAME:PASSWORD" -H "X-Requested-With: Curl" -X "POST"
-d
"action=launch&exclude_ip_per_scan=64.39.96.0-64.39.111.255&ip=10.
10.24.35,10.10.25.71,10.10.25.80,10.10.25.82,10.10.25.163,10.10.25
.165,10.10.25.238,10.10.26.98&scan_title=VM_SCAN_API_1500217432&op
tion_title=Initial Options"
"https://qualysapi.qualys.com/api/2.0/fo/scan/"
```

XML output:

See Scan Results (XML format)

### Launch host based asset report

Launch asset data report (i.e. host based report) using Report API v2. In sample below the IP 10.20.31.111 has QRDI vulnerabilities detected on it and the sample report shows detection instance (VULN_INFO tag).

API request:

```
curl -u "USERNAME:PASSWORD" -H "X-Requested-With: Curl" -X "POST"
-d
"action=launch&report_type=Scan&output_format=html&template_id=420
6744&report_title=APILaunchReport_SCAN&ips=10.20.31.111"
"https://qualysapi.qualys.com/api/2.0/fo/report/"
```

<u>XML output:</u>

Simple return as shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE GENERIC SYSTEM
"https://qualysapi.qualys.com/api/2.0/simple_return.dtd">
<SIMPLE_RETURN>
  <RESPONSE>
    <DATETIME>2017-08-02T21:45:23Z</DATETIME>
    <TEXT>New report launched</TEXT>
    <ITEM_LIST>
      <ITEM>
        <KEY>ID</KEY>
        <VALUE>1665</VALUE>
      </ITEM>
    </ITEM_LIST>
  </RESPONSE>
</SIMPLE_RETURN>
```

<u>Sample report:</u>

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE ASSET_DATA_REPORT SYSTEM
"https://qualysguard.qualys.com/asset_data_report.dtd">
<ASSET_DATA_REPORT>
  <HEADER>
    <COMPANY><![CDATA[Acme]]></COMPANY>
    <USERNAME>acme_rm</USERNAME>
    <GENERATION_DATETIME>2017-08-
04T09:54:22Z</GENERATION_DATETIME>
    <TEMPLATE><![CDATA[Host-based-Report007]]></TEMPLATE>
    <TARGET>
      <USER_ASSET_GROUPS>
        <ASSET_GROUP_TITLE><![CDATA[QRDI-AG]]></ASSET_GROUP_TITLE>
      </USER_ASSET_GROUPS>
      <COMBINED_IP_LIST>
        <RANGE network_id="2002">
          <START>10.20.31.111</START>
          <END>10.20.31.111</END>
        </RANGE>
      </COMBINED_IP_LIST>
...
  <HOST_LIST>
    <HOST>
      <IP network_id="2002">10.20.31.111</IP>
```

```
            <TRACKING_METHOD>IP</TRACKING_METHOD>
      ...
            <VULN_INFO_LIST>
              <VULN_INFO>
                <QID id="qid_410002">410002</QID>
                <TYPE>Vuln</TYPE>
                <PORT>1443</PORT>
                <PROTOCOL>tcp</PROTOCOL>
                <SSL>true</SSL>
                <RESULT><![CDATA[Regex Pattern found on page]]></RESULT>
                <FIRST_FOUND>2017-08-01T09:26:55Z</FIRST_FOUND>
                <LAST_FOUND>2017-08-03T09:05:08Z</LAST_FOUND>
                <TIMES_FOUND>4</TIMES_FOUND>
                <VULN_STATUS>Active</VULN_STATUS>
                <CVSS_FINAL>-</CVSS_FINAL>
                <CVSS3_FINAL>-</CVSS3_FINAL>
                <TICKET_NUMBER>3991</TICKET_NUMBER>
                <TICKET_STATE>OPEN</TICKET_STATE>
              </VULN_INFO>
      ...
```

## Launch scan based asset report

Launch scan based report using Report API v2. In this sample, scan reference `scan/1500204665.47146` performed QRDI vulnerability checks. QRDI vulnerability detections, if any, will be included in the downloaded report.

API request:

```
curl -u "USERNAME:PASSWORD" -H "X-Requested-With: Curl" -X "POST"
-d
"action=launch&report_type=Scan&output_format=xml&template_id=4209
732&report_title=APILaunchReport_SCAN&report_refs=scan/1500204665.
47146&ip_restriction=10.10.25.169&use_tags=1&tag_include_selector=
all&tag_set_by=name&tag_set_include=AG_IP_Multi_SA_1486742574"
"https://qualysapi.qualys.com/api/2.0/fo/report/"
```

XML output:

See Simple Return

Sample report:

See Scan Results (XML format)

## Download asset search report

Download asset search report for assets with QRDI vulnerability detections using Asset Search API v2. This sample report shows QRDI vulnerability 410002 was detected on the asset with IP address 10.20.31.111.

API request:

```
curl -u "USERNAME:PASSWORD" -H "X-Requested-With: Curl" -X "POST"
-d "action=search&output_format=xml&ips=10.20.31.111&qids=410002"
"https://qualysapi.qualys.com/api/2.0/fo/report/asset/"
```

XML output:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ASSET_SEARCH_REPORT SYSTEM
"https://qualysguard.qualys.com/asset_search_report_v2.dtd">

<ASSET_SEARCH_REPORT>
<HEADER>
  <COMPANY><![CDATA[My Corp]]></COMPANY>
  <USERNAME>corp_ab1</USERNAME>
  <GENERATION_DATETIME>2017-08-08T06:25:52Z</GENERATION_DATETIME>
  <TOTAL>1</TOTAL>
  <FILTERS>
    <IP_LIST>
      <RANGE>
        <START>10.20.31.111</START>
        <END>10.20.31.111</END>
      </RANGE>
    </IP_LIST>
    <FILTER_QID><![CDATA[410002]]></FILTER_QID>
  </FILTERS>
</HEADER>
<HOST_LIST>
  <HOST>
    <IP><![CDATA[10.20.31.111]]></IP>
    <TRACKING_METHOD>IP address</TRACKING_METHOD>
    <DNS><![CDATA[ubu-31-111.ml2k8.abc.mycorp.com]]></DNS>
    <OPERATING_SYSTEM><![CDATA[Ubuntu / Fedora / Tiny Core Linux /
Linux 3.x]]></OPERATING_SYSTEM>
    <QID_LIST>
      <QID>
        <ID><![CDATA[410002]]></ID>
      </QID>
    </QID_LIST>
    <NETWORK><![CDATA[My Custom Network]]></NETWORK>
    <LAST_SCAN_DATE>2017-08-07T10:00:38Z</LAST_SCAN_DATE>
    <FIRST_FOUND_DATE>2017-08-01T09:30:24Z</FIRST_FOUND_DATE>
  </HOST>
</HOST_LIST>
</ASSET_SEARCH_REPORT>
```

## Download Knowledgebase QIDs

Download QRDI vulnerabilities from the KnowledgeBase using KnowledgeBase APIv2. This sample shows details for the QRDI vulnerability with QID 410010.

API request:

```
curl -u "USERNAME:PASSWORD" -H "X-Requested-With: Curl" -X "POST"
-d
"action=list&details=All&show_disabled_flag=1&show_vuln_detection_
info=1&ids=410010"
"https://qualysapi.qualys.com/api/2.0/fo/knowledge_base/vuln/"
```

