

# **nessquik 2.5**

## **Developer Guide**

Tim Rupp

Last Modified  
07/06/2007

# Table of Contents

1. Introduction.....	5
2. Prerequisites.....	6
3. Coding Standards.....	8
• Comments.....	8
– Guidelines.....	8
– PHPdoc Tags.....	8
– Function and Class Comments.....	8
• Formatting.....	9
– Indenting.....	9
– PHP Tags.....	9
• Templating.....	9
• Expressions.....	10
• Functions.....	10
– Function Calls.....	10
– Function Definitions.....	11
• Naming.....	11
• Control Structures.....	12
• Including PHP Files.....	13
4. Architecture.....	14
• nessquik Web GUI.....	14
• nessquik client.....	14
– Keys.....	14
• scan-me-now.....	14
• portscan-me-now.....	14
5. What's Available and Where.....	16
• Directory Hierarchy.....	16
– async.....	16
– confs.....	17
– db.....	17
– deps.....	18
– docs.....	18
– images.....	18
– lang.....	18

–lib.....	19
–logs.....	19
–opt.....	19
–scripts.....	20
–setup.....	20
–templates.....	20
–templates_c.....	21
–tests.....	21
–upgrade.....	21
–xmlrpc.....	22
6. Database Schema.....	23
• division_group_list.....	24
• help.....	24
• help_categories.....	26
• metrics.....	26
• metrics_historic_scan_trends.....	28
• nasl_names.....	29
• plugins.....	30
• profile_list.....	32
• profile_machine_list.....	33
• profile_plugin_list.....	34
• profile_settings.....	36
• recurrence.....	39
• saved_scan_results.....	41
• scan_progress.....	42
• scanners.....	43
• scanners_groups.....	44
• special_plugin_profile.....	45
• special_plugin_profile_groups.....	46
• special_plugin_profile_items.....	47
• whitelist.....	48
7. Developing Metrics.....	50
• Metrics API.....	50
8. APIs.....	51
• Jobs API.....	52
• Using the Jobs API in PHP.....	53
–Creating a new client object.....	53
–Get the list of target machines for a particular profile.....	53

– Ask the server for a list of all the plugins.....	54
– Get the plugins for a particular profile.....	54
– Get the profile settings for a particular profile.....	54
– Get a list of the top X pending profile IDs.....	55
– Get the status of a scan profile.....	55
– Determine if a scan has been canceled.....	56
– Determine the number of scans being run by a client right now.....	56
– Specifically get plugins from a severity type.....	57
– Specifically get plugins from a family.....	57
– Get items in a specific scan profiles' special plugin list.....	57
– Reset a canceled, running, scans' status.....	58
– Set a scan's current progress.....	59
– Change a scan's status.....	60
– Set a scan's finish date.....	61
– Save a scan report.....	62
– Email scan results to users.....	63
– Add entry to exemption table.....	64
• SysOps API.....	66
• Using the SysOps API.....	66
– Creating a new client object.....	66
– Mark a nessquik client's scanner as being offline.....	67
– Mark a nessquik client's scanner as being online.....	67
9. Typical Program Flow.....	68
10. Templates vs. Themes.....	69
11. Unit Testing.....	70
12. Bug Fixes, Enhancements, Patches in General.....	71
13. Appendix A – nessquik Development Infrastructure.....	72
14. Appendix B – References.....	73
15. Appendix C - Special Plugin Profiles.....	74

# Introduction

This document provides detailed information, regarding nessquik, that developers may need if they decide to modify or enhance the system. It is divided into many sections, but overall it starts off general and becomes more specific. I start with what I consider the prerequisites for developing nessquik. This includes necessary software and suggested network configuration. Coding standards that must be followed are presented after that.

nessquik's architecture is presented next and includes information about the web UI, client, and nessquik's optional additions, scan-me-now and portscan-me-now. I also touch upon the reasoning for the particular architecture, caveats and limitations it presents, and what is being planned for the future. File structure is the next topic addressed. To be familiar with the code, it's handy to know where everything is and what the different directories contain.

This document uses several conventions. They are listed here for both your convenience and my convenience.

1. Text that refers to nessquik code uses the font type `courier`
2. Example code, for instance in the API section, is given a 20% gray background (according to OpenOffice) for clarity during reading.

## Prerequisites

I'm making the assumption that the reader has the following proficiencies.

- Moderate knowledge of the various outputs generated by the Nessus Vulnerability Scanner
- Advanced knowledge of the PHP scripting language
- Moderate to advanced knowledge of the Linux operating system. In particular Red Hat derivatives.
- Moderate knowledge of the MySQL database

I don't recommend that you modify code on a production system. That said, I think it's beneficial that you have high quality sample data to work with. I've found the only way to get good test data is to let people use the system and generate that data.

Create a backup of any live databases, and use them as your sample data when modifying the system. Ideally you'll have a separate instance of nessquik available on a different machine. This will prevent you from stepping on any production code.

I expect that you'll be using the latest release of PHP 5 for development purposes. For the general release of nessquik, a standard install of PHP 5 will work. For Fermi or site specific development, PHP will need to also have the following compiled and available

- PostgreSQL support
- Oracle Support

A copy of nessquik 2.5 with all the trimmings, including portscan-me-now and scan-me-now is also assumed.

nessquik is designed to work on Linux operating systems. No support is available for any other operating system. For this document, Scientific Linux Fermi 4.4 (a derivative of Red Hat Enterprise

Linux 4) will be used.

For development purposes, a multi-tiered installation of nessquik is assumed. See the diagram in Appendix A for details on the configuration that I am working with.

To recap, the following technical prerequisites should be met for development of nessquik.

- PHP 5
- MySQL 4.x
- Apache 2.x
- Linux. example: Red Hat
- Valid nessquik installation
- Good sample data available
- Network layout is known. (See Appendix A for an example)

## Coding Standards

To add to ease of readability and future maintenance, the following coding standards are used. If you decide to make changes to the software, it is expected that you abide by these guidelines. Any code that is submitted without following these guidelines will be returned to the sender.

Note that many of these guidelines have been borrowed (in certain cases word for word) from the GForge open source project. I've found that their expectations are very close to mine, and therefore instead of duplicating the effort of writing some of the standards, I have instead copied theirs. See the *References* section in Appendix B for a direct link to the GForge coding standards.

## Comments

### Guidelines

Non-documentation comments are strongly encouraged. A general rule of thumb is that if you look at a section of code and think "Wow, I don't want to try and describe that", you need to comment it before you forget how it works.

- C++ style comments (`/* */`) and standard C comments (`//`) are both acceptable.
- Use of perl/shell style comments (`#`) is prohibited.

