

GTTA Admin Script Integration Help

General

GTTA allows you to run some automated checks using scripts hosted on the system. These checks run in background mode, in a separate process, without affecting other automated checks or GUI. Automated checks are integrated with the main system using GTTA Script Interface (GTTA SI). In order to make an automated GTTA check, the script should meet certain GTTA SI requirements and should be registered in the GUI.

It's allowed to use one of the following interpreted languages to implement an automated check:

- Python 2.5+ (preferred)
- Perl 5.8+

This manual assumes that GTTA is installed into the `/opt/gtta/` directory of the server and all paths in this document are assumed to be relative to that directory (unless otherwise specified). Additional Tools (Metasploit, backtrack etc.) can be integrated in release 2.0 as well.

Script Interface

File Locations and Naming

- All scripts should be placed under `/opt/gtta/scripts/` directory.
- The script name should be written in lower-case letters, words should be separated by underscores (for example, `scan_this.py`).
- Python scripts should have a `py` file extension (lower-case).
- Perl scripts should have a `pl` file extension (lower-case).
- If a script needs some additional custom modules, they can be placed under the following directories:
 - `scripts/pythonlib/` - for Python modules and libraries
 - `scripts/perlbin/` - for Perl modules and libraries
- If a script needs some additional files, it's allowed to create a custom sub-directory inside `scripts/` directory named just like the script name itself, without the extension.

For example, if your script is named as `scan_this.py`, the directory should be named as `scripts/scan_this/`.

Command Line Arguments

Scripts are launched in the background mode with certain command line arguments. Every command line argument passed to GTTA script is a path to a file, which contains some input data. An input file is a sequence of lines separated by the *Line Feed* symbols (LF, 0x0A or \n) and every single line is assumed to be a separate input record.

First two command line arguments (i.e. input files) are mandatory and are passed for every GTTA script:

1. Path to a target file, which describes a target system that should be checked and some basic check settings
2. Path to an output file that should be used to write the script output

Others command line arguments are completely optional and depend on the particular check requirements.

Target File Structure

Here is a list of lines, that describe the target:

1. IP Address (Host Name) – target's network address – either IP, or FQDN
2. Protocol – a name of the protocol, that should be used during this check. For example, if a script performs some web tests, this field may have one of these values: *http* or *https*. Of course, the script can just silently ignore this option.
3. Port Number – a target's port number to connect. This option could be ignored.
4. Language – user selected language code – English or German (en, de). Currently all scripts ignore this option, but if your script needs to output different information depending on the selected language – this option might be useful.

Result File Structure

Result file is a sequence of lines, separated by *Line Feed* symbols. Result file can contain several control sequences, that will be processed by the GUI to be able to display the information for the user in more convenient way. The control sequence is a simple XML text with special tags.

Result Tables

This control sequence allows displaying tables tied to the check. The control sequence looks like this:

```
<gtta-table>
  <columns>
    <column width="0.4">Column Title</column>
    <column width="0.6">2nd Column Title</column>
  </columns>
  <row>
```

```
<cell>Some Data</cell>

<cell>88</cell>

</row>

<row>

  <cell>Some Other Data</cell>

  <cell>99</cell>

</row>

</gtta-table>
```

This example contains several required control tags:

- Tag "columns" – contains table column definitions
- Tag "column" – defines a name for a single column. Attribute named "width" contains a width definition for the column. Width is a float number ≤ 1.0 (1.0 is a full table width).
- Tag "row" – contains a list of cells for a row.
- Tag "cell" – defines a cell content.

Please note that script can return only 1 table per launch.

Result Attachments

This control sequence allows adding attachments to the check object. The control sequence looks like this:

```
<gtta-image src="/path/to/file.png"></gtta-image>
```

Attribute "src" contains a local path to the file to be attached, so the check script is totally responsible for downloading that file to the local host. After attachment is added to a project, the system automatically deletes the source file.

The system allows to attach any kind of files, the main tag is called "gtta-image" because mostly this mechanism will be used for image attachments.

Python Library

There is a standard GTTA library under `scripts/pythonlib/gtta/` directory – it helps to create automated scripts in Python. The library implements some common tasks, such

as parsing command line arguments, reading input files and writing to the output file. All Python checks should use this library.

In order to use the library, a script should import *gtta* library into its namespace. The script should declare some class inherited from the base class *gtta.Task*. The ancestor class should implement a function named *main* – this function should do the script's job. Function *main* will be called just after library parsed all command line arguments. The list of function's parameters depends on additional arguments that have been passed to the script. For example, if the script has been called like this:

```
script.py target.txt result.txt data1.txt data2.txt,
```

then *main* function will be called with 2 parameters (except *self* class reference) – the first parameter will contain a list of lines in *data1.txt* file, the second will contain a list of lines in *data2.txt* file. If input files are empty, then empty lists will be passed. For example, if the first file contains 2 lines and the second one is empty, a function call may look like this:

```
main([ 'first line', 'second line' ], [])
```

Script class inherits some attributes, that give access to data in the target file:

- *self.host* – domain name (if it was specified on the first line of the target file)
- *self.ip* – IP-address (if target file contained one). Only host name or IP address, but not both can be specified at the same time, since there is only 1 field in the target file that contains host address data.
- *self.proto* – protocol name
- *self.port* – port number
- *self.lang* – language code

The script should write a result data using *self._write_result* method – one method call will produce a single line in the output file.

Before and after doing time-consuming or blocking operations it's required to call *self._check_stop* method – it checks if the script should stop execution and exit. Scripts can be terminated by user request from GUI, so it's required to perform that kind of check.

It's required to implement error checks and try-catch blocks to handle all possible errors. If any error occurs during the execution, the script should show some information about that issue.