XML output:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE KNOWLEDGE_BASE_VULN_LIST_OUTPUT SYSTEM
"https://qualysapi.qualys.com/api/2.0/fo/knowledge_base/vuln/knowl
edge_base_vuln_list_output.dtd">
<KNOWLEDGE_BASE_VULN_LIST_OUTPUT>
  <RESPONSE>
    <DATETIME>2017-08-08T06:26:29Z</DATETIME>
    <VULN_LIST>
      <VULN>
        <QID>410010</QID>
        <VULN_TYPE>Information Gathered</VULN_TYPE>
        <SEVERITY_LEVEL>1</SEVERITY_LEVEL>
        <TITLE><![CDATA[qrdi http 10]]></TITLE>
        <CATEGORY>QRDI</CATEGORY>
        <LAST_CUSTOMIZATION>
          <DATETIME>2017-07-26T09:14:58Z</DATETIME>
          <USER_LOGIN>corp_ab1</USER_LOGIN>
        </LAST_CUSTOMIZATION>
        <LAST_SERVICE_MODIFICATION_DATETIME>2017-07-
21T09:15:09Z</LAST_SERVICE_MODIFICATION_DATETIME>
        <PUBLISHED_DATETIME>2017-07-
21T09:15:09Z</PUBLISHED_DATETIME>
        <PATCHABLE>0</PATCHABLE>
        <VENDOR_REFERENCE_LIST>
          <VENDOR_REFERENCE>
            <ID><![CDATA[ven<img src=z>dor-2]]></ID>
            <URL><![CDATA[https://vendor.com]]></URL>
          </VENDOR_REFERENCE>
        </VENDOR_REFERENCE_LIST>
        <CVE_LIST>
          <CVE>
            <ID><![CDATA[CVE-2017-1111]]></ID>
            <URL><![CDATA[http://cve.mitre.org/cgi-
bin/cvename.cgi?name=CVE-2017-1111]]></URL>
          </CVE>
        </CVE_LIST>
        <DIAGNOSIS><![CDATA[custom threat
description]]></DIAGNOSIS>
        <CONSEQUENCE><![CDATA[custom impact
description]]></CONSEQUENCE>
        <SOLUTION><![CDATA[custom solution
description]]></SOLUTION>
        <PCI_FLAG>0</PCI_FLAG>
        <DISCOVERY>
          <REMOTE>0</REMOTE>
        </DISCOVERY>
        <IS_DISABLED>0</IS_DISABLED>
      </VULN>
    </VULN_LIST>
  </RESPONSE>
</KNOWLEDGE_BASE_VULN_LIST_OUTPUT
```

## Create dynamic search list

Create a dynamic search list for all your QRDI vulnerabilities (i.e. **`categories=QRDI`**) using Search List API.

API request:

```
curl -u "USERNAME:PASSWORD" -H "X-Requested-With: Curl" -X "POST"
-d "action=create&title=QRDI-Search_List&global=0&comments=Search
List Created BY API&categories=QRDI"
"https://qualysapi.qualys.com/api/2.0/fo/qid/search_list/dynamic/"
```

XML output:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE SIMPLE_RETURN SYSTEM
"https://qualysapi.qualys.com/api/2.0/simple_return.dtd">
<SIMPLE_RETURN>
  <RESPONSE>
    <DATETIME>2017-08-08T06:31:46Z</DATETIME>
    <TEXT>New search list created successfully</TEXT>
    <ITEM_LIST>
```

```
      <ITEM>
        <KEY>ID</KEY>
        <VALUE>119176</VALUE>
      </ITEM>
    </ITEM_LIST>
  </RESPONSE>
</SIMPLE_RETURN>
```

## Download host VM detection data

Using Host VM Detection API v2, VM detection records will list QRDI vulnerabilities for selected assets. No debug info, i.e. logging data or errors, will be shown.

API request:

```
curl -u "USERNAME:PASSWORD" -H "X-Requested-With: Curl" -X "POST"
-d "action=list&ips=10.20.31.111&show_igs=1&qids=410010"
"https://qualysapi.qualys.com/api/2.0/fo/asset/host/vm/detection/"
```