### PHPdoc Tags

Inline documentation for classes should follow the PHPDoc convention, similar to Javadoc. More information about PHPDoc can be found here:

<http://www.phpdoc.de/>

### Function and Class Comments

Similarly, every function should have a block comment specifying name, parameters, and return

values.

```
/**
 * brief description
 * long description. more long description
 *
 * @author firstname lastname email
 * @param variable description
 * @return value description
 * @see
 */
```

## Note

The placement of periods in the short and long descriptions is important to the PHPdoc parser. The first period always ends the short description. All future periods are part of the long description, ending with a blank comment line. The long comment is optional.

## *Formatting*

### Indenting

All indenting is done with TABS. Before committing any file to SVN, make sure you first replace spaces with tabs and verify the formatting.

### PHP Tags

The use of `<?php ?>` to delimit PHP code is required. Using `<? ?>` is not valid. This is the most portable way to include PHP code on differing operating systems and webserver setups. Also, XML parsers are confused by the shorthand syntax.

### *Templating*

The Smarty templating engine is used for 99% of the output generated by nessquik. Templates should

contain display code and internal smarty operators only. Embedded PHP is not allowed in the templates even though Smarty supports this.

## ***Expressions***

- Use parentheses liberally to resolve ambiguity.
- Using parentheses can force an order of evaluation. This saves the time a reader may spend remembering precedence of operators.
- Don't sacrifice clarity for cleverness.
- Write conditional expressions so that they read naturally aloud.
- Keep each line simple.
- The ternary operator (`x ? 1 : 2`) usually indicates too much code on one line. `if... else if... else` is usually more readable.

## ***Functions***

### **Function Calls**

Functions shall be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter, and no space between the last parameter, the closing parenthesis, and the semicolon. Here's an example:

```
$var = foo($bar, $baz, $buz);
```

As displayed above, there should be one space on either side of an equals sign used to assign the return value of a function to a variable. In the case of a block of related assignments, more space may be inserted to promote readability:

```
$short           = foo($bar);  
  
$long_variable = foo($baz);
```

## Function Definitions

Function declarations using the following convention:

```
function foo_function($arg1, $arg2 = '') {
    if (condition) {
        statement;
    }
    return $val;
}
```

Arguments with default values go at the end of the argument list. Always attempt to return a meaningful value from a function if one is appropriate. Here is a slightly longer example:

```
function connect(&$dsn, $persistent = false) {
    if (is_array($dsn)) {
        $dsninfo = &$dsn;
    } else {
        $dsninfo = DB::parseDSN($dsn);
    }

    if (!$dsninfo || !$dsninfo['phptype']) {
        return $this->raiseError();
    }

    return true;
}
```

### *Naming*

Constants should always be uppercase, with underscores to separate words.

- True and false are built in to the php language and behave like constants, but should be written in lowercase to distinguish them from user-defined constants.

- Function names should suggest an action or verb: `updateAddress`, `makeStateSelector`
- Variable names should suggest a property or noun: `UserName`, `Width`
- Use pronounceable names. Common abbreviations are acceptable as long as they are used the same way throughout the project.
- Be consistent, use parallelism. If you are abbreviating number as 'num', always use that abbreviation. Don't switch to using `no` or `nمبر`.
- Database tables should be named with underscores `_` between words like: `my_table` and indexes on tables should be named with the table name first with the underscores removed, then field names `mytable_field1field2`.
- Variable names such as `$x` or `$y` are prohibited except in the case of loops where the value of the variable is an integer, and only used for the purpose of incrementing. In the rare times that poor variables names like these are used, inline documentation should be provided describing what the loop does.

```

$address_info = array(...);

for ($i = 0; $ < count($list); $i++)

```

## ***Control Structures***

These include `if`, `for`, `while`, `switch`, etc. Here is an example `if` statement, since it is the most complicated form.

```

if ((condition1) || (condition2)) {
    action1;
} elseif ((condition3) && (condition4)) {
    action2;
} else {
    defaultaction;
}

```

Control statements shall have one space between the control keyword and opening parenthesis, to distinguish them from function calls.

You should use curly braces even in situations where they are technically optional. Having them increases readability and decreases the likelihood of logic errors being introduced when new lines are added.

Switch statements are the exception to the rule above. The following is allowed:

```
switch ($condition) {
    case 1:
        action1;
        break;
    case 2:
        action2;
        break;
    default:
        defaultaction;
        break;
}
```

## ***Including PHP Files***

Anywhere you are unconditionally including a class file, use `require_once`. Anywhere you are conditionally including a class file (for example, factory methods), use `include_once`. Either of these will ensure that class files are included only once. They share the same file list, so you don't need to worry about mixing them - a file included with `require_once` will not be included again by `include_once`. Note: `include_once` and `require_once` are keywords, not functions. You don't need parentheses around the filename to be included, however you should do it anyway. Use of the constant prefix `_ABSPATH` is required as well to make sure you are including local install files only.

```
include(_ABSPATH.' /lib/pre.php ');
```

# Architecture

## ***nessquik Web GUI***

The web GUI provides the primary interface to the users. All activity related to creating scans, editing saved scans, and viewing results, is performed through the web. The GUI is a mixture of Javascript and HTML. It makes use of the Scriptaculous suite of javascript code. This library includes the prototype.js library. The backend nessquik code is written in PHP.

## ***nessquik client***

The nessquik client is used to schedule scans on Nessus servers. At this point in time a nessquik client is required on each Nessus scanner that will be used to run scans.

## **Keys**

To mildly control who is able to access the *Jobs API* provided by nessquik, client keys are used to distinguish which nessquik clients and which. This is a temporary solution and it is expected to change in the future.

## ***scan-me-now***

scan-me-now is a separate product from nessquik. It allows users to point their browser or other network aware software at the scan-me-now software and perform a full Nessus against the device contacting the server. scan-me-now allows administrators to give their users the ability to perform their own full scan when it is convenient for them.

## ***portscan-me-now***

portscan-me-now is a separate product from nessquik. It allows a user to point their browser or

other network aware software at the portscan-me-now software and perform a number of different port scans against the device using Nmap. portscan-me-now allows administrators to give their users the ability to perform their own scan when it is convenient for them.

# What's Available and Where

## *Directory Hierarchy*

nessquik's file structure is organized into several subdirectories. The majority of the code that performs page updating tasks is included in the `async/` directory.