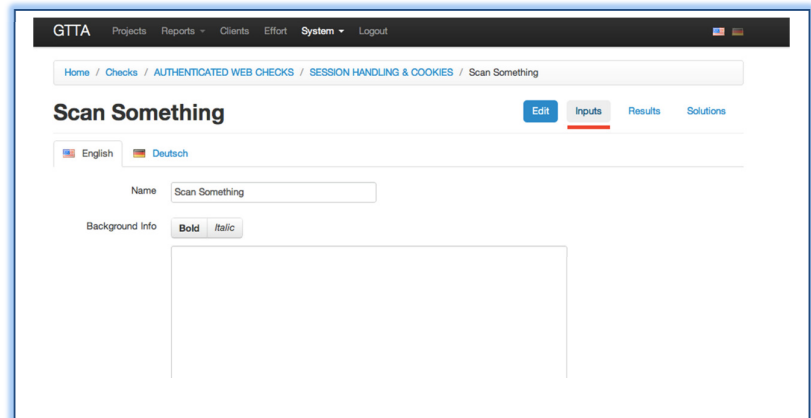
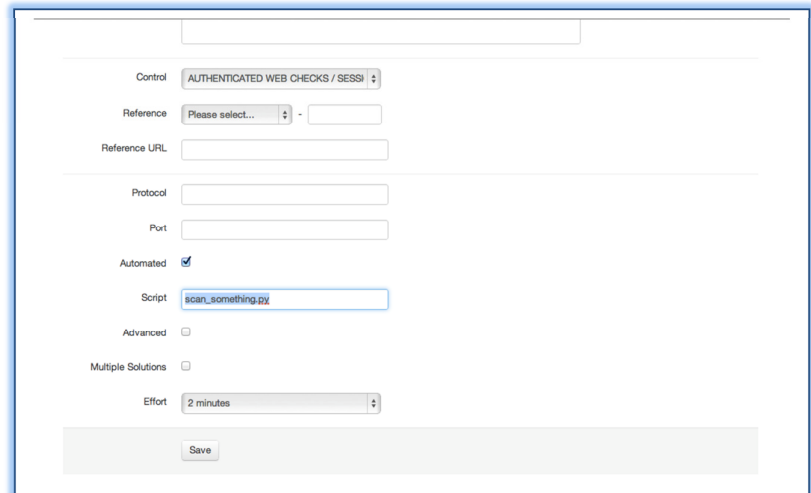
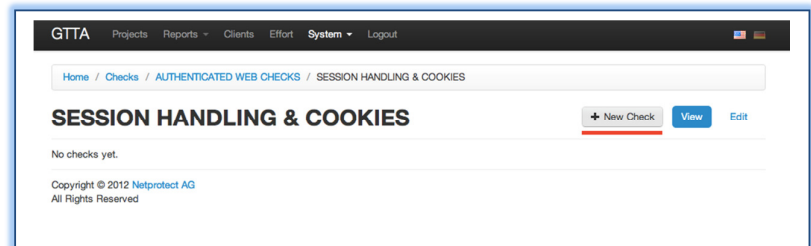
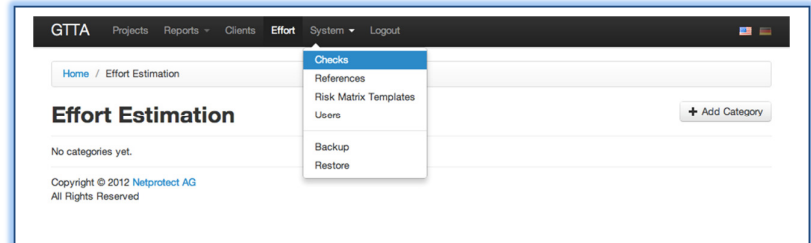
After class declaration, the script should call a function that starts check execution:

gtta.execute_task(CLASS_NAME), where CLASS_NAME is a name of the script's main class.

GUI Integration

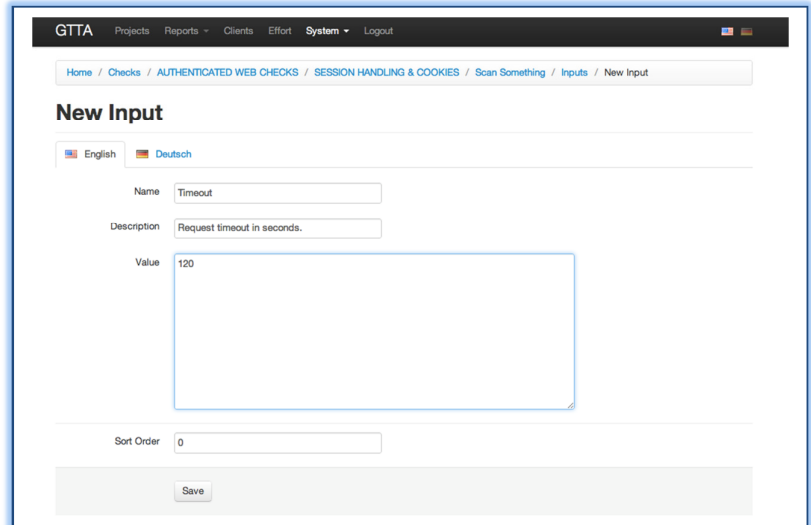
After you created a check script, you should register it in the GUI, so GTTA users will be able to use it. You will need GTTA admin privileges to do this. Please follow these steps to integrate your script:

1. Go to the *System/Checks* section in the main menu
2. Choose some check category or create a new one, then go inside that category
3. Choose some check control or create a new one, then go inside it
4. Press a **New Check** button to create a new check
5. You will see a check form. Fill in the check *Name* field, select some *Reference*, then set the *Automated* checkbox and enter your script's file name into the *Script* field
6. Press the **Save** button. You will see a message saying that your check has been saved.
7. If your script requires some additional input



data, you may create additional input fields in the *Inputs* section of the Check sub-menu

8. Press a *New Input* button to create a new input.
9. Fill in the *Name* field (how it will appear in a check list), brief *Description*, initial *Value* and *Sort Order*. *Sort Order* field is important – its value affects the order of input files, in which they will be passed to the script. Please keep the same order as the order of parameters in the *main* function of the script.