XML output:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE HOST_LIST_VM_DETECTION_OUTPUT SYSTEM
"https://qualysapi.qualys.com/api/2.0/fo/asset/host/vm/detection/h
ost_list_vm_detection_output.dtd">
<HOST_LIST_VM_DETECTION_OUTPUT>
  <RESPONSE>
    <DATETIME>2017-08-08T06:47:52Z</DATETIME>
    <HOST_LIST>
      <HOST>
        <ID>342802</ID>
        <IP>10.20.31.111</IP>
        <TRACKING_METHOD>IP</TRACKING_METHOD>
        <NETWORK_ID>2002</NETWORK_ID>
        <OS><![CDATA[Ubuntu / Fedora / Tiny Core Linux / Linux
3.x]]></OS>
        <DNS><![CDATA[ubu-31-111.ml2k8.abc.corp.com]]></DNS>
        <LAST_SCAN_DATETIME>2017-08-
07T10:02:01Z</LAST_SCAN_DATETIME>
        <LAST_VM_SCANNED_DATE>2017-08-
07T10:00:38Z</LAST_VM_SCANNED_DATE>
        <LAST_VM_SCANNED_DURATION>487</LAST_VM_SCANNED_DURATION>
        <DETECTION_LIST>
          <DETECTION>
            <QID>410010</QID>
            <TYPE>Info</TYPE>
            <PORT>8080</PORT>
            <PROTOCOL>tcp</PROTOCOL>
            <RESULTS><![CDATA[Regex Pattern found on
page]]></RESULTS>
            <LAST_PROCESSED_DATETIME>2017-08-
07T10:02:01Z</LAST_PROCESSED_DATETIME>
          </DETECTION>
        </DETECTION_LIST>
      </HOST>
    </HOST_LIST>
  </RESPONSE>
</HOST_LIST_VM_DETECTION_OUTPUT>
```

## Export/Import Option Profile

The Option Profile API allows customers to export/import option profiles from one subscription to another in XML format. The API user must have the Manager role.

### Export Option Profile

You'll see <QRDI_CHECKS>1</QRDI_CHECKS> in the XML when the option "All QRDI checks" is selected in the option profile.

API request:

```
curl -u "USERNAME:PASSWORD" -H "X-Requested-With:curl"
-X GET "action=export"
"https://qualysapi.qualys.com/api/2.0/fo/subscription/option_profi
le/"
```

All the option profiles in the user's account get exported in XML format.

XML output:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE OPTION_PROFILES SYSTEM
"https://qualysapi.qualys.com/api/2.0/fo/subscription/option_profi
le/option_profile_info.dtd">
<OPTION_PROFILES>
  <OPTION_PROFILE>
    <BASIC_INFO>
      <ID>111186</ID>
      <GROUP_NAME><![CDATA[OP-SCAN]]></GROUP_NAME>
      <GROUP_TYPE>user</GROUP_TYPE>
      <USER_ID><![CDATA[John Doe(john_doe)]]></USER_ID>
      <UNIT_ID>0</UNIT_ID>
      <SUBSCRIPTION_ID>44</SUBSCRIPTION_ID>
      <IS_DEFAULT>0</IS_DEFAULT>
      <IS_GLOBAL>1</IS_GLOBAL>
      <IS_OFFLINE_SYNCABLE>0</IS_OFFLINE_SYNCABLE>
      <UPDATE_DATE>N/A</UPDATE_DATE>
    </BASIC_INFO>
    <SCAN>
      <PORTS>
        <TCP_PORTS>
          <TCP_PORTS_TYPE>full</TCP_PORTS_TYPE>
          <THREE_WAY_HANDSHAKE>1</THREE_WAY_HANDSHAKE>
        </TCP_PORTS>
...
      <VULNERABILITY_DETECTION>
        <CUSTOM_LIST>
          <CUSTOM>
            <ID>2096</ID>
            <TITLE><![CDATA[Scan Report Template: High Severity
Report]]></TITLE>
          </CUSTOM>
          <CUSTOM>
            <ID>87938</ID>
            <TITLE><![CDATA[Windows Authentication Results
v.1]]></TITLE>
          </CUSTOM>
          <CUSTOM>
            <ID>87939</ID>
            <TITLE><![CDATA[Unix Authentication Results
v.1]]></TITLE>
          </CUSTOM>
        </CUSTOM_LIST>
        <DETECTION_INCLUDE>
          <BASIC_HOST_INFO_CHECKS>1</BASIC_HOST_INFO_CHECKS>
          <OVAL_CHECKS>1</OVAL_CHECKS>
          <QRDI_CHECKS>1</QRDI_CHECKS>
        </DETECTION_INCLUDE>
...
```

### Import Option Profile

Include <QRDI_CHECKS>1</QRDI_CHECKS> in the option profile XML to enable the option "All QRDI checks" in the profile.

When calling the Import Option Profile API the user needs to pass the proper XML with Content-Type XML. This will create option profiles in that user's subscription. All validations are applied as in the Qualys portal UI while creating option profiles using the Import Option Profile API. Complete validation details are provided in the Qualys API V2 User Guide.