## **async**

*Scripts that are run when asynchronous calls come in from the javascript code*

The format of a these files typically looks like this

```
<?php

session_name('nessquik');
session_start();
if (!defined("_ABSPATH")) {
    define("_ABSPATH", dirname(dirname(__FILE__)));
}

// requires go here
require_once(_ABSPATH.'/lib/Smarty.php');

// global variables come next
$username      = import_var('username','S');

// Then instantiate global objects
$db            = nessquikDB::getInstance();
$tpl           = SmartyTemplate::getInstance();

// Assign smarty paths
$tpl->template_dir      = _ABSPATH.'/templates/';
```

```

$tpl->compile_dir      = _ABSPATH.'/templates_c/';

// Get your action
$action = import_var('action', 'P');

// Switch on your action
switch($action) {
    case "javascript_action":
        action1;
        break;
    case "javascript_action_2":
        action2;
        break;
}
?>

```

## **confs**

*Configuration files that are specific to the nessquik web-ui software.*

For Fermi's version of nessquik, this directory also contains the errors/ folder which holds the 403.php script. This script is run when the user attempts to access the system without a valid KCA certificate. This script is specific to Fermi's nessquik. It will display output that is formatted like the rest of the system as opposed to a default 403 error page generated by Apache.

## **db**

*Database abstraction layers that nessquik supports.*

nessquik uses MySQL as it's primary database, however many more databases are supported out of necessity in Fermilab's environment. An abstraction layer for the Postgres database is included, and a stripped down version of ADODB is included for Oracle support. As per a request, I have also included MySQLi support as a replacement for the generic MySQL support in PHP.

## **deps**

*Software that I consider dependencies for nessquik.*

For the general release of nessquik, the nessquik client is the only dependency. For Fermilab's installation, the PHP oci8 PECL extension and a directory for certificate redirection is included too.

## **docs**

*Documentation associated with nessquik*

All the most current documentation is included in this folder. This includes items such as the AUTHORS file, CHANGELOG, INSTALL, and README. Items that are more complete than the provided documentation ( such as this developer manual and the user manual) are made available from the nessquik website.

## **images**

*Images that are displayed on all the webpages are kept here*

If you have an image and want to use it on any webpage, it should be placed in this directory and then referenced in the template using the path relative to the nessquik installation. For instance, the image hand.png would be linked in a template as.

```

```

## **lang**

*Holds word constants for different languages*

At this point in time, this directory includes nothing that is used by nessquik. Instead it is a placeholder for the future when I begin to replace static wording in the template files with Smarty variables. This directory may or may not be removed in the future, so please to not plan anything

around it.

## **lib**

*Contains all libraries of nessquik code*

All libraries that nessquik uses are contained in this directory. These libraries include classes, simple function files, and subdirectories which contain 3<sup>rd</sup> party software.

## **logs**

*All nessquik generated log files go in this directory*

If nessquik generates any log, that log will be written to this directory. Logs generally include the timestamp that they were created in their filename. Logs should never overwrite each other, and semi-random numbers have been added to files to make sure this doesn't happen. This directory by default is not viewable from the web. An htaccess file exists in it that denies GET and POST access to the directory.

## **opt**

*All optional software packaged with nessquik*

nessquik includes several optional software packages in the release. These packages, such as scan-me-now and portscan-me-now, are included in this directory. Unlike the linux /opt directory, none of the software in this folder is required to go in /opt. Instead, each package includes its own installation routines that should be followed.

## **scripts**

*Maintenance and routinely run scripts use by nessquik*

For development purposes, several extra scripts are included in the scripts directory to facilitate ease of maintaining the code. These scripts are normally removed before a release because they are useless outside of my development environment. This scripts directory also includes scripts that are necessary to make nessquik work properly. Normally users are instructed during installation to set up a particular crontab configuration. These scripts are usually referenced in those particular steps.

## **setup**

*All files needed to set up nessquik correctly*

During initial installation, users are directed to this folder to run scripts and perform tasks that are needed to get nessquik up and running. Once installation is complete, it's advised that the user remove this directory. At this point in time, none of the file can be run from the web. This makes more sense that you might initially think because a majority of the nessquik installation steps require CLI access anyway. In the future, a web based installation wized may become available.

## **templates**

*Web related output files are here*

This directory should not be confused with another common practice in web design called *theming*. Nessquik does not do themes, and there is no intention to ever add that functionality. The templates contained in this directory are used instead simply to separate the PHP logic from the HTML display logic.

## **templates\_c**

*Smarty cache directory for generating web pages*

This directory contains the cached files that Smarty will create when generating the webpages displayed in nessquik. This directory needs to be writable by the webserver, but does **not** need to be viewable by just anyone. It's recommended that htaccess files be used to restrict all access to this directory. If these precautions are not taken, it could introduce a substantial security hole.

## **tests**

*nessquik unit tests and other testing infrastructure*

Normally this directory is removed before a new release of nessquik is made. At this point in time, the files contained in this directory are primarily PHPUnit test cases. They are run during development to catch bugs before they are found by the general public. This directory didn't exist until 2.5. It was created out of upper management decision that came down the pipe requiring that all in-house software packages be tested extensively before production release.

In the future many more tests will be added to this directory included UI tests using the Selenium framework.

## **upgrade**

*Includes all scripts and documents necessary to upgrade nessquik*

Each upgrade sub-directory in this directory contains the scripts, documentation and other files that are needed to successfully upgrade nessquik from version to version. Each upgrade path is included in it's own subdirectory. Upgrades are **not** cumulative. You must upgrade sequentially even if you are using a very old release.

## **xmlrpc**

*API files that expose nessquik functionality to 3<sup>rd</sup> party developers*

nessquik exposes a small subset of its total codebase to an XML-RPC API. The files that make up the API are stored in this directory. In the future much more functionality will be added to the API so that developers in 3<sup>rd</sup> party organizations will be able to integrate nessquik into their environment more easily.

## Database Schema

The current database schema that nessquik uses is described below. Each field is documented and, where necessary, includes use cases in the code where that field is used. Sample data that may populate the field is also included. For more detailed information on each particular field, please refer to the actual SQL code that is provided with nessquik.

In nessquik 2.5, there are several new tables. Overall, there are twenty (20) tables. Some of them are not included or not used in the General release of nessquik. Those tables are marked as such in the list below.

- [division\\_group\\_list](#)
- [help](#)
- [help\\_categories](#)
- [metrics](#) (Fermi)
- [metrics\\_historic\\_scan\\_trends](#) (Fermi)
- [nasl\\_names](#)
- [plugins](#)
- [profile\\_list](#)
- [profile\\_machine\\_list](#)
- [profile\\_plugin\\_list](#)
- [profile\\_settings](#)
- [recurrence](#)
- [saved\\_scan\\_results](#)
- [scan\\_progress](#)
- [scanners](#)
- [scanners\\_groups](#)
- [special\\_plugin\\_profile](#)
- [special\\_plugin\\_profile\\_groups](#)
- [special\\_plugin\\_profile\\_items](#)
- [whitelist](#) (Fermi)

## ***division\_group\_list***

This table contains a list of all the groups that nessquik knows about. This table will always contain at least one (1) row with the value “All Groups”. Other than that, this table is used for different purposes depending on the nessquik release.

This table only has two fields.

- *group\_id*

A unique ID for each group

- *group\_name*

The name of a group pulled either from LDAP, or in the future, added manually.