The screenshot shows the 'New Input' form in the GTTA application. The form is titled 'New Input' and has a language selector set to 'Deutsch'. It contains four input fields: 'Name' with the value 'Timeout', 'Description' with the value 'Request timeout in seconds.', 'Value' with the value '120', and 'Sort Order' with the value '0'. A 'Save' button is at the bottom.

10. Press *Save* button and repeat steps 8-9 to add other inputs.

After doing the steps above, the script is ready to be used from the GUI. Don't forget to create some test project and do a test run of the script.

Example

You will find script examples in the `gtta/` source directory of your distributive. Open the `gtta/scripts/` directory and choose any Python file as an example.

Full GTTA Integration Manual

Introduction

GTТА uses the unique packaging system to manage scripts that are used in checks. The system supports 2 types of packages - script and library packages.

- *Library Package* - can be used by *Script Packages* or other *Library Packages*. Usually it provides some common functions that can be useful for several scripts or libraries (for example, website crawler).
- *Script Package* - the script that performs test or attack actions. User can launch the script by linking it to a particular automated check and pressing the *Start* button on this check in some project that uses that check.

GTТА provides a lot of useful libraries and scripts out of the box. The built-in scripts apparently can't cover all possible needs, so you are able to upload your custom libraries and scripts as well. This guide describes all required steps to do that.

Currently GTТА supports scripts in Python and Perl programming languages. If you wish to use any other language or even a compiled binary program (yeah, that's possible), you should use either a Python or Perl wrapper for it. All scripts you upload must comply with GTТА API requirements (see the API section below for more information).

All scripts run in the jailed environment (the Sandbox) which is separate from the main system, so you may use scripts without being afraid to delete something important. In the worst case your Sandbox can become corrupted (for example, you may accidentally delete some system file). If this happens, you can just regenerate the Sandbox from the *System* → *Scripts* menu - it will delete the old Sandbox and create a new one with all scripts and dependencies installed.

Package Structure

GTТА package (both library and script) is a folder with some files in it. All packages at least contain 1 file named *package.yaml*, which contains package description. The package description file is a [YAML](#) file.

Example *package.yaml* (all section meanings are described below):

```
type: library
name: example
version: 1.7
system: false
dependencies:
  library:
    - testlibrary
    - customlibrary
  script:
    - somescript
```

```
- test
system:
- tcpdump
- nikto
deb:
- nmap_6.66-1_i386.deb
python:
- nltk
- pybloomfiltermmap
perl:
- Net::SSL
- LWP::UserAgent
```

The package description file has several sections. All sections are mandatory, unless otherwise specified.

- *type* - package type, either **library** or **script**.
- *name* - package name, allowed characters - a-z, digits, underline, comma and colon. The name must be in lower case letters. The name must be unique, i.e. there should be no packages with the same name in the system, otherwise the installation will fail.
- *version* - package version, any format.
- *system* - system package flag, either **true** or **false**. All custom packages must have this option set to **false**, since **true** is allowed only for built-in packages.
- *dependencies* - a list of dependencies for this package. This section is optional - some packages can be fully independent.

As shown above, there can be 6 types of dependencies. All sections described below are optional:

- *library* - here you can specify a list of GTTA libraries required for this package. The libraries specified in this section must be already installed when you package is being installed, otherwise the installation process will fail.
- *script* - this section contains a list of GTTA script dependencies. The scripts must be already installed prior to this package installation.
- *system* - system dependencies. GTTA uses Debian 6 operating system underneath and this section contains a list of system package dependencies that can be installed from the Debian APT repository.
- *deb* - Debian binary package dependencies. Not all required packages can be found in the standard Debian repository, so you can use your custom-built (or downloaded from the internet) Debian binary packages to install the required software. In order to install a custom Debian binary package you should put it to the GTTA package folder (the same folder that contains your *package.yaml*) and include it to this section of *package.yaml*. Please note, that the Debian binary package name must conform to the standard Debian naming convention: `<name>_<VersionNumber>-<RevisionNumber>_i386.deb`. Please refer to this documentation for more info - http://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html.
- *python* - Python library dependencies that will be installed using [pip package manager](#). Debian APT repository doesn't contain all available Python libraries and a lot of Python libraries in APT are outdated, so you might want to install recent Python libraries using pip. If you want to install the specific version of a Python

package and not the latest one, then please specify the name in the following format - **name==version** (for example, **pybloomfiltermmap==0.2.0**).

- *perl* - Perl library dependencies that will be installed using [CPAN](#). Unlike Debian APT repository, CPAN contains all available and most recent Perl libraries, so you might want to use it to install Perl libraries.

Please note, that **library** and **script** dependencies must be installed manually (using GTTA GUI) prior to your package installation, otherwise the installation will fail. Other dependency sections contain information about software that will be installed automatically during your package installation.

Library Package

Library Package is a package that provides some services (functions, data, etc.) for other GTTA libraries or scripts. There are no additional requirements for *Library Packages* - they can contain literally anything - custom Python modules, custom Perl modules, shell scripts, anything. GTTA API for scripts contains methods that will help you to get an absolute filesystem path for a certain library using only its name, so you will be able to use all files in a library from your scripts by including them (if we are talking about Python or Perl modules), by running them or by opening and reading them as regular files.

For example, if you have a Python library and you are planning to use it from multiple scripts, it's a good idea to upload that library as a GTTA *Library Package*, so your *Script Packages* won't have to provide their own copies of the same library. The same thing is with Perl libraries or custom-built Debian binary packages.

Script Package

Script Package is a package that contains a script that can be launched by GTTA to do some test or attack. GTTA has very strict requirements for scripts in order to be able to run them and make the developer's life much easier. There are 3 basic requirements for all *Script Packages*:

1. The script must be coded in Python 2.7 (preferred) or Perl 5.14 scripting language. You may create a script in any other language, but in this case you must create a Python or Perl wrapper for it.
2. The script must have a predefined entry point script named *run.py* or *run.pl* (for Python and Perl scripts, respectively).
3. The script must use GTTA API for the corresponding programming language.