<u>API request:</u>

```
curl -u "USERNAME:PASSWORD" -H "content-type: text/xml"-X "POST"
--data-binary @Export_OP.xml
"https://qualysapi.qualys.com/api/2.0/fo/subscription/option_profi
le/?action=import"
```

Note: “Export_OP.xml” contains the request POST data.

<u>Request POST data:</u>

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OPTION_PROFILES SYSTEM
"https://qualysapi.qualys.com/api/2.0/fo/subscription/option_profi
le/option_profile_info.dtd">
<OPTION_PROFILES>
  <OPTION_PROFILE>
    <BASIC_INFO>
      <ID>11123</ID>
      <GROUP_NAME><![CDATA[OP-SCAN]]></GROUP_NAME>
      <GROUP_TYPE>user</GROUP_TYPE>
      <USER_ID><![CDATA[John Doe (john_doe)]]></USER_ID>
      <UNIT_ID>0</UNIT_ID>
      <SUBSCRIPTION_ID>76084</SUBSCRIPTION_ID>
      <IS_DEFAULT>0</IS_DEFAULT>
      <IS_GLOBAL>1</IS_GLOBAL>
      <IS_OFFLINE_SYNCABLE>0</IS_OFFLINE_SYNCABLE>
      <UPDATE_DATE>N/A</UPDATE_DATE>
    </BASIC_INFO>
    <SCAN>
      <PORTS>
        <TCP_PORTS>
          <TCP_PORTS_TYPE>full</TCP_PORTS_TYPE>
          <THREE_WAY_HANDSHAKE>1</THREE_WAY_HANDSHAKE>
        </TCP_PORTS>
...
      <VULNERABILITY_DETECTION>
        <CUSTOM_LIST>
          <CUSTOM>
            <ID>2096</ID>
            <TITLE><![CDATA[Scan Report Template: High Severity
Report]]></TITLE>
          </CUSTOM>
          <CUSTOM>
            <ID>87938</ID>
            <TITLE><![CDATA[Windows Authentication Results
v.1]]></TITLE>
```

```
          </CUSTOM>
          <CUSTOM>
            <ID>87939</ID>
            <TITLE><![CDATA[Unix Authentication Results
v.1]]></TITLE>
          </CUSTOM>
        </CUSTOM_LIST>
        <DETECTION_INCLUDE>
          <BASIC_HOST_INFO_CHECKS>1</BASIC_HOST_INFO_CHECKS>
          <OVAL_CHECKS>1</OVAL_CHECKS>
          <QRDI_CHECKS>1</QRDI_CHECKS>
        </DETECTION_INCLUDE>
...
```

XML Response:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE SIMPLE_RETURN SYSTEM
"https://qualysapi.qualys.com/api/2.0/simple_return.dtd">
<SIMPLE_RETURN>
 <RESPONSE>
   <DATETIME>2017-04-03T11:17:43Z</DATETIME>
   <TEXT>Successfully imported Option profile for the subscription
Id 76084</TEXT>
   <ITEM_LIST>
     <ITEM>
       <KEY>111234</KEY>
       <VALUE>My-OP</VALUE>
     </ITEM>
   </ITEM_LIST>
 </RESPONSE>
</SIMPLE_RETURN>
```

## API v1 support

API v1 calls support scanning and reporting on QRDI vulnerabilities. Several examples provided below.

### Launch scan v1 (/msp/scan.php)

Launches scan that performs QRDI vulnerability checks when defined in scan's option profile.

### Configure scheduled scan v1 (/msp/scheduled_scans.php)

Defines scheduled scan that performs QRDI vulnerability checks when defined in scheduled scan's option profile.

### Download scan report v1 (/msp/scan_report.php)

Downloads scan report containing QRDI vulnerabilities when detected by the scan.

### Download asset search report v1 (/msp/asset_search.php)

Downloads asset search report containing QRDI vulnerabilities when detected on target assets.

### Download ticket list v1 (/msp/ticket_list.php)

Downloads remediation ticket list containing QRDI vulnerabilities when detected on target assets.

### Download KnowledgeBase list v1 (/msp/knowledgebase_download.php)

Downloads complete list of QIDs in the KnowledgeBase including QRDI vulnerabilities when added to your subscription.