For Fermi, this table is used as a fast lookup table to supplement the groups found in LDAP. It's updated on a nightly basis to reflect new groups that are added. A sample entry in the database is shown below.

```
+-----+-----+
| group_id | group_name |
+-----+-----+
|          10 | CD-DATABASES & APPLICATIONS |
+-----+-----+
```

Old groups are not removed, therefore the table will grow indefinitely. The General release of nessquik does not use this table nearly as much as Fermi. For comparabilities sake, this table should only include the single row for “All Groups”. This table will be used much more by the General release in the future.

## ***help***

The help table contains the *topics* that are seen in the help pages of nessquik. Categories are stored in a separate table, and the categories' ID is used in this table to associate a particular category

with a help topic.

In nessquik 2.5 there are four fields in this table.

- *help\_id*

A unique ID to associate with the help topic. Used by update and delete functions in nessquik.

- *category\_id*

The ID of the category that the particular help topic is located in

- *question*

The question, or topic, that is displayed to the end user on the help pages. The user will click on this text to view the answer

- *answer*

The complete answer to the question or topic that is posed in the previous field.

The questions and answers in this table are stored in the LONGTEXT MySQL field type. This allows for extremely long questions and answers to be stored. For all practical purposes these field lengths are longer than anyone will ever need. A sample (truncated) output is shown below.

```
+-----+-----+-----+-----+
| help_id | category_id | question | answer |
+-----+-----+-----+-----+
| 1 | 1 | What types of... | You can ad... |
+-----+-----+-----+-----+
```

The only difference between the Fermi and General releases of nessquik with regards to the help table is the particular topics that are in the table. Obviously those that relate Fermi specific features are not included in the General release.

## help\_categories

Help categories group together the topics that are found on the help page. Categories would be what you see to the left when you reach the help page. Clicking on a particular category will show all the questions or topics in that category.

There are three fields in the categories table.

- *category\_id*

A unique ID for the category. Used for maintenance of the category and for relating topics to categories.

- *type*

The type of category. There are two supported values; **A** for *admin* and **G** for *general*. Admin categories include topics that only an admin would have any interest in. General include topics that affect the whole user base.

- *category*

The name of the category as it will appear on the left hand side of the help screen.

A sample of the data contained in the categories table is shown below.

```
+-----+-----+-----+
| category_id | type | category      |
+-----+-----+-----+
|           1 | A    | whitelist     |
+-----+-----+-----+
```

As with the *help* table, the *help\_categories* table may be slightly different between releases.

## metrics

In the Fermi release of nessquik, a metrics table is used to store metric installation data. This

table is also included in the General release, but none of the General code makes use of it. It is included simply for future use after the authentication system has been added to nessquik General.

There are five fields in the metrics table.

- *metric\_id*

A unique ID for each metric so that when it comes time to remove a particular metric or perform other operations on it, there is no confusion due to name or displayed name.

- *type*

The type of the metric. In nessquik 2.5 there are two possible values; *graphs* and *reports*.

- *name*

The name of the graph, or more appropriately, the **class name** of the graph. This value is used especially by the metric maintenance, and metric admin scripts when instantiating metric classes.

- *display\_name*

The name of the metric as displayed to the administrator

- *description*

A short description of the metric.

A sample (truncated) of the data that you would find in this table is shown below.

```
+-----+-----+-----+-----+-----+
| metric_id | type   | name      | display_name | description |
+-----+-----+-----+-----+-----+
|      144 | graphs | NumberOf... | Number o... | Create graph... |
+-----+-----+-----+-----+-----+
```

For the General release of nessquik, the metrics table will become more useful in version 2.6 when user accounts are added to the application.

## **metrics\_historic\_scan\_trends**

This table is not really a default table that comes with the standard nessquik schema. Instead it is created by the Historic Scan Trends metric when that metric is installed. It is used to store counters for scan severity so that they can be quickly graphed without needing to mine data from the scan results.

There are eight fields in this table.

- *row\_id*

A unique ID for the trend in the database

- *username*

The username of the person who scheduled the scan

- *profile\_id*

The ID of the profile that was used when the scan was run

- *scanner\_id*

The unique ID of the scanner where the scan ran

- *date\_recorded*

The date that this record was recorded in the table

- *hole\_count*

The total number of holes found upon completion of the scan

- *warning\_count*

The total number of warnings found upon completion of the scan

- *note\_count*

The total number of notes found upon completion of the scan

A sample (truncated) of the data that would be contained in this table is shown below

row_id	username	profile_id	scanner_id	date_recorded
19	clifford	5b969088...	1	2007-03-17

hole_count	warning_count	note_count
0	0	12

Note that this table is also not available in the General release of nessquik because of the authentication limitation. In 2.6 it is likely that this table and it's features will be made available.

## nasl\_names

This table is used as another quick lookup table in nessquik. Every night, or depending on when you set the crontab entry, nessquik will query the Nessus server for its current list of plugins. It will also parse all the available plugins in the Nessus plugins directory. When it parses the file, it takes the file name, matches it with the plugin ID, and sticks the two in this lookup table.

In the web UI this table is queried when you are viewing more detailed information about a plugin. The nasl name, for instance, will show up in this more detailed view.

There are only two fields in this table.

- *pluginid*

The ID of the plugin

- *script\_name*

The name of the actual NASL script

An example of the data that may be contained in this table is shown below.

```
+-----+-----+
| pluginid | script_name |
+-----+-----+
|    20149 | Slackware_SSA_2005-310-01.nasl |
+-----+-----+
```

## plugins

The plugins table is one of the more important tables in nessquik. It is populated by querying the Nessus server for a list of all it's plugins, and then taking, literally, the full output and shoveling it into the table for quick retrieval by nessquik.

This table, like the groups table, is a continuously growing table at this point in time. While the nightly update-plugins.php script will not insert duplicate plugins, it also will not remove plugins that no longer exist. There are plans to fix this. There are also plans to monitor plugin updates and provide that data to the admin in a report.

There are eleven (11) fields in this table.

- *pluginid*

The unique ID of the plugin

- *family*

The family that the plugin is in. This is used in nessquik when you click the *by family* link

- *kb*

Ok, you got me. I haven't the foggiest idea what this field is used for. Until I find the documentation on it, it'll remain that way. I can say with certainty though that this field is not used by nessquik.

- *sev*

The severity of the plugin. The is used in nessquik when you click the *by severity* link

- *copyright*

The copyright applied to the plugin. Usually it is listed as Tenable because that's who authors most of them, but in addition to Tenable, user contributed plugins are also included.

- *shortdesc*

The plugin's short description. This info is used when displaying individual plugins in the nessquik UI. For instance, when you type in the search box and it returns a list of plugins.

- *rev*

The current revision of the plugin. As problems are found, or features are added to existing plugins, this rev number will be increased.