The *Script Package* may contain any number of additional files. You can do with them anything you need - include them from your script as additional modules, you can read and write them or run them as an external program.

API

As have been said before, all scripts run in the Sandbox and are not connected to the main system. All changes done by scripts may be lost on Sandbox Regeneration, so don't store any sensitive information from your scripts to any files besides the result file. The result file is transferred to the main system after the script finishes its work, so it's the only safe place to store the information.

Low Level API

This section describes how scripts work on the lowest level and this information should be used only to understand how the process goes underneath. When you will create your script, you must use GTTA API, since the Low Level API may change in future and your scripts will stop working in the next GTTA versions, but GTTA API will remain the same as long as possible.

Scripts are launched in the background mode with certain command line arguments. Every command line argument passed to GTTA script is a path to a file, which contains some input data. An input file is a sequence of lines separated by the Line Feed symbols (LF, 0x0A or \n) and every single line is assumed to be a separate input record.

First two command line arguments (i.e. input files) are mandatory and are passed to every GTTA script:

1. Path to a target file, which describes a target system that should be checked and some basic check settings.
2. Path to an output file that should be used to write the script output.

Other command line arguments are completely optional and depend on the particular check requirements.

Target File Structure

Here is a list of lines, that describe the target:

1. *IP Address (Host Name)* – target's network address – either IP, or FQDN.
2. *Protocol* – a name of the protocol, that should be used during this check. For example, if a script performs a web test, this field may have one of these values: http or https. Of course, the script can just silently ignore this option.

Port Number – a target's port number to connect. This option could be ignored.

3. *Language* – user selected language code – English or German (en, de). Currently all scripts ignore this option, but if your script needs to output different information depending on the selected language – this option might be useful.

Result File Structure

Result file is a sequence of lines, separated by Line Feed symbols. Result file can contain several control sequences, that will be processed by the GUI to be able to display the information for the user in more convenient way. The control sequence is a simple XML text with special tags.

Result Tables

This control sequence allows displaying tables tied to the check. The control sequence looks like this:

```
<gtta-table>
  <columns>
```

```
<column width="0.4">Column Title</column>
<column width="0.6">2nd Column Title</column>
</columns>
<row>
  <cell>Some Data</cell>
  <cell>88</cell>
</row>
<row>
  <cell>Some Other Data</cell>
  <cell>99</cell>
</row>
</gtta-table>
```

This example contains several required control tags:

- *columns* – contains table column definitions
- *column* – defines a name for a single column. Attribute named *width* contains a width definition for the column. Width is a float number ≤ 1.0 (1.0 is a full table width).
- *row* – contains a list of cells for a row.
- *cell* – defines a cell content.

Please note that script can return only 1 table per launch.

Result Attachments

This control sequence allows adding attachments to the check object. The control sequence looks like this:

```
<gtta-image src="/path/to/file.png"></gtta-image>
```

Attribute *src* contains a local path to the file to be attached, so the check script is totally responsible for downloading that file to the local host. After attachment is added to a project, the system automatically deletes the source file.

The system allows to attach any kind of files, the main tag is called *gtta-image* because mostly this mechanism will be used for image attachments.

Python API

Downloads:

- [API Library](#)
- [Python Script Example](#)

Python API is a standard GTTA library which helps to create automated scripts in Python. The library implements some common tasks, such as parsing command line arguments, reading input files and writing to the output file. All Python checks should use this library. The Python API library is the preferred way to do automated check scripts.

In order to use the library, a script should import *core* library into its namespace. The script should declare some class inherited from the base class *Task*. The ancestor class should

implement a function named *main* – this function should do the script's job. Function *main* will be called just after library parsed all command line arguments. The list of function's parameters depends on additional arguments that have been passed to the script. Here's the example:

```
# coding: utf-8

from core import Task, execute_task

class Example(Task):
    """Example task"""

    def main(self, *args):
        """Main function"""
        self._write_result("Hello there!")

    def test(self):
        """Test function"""
        self.main()

execute_task(Example)
```

For example, if the script has been called like this:

```
run.py target.txt result.txt data1.txt data2.txt
```

then *main* function will be called with 2 parameters (except *self* class reference) – the first parameter will contain a list of lines in *data1.txt* file, the second will contain a list of lines in *data2.txt* file. If input files are empty, then empty lists will be passed.

For example, if the first file contains 2 lines and the second one is empty, a function call may look like this:

```
main(["first line", "second line"], [])
```

Attributes

Task class has some attributes, which give access to data in the target file:

- *self.target* – target name (either host name or IP address).
- *self.host* – domain name (if it was specified on the first line of the target file).
- *self.ip* – IP-address (if target file contained one). Only host name or IP address, but not both can be specified at the same time, since there is only 1 field in the target file that contains host address data.
- *self.proto* – protocol name.
- *self.port* – port number.
- *self.lang* – language code.

Constants

Task class has some constants, which are responsible for how input files are processed and how much time this particular task can take. You can override these constants in your script class:

- *TIMEOUT* - the maximum amount of time in seconds this script can work. After that time passes, the script gets killed. The default value is 60 seconds.
- *TEST_TIMEOUT* - the maximum amount of time in seconds this script can work in test mode. After that time passes, the script gets killed. The default value is 30 seconds.
- *PARSE_FILES* - boolean value, which determines if core library should split input files by lines. If this constant is **True** (default), then API reads all input files into memory and splits them by line. If this constant is **False**, then API just passes the file name as input argument. This constant can be useful if you have big input files that can consume a lot of resources if you read them in memory.

Modules, Classes, Methods and Functions

There are several useful functions and methods provided by API that you can use in your scripts:

- *core* - core module that contains everything needed to create a script.
 - *execute_task(class_name)* - function that is used to show which class contains the main script code. After class declaration, the script should call this function with the name of your script class as an argument.
 - *Task* - base class for all scripts.
 - *Task.main(*args)* - main script function. This function implementation is required for every script.
 - *Task.test()* - you can use this function to test the script without creating all the required files. Just run the script like this:

```
run.py --test
```

and the API will call *Task.test* function instead of *Task.main*.

- *Task._write_result(data)* - writes *data* into a result file. One method call will produce a single line in the output file.
- *Task._check_stop()* - call this method before and after doing time-consuming or blocking operations – it checks if the script should stop execution and exit. Scripts can be terminated by user request from GUI, so it's required to perform this kind of check.
- *Task._get_library_path(library)* - get local filesystem path to a *Library Package* named *library*. This method is useful when you need to call or use some files from the *Library Package*.
- *ResultTable* - result table class, which helps to create Result Tables very easily.
 - *__init__(columns)* - create table with *columns* as a list of column definitions. This list should contain of dictionaries with the following keys:
 - *name* - column name
 - *width* - column width (0 .. 1, 1 = 100%)

Example:

```
[{"name": "User", "width": 0.5}, {"name": "Count", "width": 0.5}]
```

- *add_row(row)* - add row with data to a table. *row* is a list of respective column values.

Example:

```
["John", "733"]
```

- `render()` - get a rendered XML table. You can get a result of this function and directly write it to a result file using `Task._write_result`.

Typical `ResultTable` usage example:

```
# create table columns
table = ResultTable([
    {'name': "Number", "width": 0.3},
    {'name': "E-mail", "width": 0.5},
    {'name': "Value", "width": 0.2},
])

# add rows
table.add_row(["1", "john@doe.com", "123"])
table.add_row(["2", "hello@world.com", "666"])
table.add_row(["3", "bob@bob.com", "444"])

# output table to results
self._write_result(table.render())
```

- `call` - calling external processes.
 - `call(command)` **[DEPRECATED]** - run external *command*. Returns a list of 2 values:
 - `ok` - boolean value, **True** if the program has run without any errors, **False** if program has finished with error.
 - `result` - text output of the called program.

WARNING! This function is deprecated and will be removed in upcoming releases of GTTA. Please use Python's standard `subprocess` module instead.

- `cd(directory)` - directory changing context manager. You can use it as follows:
 - with `cd("/path/to/some/dir")`:
 - # everything that goes here will be executed under `/path/to/some/dir` directory


```
test_file = open("test.txt", "r") # <-- the program tries to open
/path/to/some/dir/test.txt
```

Other Requirements

- It's required to implement error checks and try-catch blocks to handle all possible errors. If any error occurs during the execution, the script should show some information about that issue.

Perl API

Downloads:

- [API Library](#)
- [Perl Script Example](#)

Perl API is a standard GTTA library which helps to create automated scripts in Perl. The library implements some common tasks, such as parsing command line arguments, reading input files and writing to the output file. All Perl checks should use this library.

In order to use the library, a script should import [MooseX::Declare](#) (Perl API uses it for object-oriented interface) and `core::task` modules. The script should declare some class inherited from the base class `Task`. The ancestor class should implement a method named `main` – this function should do the script's job. Function `main` will be called just after library parsed all command line arguments. The list of function's parameters depends on additional arguments that have been passed to the script. Here's the example:

```
use MooseX::Declare;
use core::task qw(execute);

# Example Perl script
class Example extends Task {
    # Main function
    method main($args) {
        $self->_write_result("Hello there!");
    }

    # Test function
    method test {
        $self->main();
    }
}

execute(Example->new());
```

For example, if the script has been called like this:

```
run.pl target.txt result.txt data1.txt data2.txt
```

then `main` function will be called with 2 parameters (except self class reference) – the first parameter will contain a list of lines in `data1.txt` file, the second will contain a list of lines in `data2.txt` file. If input files are empty, then empty lists will be passed.

For example, if the first file contains 2 lines and the second one is empty, a function call may look like this:

```
main(["first line", "second line"], []);
```

Attributes

Task class has some attributes, which give access to data in the target file:

- `$self->target` – target name (either host name or IP address).
- `$self->host` – domain name (if it was specified on the first line of the target file).
- `$self->ip` – IP-address (if target file contained one). Only host name or IP address, but not both can be specified at the same time, since there is only 1 field in the target file that contains host address data.
- `$self->proto` – protocol name.
- `$self->port` – port number.
- `$self->lang` – language code.

Constants

Task class has some constants, which are responsible for how input files are processed and how much time this particular task can take. You can override these constants in your script class:

- `TIMEOUT` - the maximum amount of time in seconds this script can work. After that time passes, the script gets killed. The default value is 60 seconds.
- `TEST_TIMEOUT` - the maximum amount of time in seconds this script can work in test mode. After that time passes, the script gets killed. The default value is 30 seconds.
- `PARSE_FILES` - boolean value, which determines if core library should split input files by lines. If this constant is **1** (default), then API reads all input files into memory and splits them by line. If this constant is **0**, then API just passes the file name as input argument. This constant can be useful if you have big input files that can consume a lot of resources if you read them in memory.

Modules, Classes, Methods and Functions

There are several useful functions and methods provided by API that you can use in your scripts:

- `core::task` - core module that contains everything needed to create a script.
 - `execute(task_object)` - function that is used to show which object contains the main script code. After class declaration, the script should call this function with the instantiated object of your script class as an argument.
 - `Task` - base class for all scripts.
 - `Task->main($args)` - main script function. This function implementation is required for every script.
 - `Task->test()` - you can use this function to test the script without creating all the required files. Just run the script like this:

```
run.pl --test
```

and the API will call `Task->test` function instead of `Task->main`.