- *cve*

The CVE IDs that this plugin addresses

- *bugtraq1*

The bugtraq IDs that this plugin addresses

- *bugtraq2*

An alternative ID to refer to the check that this plugin performs. For instance, no bugtraq ID may exist (as seen by the value "NOBID" in the table, and this field would have "GLSA:200406-17" in it, referencing a Gentoo Linux advisory.

- *desc*

This is the full plugin description

Because of the volume of data in this table its difficult to show an example of it even truncated. As such, if you're truly interested in the contents, I'd ask that you go look at the actual data using the MySQL CLI tool or any of a myriad of different GUI tools.

## **profile\_list**

This table is the central hub for all profile related content. There are other profile tables that extend the data in this table, but for all practical purposes, if you are going to work with profiles, you should familiarize yourself with the content of this table because it is used a lot in joining with other data in other profile tables.

In nessquik 2.5, there are six fields in this table.

- *profile\_id*

This is a unique ID for the profile. It is used for a variety of operations that can be done on the profile including updating and removing items that are specific to it. This is also used as a foreign key in a number of other profile related tables.

- *username*

The name of the user who created the particular profile.

- *date\_scheduled*

The date the scan profile was scheduled to be run. This field will be updated if a plugin is rescheduled.

- *date\_finished*

The date the scan profile finished running. This field will be updated as the scan is

rescheduled and finishes running.

- *status*

The current status of the scan profile. This field can be one of several values.

- **N** for Not ready to run
- **P** for pending
- **R** for running
- **F** for finished.

- *cancel*

This field acts as an *on or off* switch. When a scan is running, if you set the value of this field to **Y**, nessquik will stop the scan. If the value is set to **N**, nessquik will not try to stop the scan.

An example of the tables (truncated) contents are shown below.

```
+-----+-----+-----+
| profile_id | username | date_scheduled |
+-----+-----+-----+ ...
| 38eaf6a3eb8d8573b | tarupp | 2007-03-16 09:26:01 |
+-----+-----+-----+
```

```

+-----+-----+-----+
| date_finished | status | cancel |
... +-----+-----+-----+
| 2007-03-16 19:26:00 | F | N |
+-----+-----+-----+
```

## profile\_machine\_list

In the machine list table, nessquik stores all the targets for all the scan profiles. This table is

used during scan run time to get a list of the targets to scan. It's a pretty simple table.

The machine list table has three fields.

- *row\_id*

A unique identifier for the machine entry. This is used for updating or removing information about the particular entry when using the web UI.

- *profile\_id*

The profile ID that the particular target is associated with. This ID must match at least one ID in the profile\_list table.

- *machine*

The target to include in the profile. At this point in time the target includes a prefix that specifies what type of target it is (since nessquik supports many). In the future I plan on splitting this prefix off to a separate field.

A sample of the data that is contained in this table is shown below.

```
+-----+-----+-----+
| row_id | profile_id | machine |
+-----+-----+-----+
| 165 | 22caddb1ba799bdd5ba417c721ec4a97 | :reg:172.17.2.21 |
+-----+-----+-----+
```

## **profile\_plugin\_list**

The plugin list table contains all the plugins that are associated with all the profiles. Upon creating a new scan, the list of plugins that you choose is inserted into this table so that the scan-maker can look them up during scan run time. This table is manipulated directly when you edit a scan profile too. For instance, removing plugins from your scan profile removes them from this table.

There are four fields in this table.

- *row\_id*

A unique identifier for the plugin entry in the table

- *profile\_id*

The scan profile ID that this plugin entry is associated with

- *plugin\_type*

The type of the entry in the table. There are five valid types

- all - All the plugins
- plu - A specific plugin
- sev - A severity category of plugins
- fam - A family category of plugins
- spe - A special plugin profile

- *plugin*

The value of the plugin. Each plugin type will have a different format of data that will be inserted in this field. The types and their expected data are listed below.

- all - The word **all**
- plu - The plugin ID of the plugin to use
- sev - The severity name of the severity of plugins to use
- fam - The family name of the family of plugins to use
- spe - The special plugin profile ID

An example of the data that you might find in this table is shown below.

```
+-----+-----+-----+-----+
| row_id | profile_id | plugin_type | plugin |
```

19	0797185328b32e2fb8ca29b9b3bbdfb1	all	all
----	----------------------------------	-----	-----

The plugin field in this table is semi-free-form. What I mean by this is depending on the value of the `plugin_type` field, a different value will be stored in the `plugin` field. For instance, if the `plugin_type` is *plu* then the value in `plugin` will be a number representing the plugin ID. A type of *fam* will result in the `plugin` value being the family name.

## profile\_settings

In nessquik 2.0, this table was called `user_settings`. Since then, it's purpose has become more geared towards settings that are specific to each scan profile. It contains a number of fields that can be customized for the scan profile. As more settings are added in the future, it is likely that this table will either be expanded, or instead be complemented with other setting tables.

In addition to profile settings, this table includes the general settings that you are able to configure. Your general settings are distinguished from scan profile settings by their use of the `sys` word in the `setting_type`.

In Fermi's release of nessquik, a user's `sys` entry is created upon first surfing to the nessquik page. Actually, this holds true for the General release as well. In nessquik 2.6 this is expected to change with the addition of the authentication system. When this system is in place, creation of the `sys` entry will be made at account creation time.

It contains sixteen (16) fields in nessquik 2.5

- *setting\_id*

A unique identifier for the particular entry in the settings table.

- *username*

The username of the person who either created the profile, or if this is their first visit to

nessquik, the person who visited the page and therefore had their `sys` entry created for them.

- *profile\_id*

The profile ID used for updating information about the profile in this table. Normal scan profiles are not the only ones to have profile IDs. Sys entries have profile IDs too.

- *setting\_name*

The name of the scan profile. If the entry is a sys entry, then a value of zero (0) is used.

- *setting\_type*

The type of the setting. There are two values allowed here.

- `sys` - Each user has one (1) `sys` entry. This defines the values that are found on the *general* settings page.
- `user` - Each scan that is created by a user is given a `user` entry. All the settings for the scan profile are then stored in this entry in the table. A user may have an arbitrary number of `user` entries in this table.

- *short\_plugin\_listing*

An on/off switch that tells the web UI whether to display extra details in the plugin search list or not. Note that this setting has no effect for individual scan profiles.

- *ping\_host\_first*

This setting is an on/off switch that tells the nessquik clients' scan maker whether to create a `nessusr` file with hosting pinging enabled or disabled. A value of one (1) will enable pinging. A value of zero (0) will disable it.

- *report\_format*

The default format for a scan profiles output. This is the format that is sent via email to the

user. If the report is saved, the user can go back and choose to view the report in a number of other different formats.

- *save\_scan\_report*

Specifies whether nessquik should save the scan results to the database or just drop the results after it emails them. By default, all scan results are saved.