- `Task->_write_result($data)` - writes `$data` into a result file. One method call will produce a single line in the output file.
- `Task->_check_stop()` - call this method before and after doing time-consuming or blocking operations – it checks if the script should stop

- execution and exit. Scripts can be terminated by user request from GUI, so it's required to perform this kind of check.
 - *Task->_get_library_path(\$library)* - get local filesystem path to a *Library Package* named *\$library*. This method is useful when you need to call or use some files from the *Library Package*.
- *core::resulttable* - Result Table functions and classes.
 - *ResultTable* - result table class, which helps to create Result Tables very easily.
 - *new(\$columns)* - create table with *\$columns* as a list of column definitions. This list should contain of hashes with the following keys:
 - *name* - column name
 - *width* - column width (0 .. 1, 1 = 100%)

Example:

```
{name => "User", width => 0.5}, {name => "Count", width => 0.5}]
```

- *add_row(\$row)* - add row with data to a table. *\$row* is a list of respective column values.

Example:

```
["John", "733"]
```

- *render()* - get a rendered XML table. You can get a result of this function and directly write it to a result file using *Task->_write_result()*.

Typical *ResultTable* usage example:

```
# create table columns
$table = ResultTable->new([
  {name => "Number", width => 0.3},
  {name => "E-mail", width => 0.5},
  {name => "Value", width => 0.2},
]);

# add rows
$table->add_row(["1", "john@doe.com", "123"]);
$table->add_row(["2", "hello@world.com", "666"]);
$table->add_row(["3", "bob@bob.com", "444"]);

# output table to results
$self->_write_result($table->render());
```

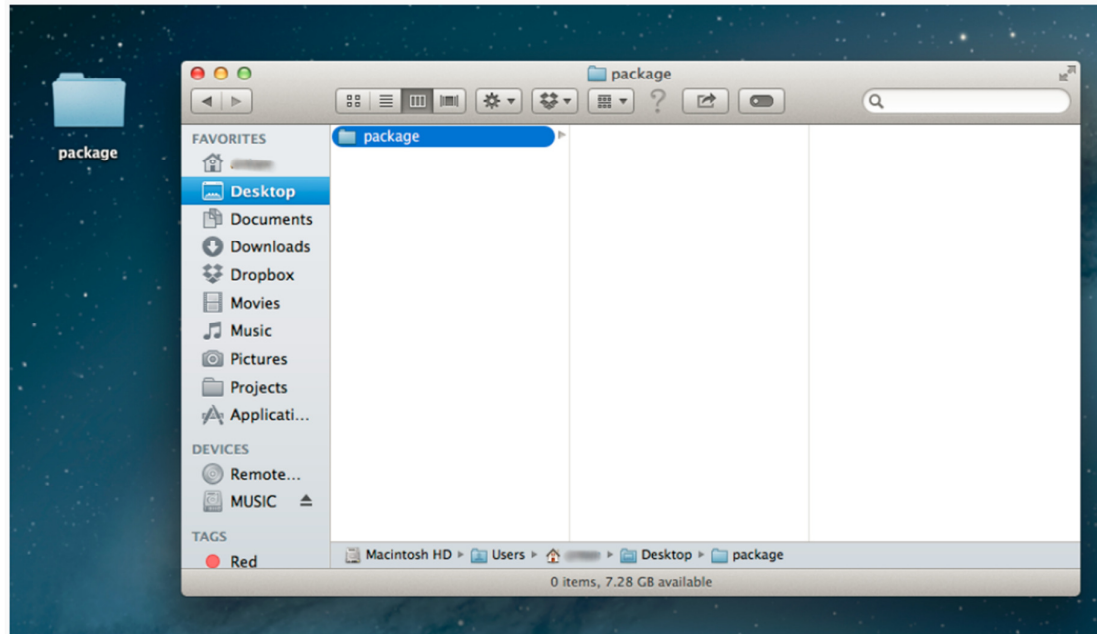
Other Requirements

- It's required to implement error checks and eval blocks to handle all possible errors. If any error occurs during the execution, the script should show some information about that issue.

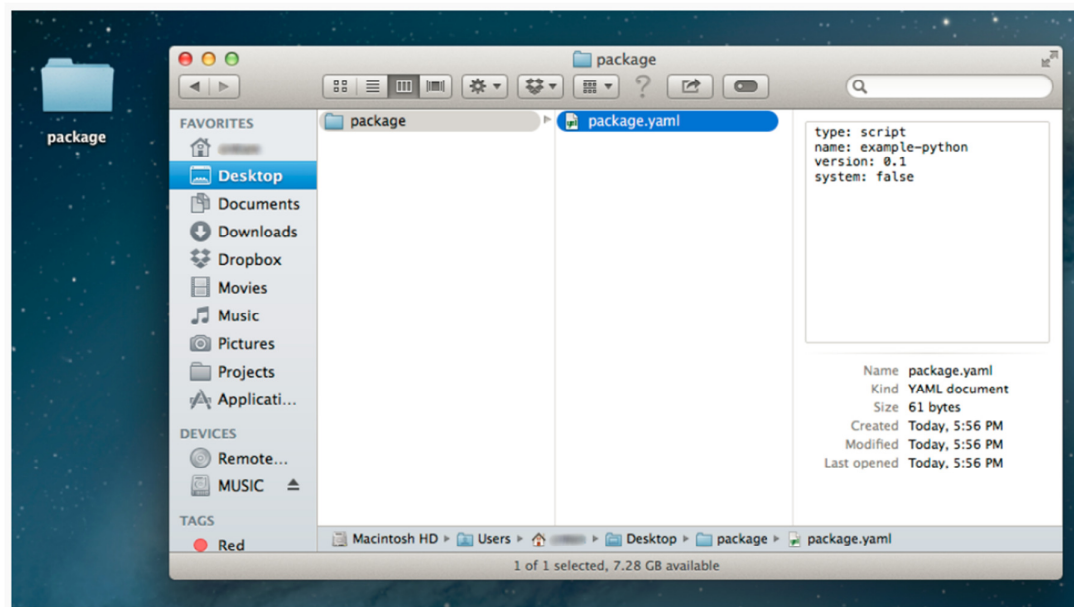
Building Your Package

Please follow these steps to build your package.