- *port\_range*

The port range to use in the scan. Nessquik only supports simple port ranges at this point in time. You do not have the ability to specify more complex forms, such as “23, 45-90, 1024”. This functionality will likely be added in the future though because Nessus supports this format for ports. By default, the word *default* is used for the port range. According to the Edgeos documentation (Appendix B), the keyword is expanded by Nessus to equal the port range 1 to 15,000.

- *custom\_email\_subject*

This field holds the custom email subject that nessquik's scan maker will use when sending you an email. Nessquik supports macros in the subject line. These macros are stored as-is in the table field. They are replaced with real data upon scan completion by the scan-maker.

- *alternative\_email\_list*

A colon (:) separated list of email addresses where the scan report will **also** be sent to. Scan reports are always sent to the original creator. In addition, you can send the report to others with this list.

- *alternative\_cgibin\_list*

A colon (:) separated list of paths that will be used by Nessus when detecting vulnerable CGI scripts. These paths should be the full path from the base domain name that you are

scanning.

- *recurring*

An on/off toggle switch that tells nessquik whether the scan profile has any recurrence associated with it.

- *scanner\_id*

The ID of the scanner that this profile is associated with. Scanners must be defined in the administration area of nessquik. Scans must then be assigned to a scanner so that the specific nessquik clients know who is supposed to run which scans.

Due to the large amount of data that is contained in this table, if you are truly interested in seeing an example of the data, I'd ask that you refer directly to the MySQL table once you have created at least one scan.

## **recurrence**

A scan profile's recurrence settings are stored in this table. In nessquik, there are a number of ways to configure recurrence. Daily, weekly, or monthly, this table holds the rules that the cron script follows when it goes about rescheduling your scans. This table is only used by the cron script when it comes to automatically rescheduling your scans.

There are six fields in this table.

- *recurrence\_id*

A unique identifier for the recurrence entry

- *profile\_id*

The ID of the profile that this recurrence entry is associated with.

- *recur\_type*

The type of scan recurrence that you chose. There are three valid choices

- **D** for daily recurrence
- **W** for weekly
- **M** for monthly

- *the\_interval*

The interval that the recurrence will happen. Think of it as saying “reschedule the scan every \_\_\_\_\_ days/weeks/months” and fill in the blank with this number.

- *specific\_time*

This is the time that you want the scan to reoccur on the date or days you chose. It's the value that is decided by the small clock fields that are shown when you click the recurrence check box. The full date is saved, however only the time is used.

- *rules\_string*

The recurrence rules. Depending on the type of recurrence that you chose, there are different ways that the next scheduled date should be determined. For example with monthly recurrence, you can specify which day of the month to reschedule the scan on. That day is stored in the rules string.

An example of the data that you might find in this table is shown below.

```
+-----+-----+-----+
| recurrence_id | profile_id           | recur_type |
+-----+-----+-----+ ...
|           1 | a0b772d5b304c9396d802bc54a... |      M     |
+-----+-----+-----+

+-----+-----+-----+
| the_interval | specific_time       | rules_string |
```

```

... +-----+-----+-----+
    |           1 | 2007-05-09 11... | day:1           |
    +-----+-----+-----+

```

## saved\_scan\_results

This table acts as central storage for all of the scan results. In nessquik 2.5, saving reports is enabled by default, so there is a very good likelihood that this table will grow to become very large. In Fermi's environment, this table is located on a completely separate database due to security concerns. In the General release of nessquik this functionality is not available. It's not impossible to add, but I figure it's not something that the majority of people need.

There are four fields in this table.

- *results\_id*

A unique identifier for the entry containing a set of results

- *profile\_id*

The ID of the profile that the results are associated with

- *saved\_on*

The date and time that the results were saved to the database

- *scan\_results*

The actual scan results. For the most part, this is raw NBE with the exception that it is triple colon delimited (:::). The theoretical limit for the amount of data that this field can hold is something on the order of 4 GB. I don't expect any scan results to ever come close to reaching this. If you do come close, please don't use nessquik.

Due to the massive volume of data contained in this table, it is impractical to show a data sample. Please see the raw database entries if you are interested.

## scan\_progress

This table holds the data that is used to create the progress meters on the scans page while the scans are running. There are two types of progress as reported by Nessus' output while the scan is running; portscan and attack.

Portscan progress is a tricky beast because it's not entirely accurate. In the past this output would cause the progress bar to appear to skip. In nessquik 2.5 I have decided to only report on the progress of attacks. The portscan status is still logged, however it is not used in the UI.

The data in this table is updated mainly during the scan, however if you reschedule a scan, this table will be updated and the progress for the scan profile will be reset to zero.

There are four fields in this table.

- *scan\_id*

A unique ID for the entry in the progress table

- *profile\_id*

The profile ID to associate the progress with

- *portscan\_percent*

The percent completion of the portscans against the targets

- *attack\_percent*

The percent completion of the attacks against the targets

If you've been keeping track at home, it's not difficult to figure out what the data in this table is going to look like. The percents that are stored in the table are out of 100%. See below for an example.

```
+-----+-----+-----+-----+
| scan_id | profile_id      | portscan_percent | attack_percent |
+-----+-----+-----+-----+
```

32	5bd3464425788...	0	72
----	------------------	---	----

+-----+-----+-----+-----+

## scanners

The scanners table contains all the information about the Nessus scanners you have configured the nessquik clients to use. I expect that this table will be greatly extended in the future to accommodate many requests that I've received with regards to the scanners.

This table holds the information about the scanners that are associated with your scan profiles. If a scanner doesn't exist in this table, you won't be able to schedule a scan. You can manipulate this table by surfing to the settings page in the administration area of nessquik.

There are five fields in this table, however only three of them are actively used.

- *scanner\_id*

A unique ID created for each scanner. This ID is used by profiles to specify which scanner to run a given scan on.

- *name*

The name of the scanner. This does not need to be the hostname for the machine. This information is only used as a convenience for the end user so that if they are scheduling a scan, they know which server it will run from.

- *client\_key*

A special, unique, key that acts as a means of authenticating a nessquik client when requesting details about a scan.

- *privileged*

Used only by Fermi at the moment. This is a simple on/off bit in the table and is used to specify whether a scanner should be granted more privileges than normal. In our particular

case, we offer the nessquik client to a number of different groups on site, but we dont allow those groups to arbitrarily create exemptions for themselves. By setting the privileged flag, it would allow the scanner to perform operations on the database that go beyond normal scanner requests.

- *online*

Currently not used. In the future this will be an on/off bit that will be used to tell the status of a Nessus scanner. Of course, the processes that twiddle this bit will need to either be on the Nessus server, or have credentials available to log in to the nessus server.

```
+-----+-----+-----+-----+-----+
| scanner_id | name      | client_key      | privileged | online |
+-----+-----+-----+-----+-----+
|           1 | ovaltine  | wt4gIlno6Hbi... | 0          | 0      |
+-----+-----+-----+-----+-----+
```

### ***scanners\_groups***

This table acts as a mapping table between the scanners and the groups that are allowed to use the scanners. In Fermi's environment this table allows us to maintain a much finer level of control over who can use which scanners. Since groups often set up their own scanners, this requirement is more of a necessity than a convenience.

There are three fields in this table.

- *row\_id*

A unique identifier for the row in the table

- *group\_id*

The ID of the group that a scanner is associated with. This ID is associated with the *group\_id* found in the *division\_group\_list* table

- *scanner\_id*

The ID of the scanner that the group is allowed to use. This ID is associated with the *scanner\_id* found in the *scanners* table.

A sample of the data that will likely be stored in this table is shown below.

```

+-----+-----+-----+
| row_id | group_id | scanner_id |
+-----+-----+-----+
|      5 |      105 |          1 |
|      7 |      105 |          2 |
+-----+-----+-----+

```

### ***special\_plugin\_profile***

Special plugin profiles are just user defined meta plugins (plugins containing more plugins). See Appendix C for a graphic describing a special plugin profile. This table is the primary place where general information about the plugin profiles is stored. Other tables complement the data stored in here by referencing fields in this table.

When you create a plugin profile, it's name is stored in here and a unique ID for it is generated. That name is displayed to everyone who is allowed to access that profile (as defined by the *special\_plugin\_profile\_groups* table).

There are only two fields in this table

- *profile\_id*

A unique ID for the plugin profile. This ID is used in a number of different areas in nessquik. It is a foreign key to the other profile plugin tables, and it is used as the identifier for the plugin when you choose it from the “special plugins” list when creating a new scan.

- *profile\_name*

The name of the special plugin profile as will be seen by users

A sample of the data that you are likely to see in this table is shown below

```
+-----+-----+
| profile_id | profile_name |
+-----+-----+
|          1 | critical vulnerabilities |
+-----+-----+
```

### ***special\_plugin\_profile\_groups***

The purpose of this table is nearly identical to that of the `scanners_groups` table, except that this guy relates to the plugin profiles. Access to plugin profiles is controlled by groups. This table is a quick lookup table used for checking if a specific user is allowed to use a plugin profile based on their group membership.

There are three fields in this table

- *row\_id*

A unique identifier for this record in the table

- *group\_id*

The ID of the group that is allowed to use a specific plugin profile. This ID is associated with the `group_id` found in the `division_group_list` table

- *profile\_id*

The ID of the special plugin profile that the group is allowed to use. This ID is associated with the `profile_id` found in the `special_plugin_profile` table.

A sample of the data that you will find in the table is shown below.

```
+-----+-----+-----+
| row_id | group_id | profile_id |
+-----+-----+-----+
```

1	0	1
---	---	---

## ***special\_plugin\_profile\_items***

This table stores the contents of the special plugin profiles. Since a plugin profile can be made up of a variety of other plugin types, this table is needed to know what kind of stuff is in the plugin profile. In the scan-maker script, the content in this table is evaluated and parsed down to reach the eventual list of plugins to use for the scan. Families, severities, and individual plugins can all be included in a plugin profile.

This table contains four fields. Most of which you'll probably be familiar with.

- *row\_id*

A unique identifier for this row in the table

- *profile\_id*

The ID of the plugin profile that this particular plugin is associated with. A profile can have an arbitrary number of items in it.

- *plugin\_type*

The type of the plugin for this entry. There are three valid types

- **plu** for an individual plugin
- **sev** for a severity of plugins
- **fam** for a family of plugins

- *plugin*

The value that matches the type of plugin specified. Each type of plugin has a different data format.

- plugins will have the ID of the plugin in this field
- families will have the name of the family
- severities will have the name of the severity

A sample of the data that you will see in this table is shown below.

```

+-----+-----+-----+-----+
| row_id | profile_id | plugin_type | plugin |
+-----+-----+-----+-----+
|      1 |          1 | plu          | 15545 |
+-----+-----+-----+-----+

```

## ***whitelist***

The whitelist table is specific to the Fermi release of nessquik. There are separate databases that we talk to when verifying a user is allowed to scan a particular machine. Computer Security does not have direct control over those databases, so it is not possible to change the data in them to say that a specific user is now allowed to scan a machine.

We needed to be able to override the permissions stored in these databases though on a case by case basis. The whitelist is a table that CST controls and is able to add new information to. Entries in the whitelist table are assumed to be allowed to scan a particular device regardless of what the other databases on site say.

As an example of how this is used, consider the position a GCSC is put in when they need to scan a machine they are not a registered admin for. GCSC's need to be able to scan any machine under their jurisdiction, however just because it may be under their jurisdiction doesn't mean they are a registered admin for the machine. In the normal flow of things, this would prevent them from legitimately scanning a target.

Central web admins are another good example especially when it comes to VHosts. The central web admins must be able to scan any and all VHosts regardless of whether they have explicit

permission to do so or not. The whitelist covers them too.

There are three fields in this table.

- *whitelist\_id*

A unique identifier for the entry in the table

- *username*

The username of the person who will be allowed to target the listed entry

- *listed\_entry*

The entry that the user is allowed to scan. Data in this field can be either one of three valid types.

- IP address
- CIDR block
- Range of IP addresses

An example of the data that is stored in this table is shown below.

```
+-----+-----+-----+
| whitelist_id | username | listed_entry |
+-----+-----+-----+
|           56 | tarupp   | 131.225.70.20 |
+-----+-----+-----+
```

## Developing Metrics

nessquik stores the vulnerability scan results in a database table. Using this information, nessquik is able to create metrics. Metrics are only available to administrators at this point in time. This functionality will be extended to the end user in the future.

The metrics included with nessquik 2.5 are

- Number of Scans
- Scan Trends

### ***Metrics API***

New metrics can be created to create graphs or reports as the developer sees fit. For detailed information about developing metrics, see the document *Metric Developers API*.

## APIs

At the present, nessquik includes two APIs. More APIs are planned for the future that will expose more of the system to be used in development of 3<sup>rd</sup> party software. XML-RPC is used for the underlying API.

For PHP developers interested in using either of the APIs, I strongly encourage you to use the IXR library. This library can be found in the `libs/` directory in nessquik. This library makes it much easier to use the available methods.

For all other developers, I'm afraid I can't commit to a specific client side library since I'm not familiar enough with other languages and their support for XML-RPC libraries. If you use a particular library and would like to contribute examples of how you use that library, please contact me and I will include your examples for others.

In addition to providing code samples in the documentation below, I also include the raw XML that is returned by nessquik. This should allow you as a developer to understand the format of the returned data so that you can work with it even if your particular language doesn't have a specific library available for XML-RPC.