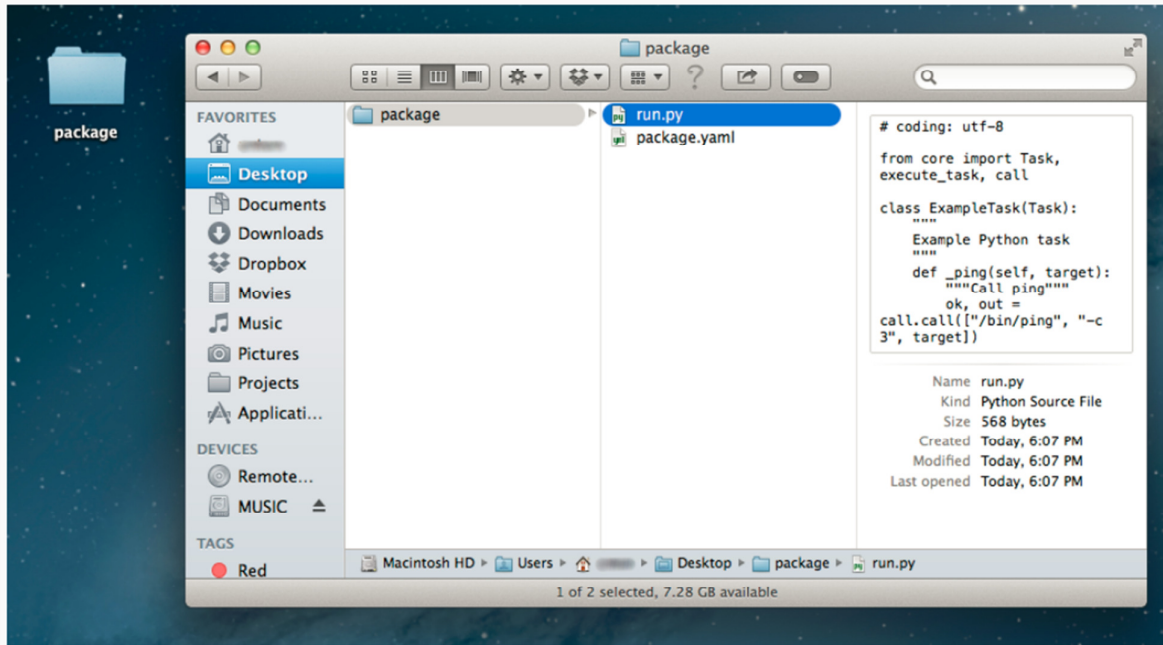
1. Create an empty folder which will contain package files (for example, *package*).



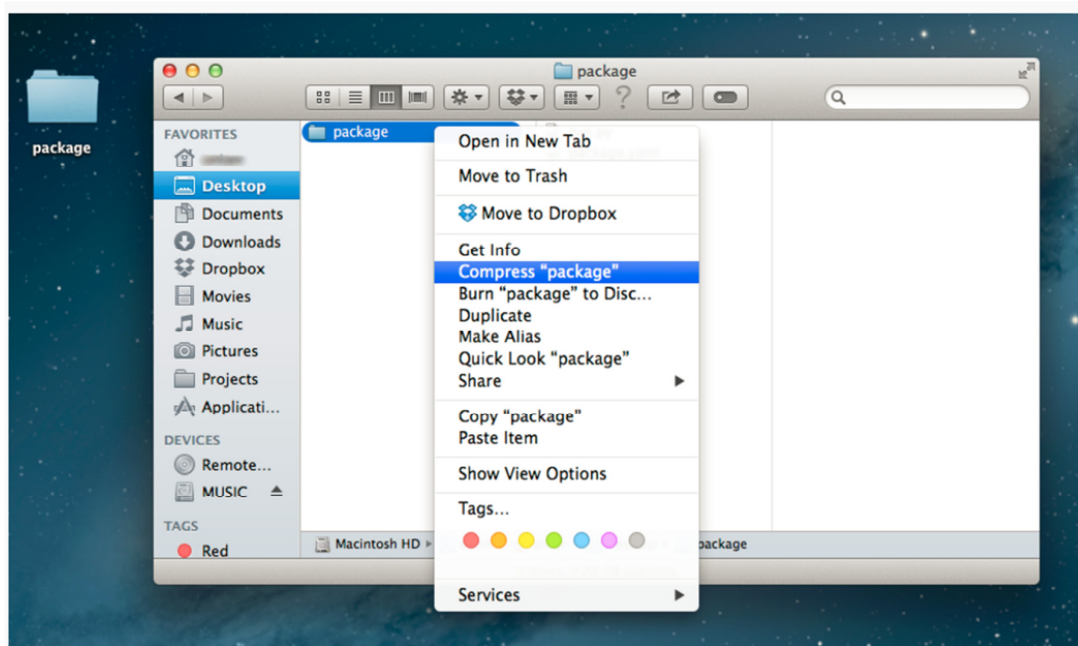
2. Create a package description file named *package.yaml* and fill it with required contents.

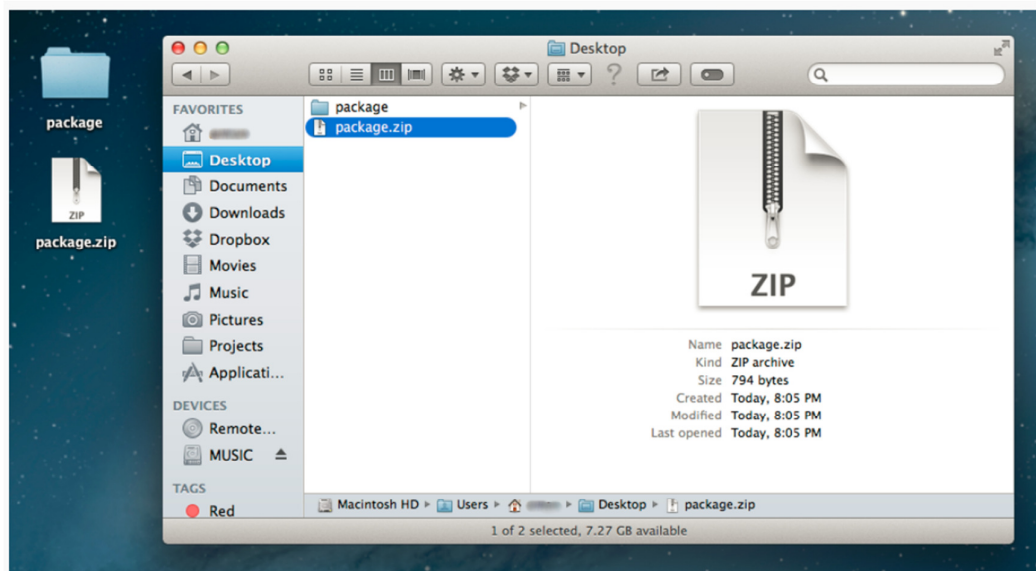


3. Put all required files into this folder (if you are doing the *Script Package*, then don't forget to put an entry point file - *run.py* or *run.pl*).

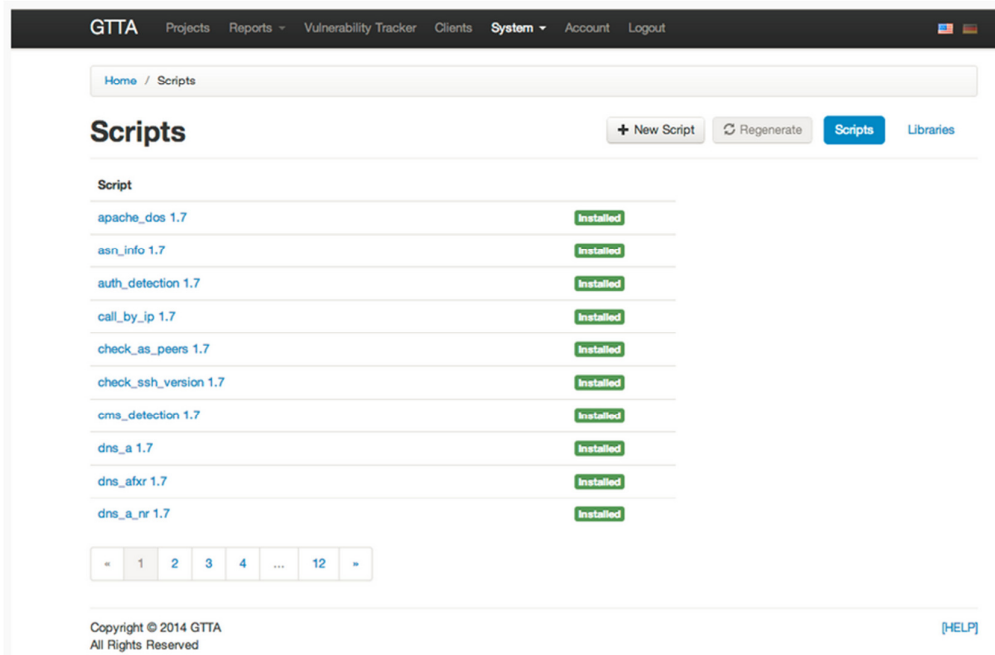


4. ZIP your folder, so you will have a file named *package.zip* as a result.





5. Go to *System* → *Scripts*, then choose the corresponding category (*Libraries* or *Scripts*).



6. Press the *New Library* or *New Script* button and upload the package file.

[Home](#) / [Scripts](#) / New Script

New Script

Package [Upload Package](#)

Install

[Home](#) / [Scripts](#) / New Script

New Script

Package [Upload Package](#)

Type

Name

Version

Install

7. Go to the last page of package list and check the status of your package. It will take a while until your package is installed.

[Home](#) / [Scripts](#)

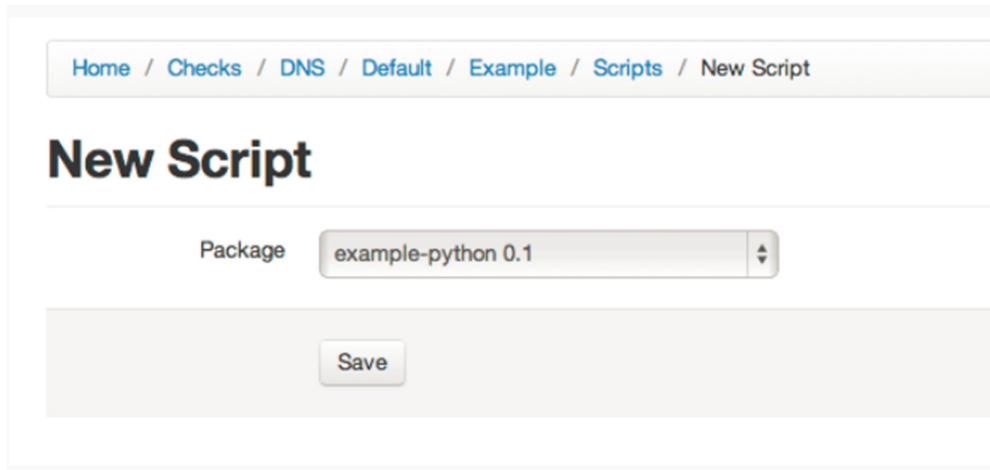
Scripts

[+ New Script](#) [Regenerate](#) [Scripts](#) [Libraries](#)

Script	
www_dir_scanner 1.7	installed
www_file_scanner 1.7	installed
example-python 0.1	installed ✕

[«](#) [1](#) [...](#) [9](#) [10](#) [11](#) [12](#) [»](#)

8. Done! Now you can use your package in checks or other scripts!



The screenshot shows a web interface for creating a new script. At the top, there is a breadcrumb navigation bar with links: Home / Checks / DNS / Default / Example / Scripts / New Script. Below this, the title 'New Script' is displayed in a large, bold font. Under the title, there is a label 'Package' followed by a dropdown menu that currently shows 'example-python 0.1'. At the bottom of the form, there is a 'Save' button.