The two current APIs are listed in detail on the next several pages.

## ***Jobs API***

nessquik supports an API called the *Jobs API*. This is primarily used by the nessquik client when running or scheduling new jobs. The jobs API includes a standard set of available methods. Several of these methods require authentication and or authorization before they will run. See the table below for a list of the available standard API methods.

<i>Method Name</i>
jobs.getMachines
jobs.getAllPlugins
jobs.getProfilePlugins
jobs.getProfileSettings
jobs.getPendingProfileIds
jobs.getStatus
jobs.getCancel
jobs.getCountRunning
jobs.getPluginsBySeverity
jobs.getPluginsByFamily
jobs.getSpecialProfileItems
jobs.setResetCancel
jobs.setProgress
jobs.setStatus
jobs.setFinishedDate
jobs.saveReport
jobs.emailResults

nessquik's *Jobs API* can also be extended to include third party methods. Fermilab for instance has added their own API calls for maintaining their exemptions table. Most of the methods supplied in the *Jobs API* require a key be specified.

## Using the Jobs API in PHP

Using the *Jobs API* from PHP is very easy. In the following examples I will specifically be using the IXR library. I encourage all PHP developers to do the same. In several of the examples, I make reference to a PHP constant called `_CLIENT_KEY`. This constant must be defined by you prior to making several API calls. This client key is used in a way similar to an authentication token. Profiles are typically associated with a scanner, and that scanner is given a key so that it can prove to nessquik that it is allowed to ask for profile information. An example client key is shown below, including the define statement.

```
define('_CLIENT_KEY', 'y4RRYo0hysf33KjIDZfPeT8775x8gume');
```

## Creating a new client object

In PHP, before you can use the API, you need to create a client object that can talk the API. This is very simple and can be accomplished with the following code. All future operations can be done through this new object.

```
// Required. Before any other API calls,  
// these two lines must run  
require_once(_ABSPATH.'/lib/IXR_Library.php');  
$client = getIXRClient();
```

## Get the list of target machines for a particular profile

This API method will query nessquik for a list of the machines targeted in a particular profile.

```
// Profile ID as found in the profile_list database table  
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";  
  
$client->query('jobs.getMachines', _CLIENT_KEY, $profile_id);  
$machines = $client->getResponse();
```

## Ask the server for a list of all the plugins

The API allows you to query nessquik for a list of all the plugins that nessquik knows about. The list that is returned is very large, but only contains the plugin ID.

```
$client->query('jobs.getAllPlugins');
$scanner_set = $client->getResponse();
```

## Get the plugins for a particular profile

You may be interested in knowing which plugins are currently used in a particular scan profile. This method will query nessquik and return a list of the plugins to you for that profile. You must specify a plugin type to ask for with this method call. The available *types* are

- all
- sev
- fam
- plu

You'll also need the profile ID that you are asking for plugins for, and a valid client key that is allowed to read that particular profile.

```
// Profile ID as found in the profile_list database table
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

// To query for plugins that were chosen by severity
$client->query('jobs.getProfilePlugins,
              _CLIENT_KEY,
              $profile_id,
              "sev"
            );
$plugins = $client->getResponse();
```

## Get the profile settings for a particular profile

You can retrieve all the profile settings for any scan profile assuming you have a client key that is allowed to read those profiles, and you have the profile ID that you want to read.

```
// Profile ID as found in the profile_list database table
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

$client->query('jobs.getProfileSettings',
              _CLIENT_KEY, $profile_id
);
$settings = $client->getResponse();
```

## Get a list of the top X pending profile IDs

This API method call is used primarily by the scan-runner when forking off an arbitrary number of scans to run. The method returns a list of profile IDs that are in a pending state that need to be run. A valid client key must be provided, as must a limit. The limit is the number of profiles that you want returned, and must be a number.

```
// Specify the number of profile IDs
// That I want returned
$limit = 20;

$client->query('jobs.getPendingProfileIds',
              _CLIENT_KEY,
              $limit
);
$profile_ids = $client->getResponse();
```

## Get the status of a scan profile

Oftentimes you may want to know the current status of a scan profile. nessquik uses this a lot on the scans page dashboard to show you which scans you have pending, or running, etc. This API

method will return the single character status of the scan. The return value will be one of the following characters

- **N** – Not Ready to Run
- **R** – Running
- **P** – Pending
- **F** – Finished

To call the method, see the example below.

```
// Profile ID as found in the profile_list database table
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

$client->query('jobs.getStatus', _CLIENT_KEY, $profile_id);
$status = $client->getResponse();
```

## Determine if a scan has been canceled

There's a good likelihood that if you use this method, you'll always only see a return value of **N**. While you might start to think that this is a bug, it should be noted that a properly set up nessquik client will call this method every 10 seconds during a running scan. Therefore if you're really interested in using this method, you'll have about a 10 second window where it could return a value of **Y**.

In any case, see the example below.

```
// Profile ID as found in the profile_list database table
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

$client->query('jobs.getCancel', $profile_id);
$canceled = $client->getResponse();
```

## Determine the number of scans being run by a client right now

This method returns the number of scans that a nessquik client is currently running. A valid client key must be supplied so that the API knows which client to pull counters for.

```
// As for the number of scans
```

```
$client->query('jobs.getCountRunning', _CLIENT_KEY);  
$running_count = $client->getResponse();
```

## Specifically get plugins from a severity type

This method is a helper method that wraps around an internal API method. It's a shortcut for retrieving only the plugins of a specific severity instead of all the plugins.

```
// Only get plugins that have the severity "dos"  
$severity = "dos";  
  
$client->query('jobs.getPluginsBySeverity', $severity);  
$plugins = $client->getResponse();
```

## Specifically get plugins from a family

This method is a helper method that wraps around an internal API method. It's a shortcut for retrieving only the plugins of a specific family instead of all the plugins.

```
// Only get plugins that are in the "Backdoors" family  
$family = "Backdoors";  
  
$client->query('jobs.getPluginsByFamily', $family);  
$plugins = $client->getResponse();
```

## Get items in a specific scan profiles' special plugin list

What this method does is it gives you a list of the items in a special plugin profile that is associated with a scan profile. As you've heard, scan profiles can have a number of different plugins in them; severity, individual plugins, families, etc. Scan profiles can also have *special plugin profiles* associated with them. These plugin profiles can contain plugins such as severity groups, individual plugins, etc. This method returns all the items in a scan profile's special plugin profile.

For example, a scan profile may have the following plugins

- :plu:12345
- :sev:denial
- :spe:critical vulnerabilities

In the above case, the **:spe:** item is a special plugin profile. If you were to look at the database table associated with this special plugin profile, you may see the following

- :sev:destructive\_attack
- :plu:12523
- :plu:13423
- :plu:11232

This method will return the above four (4) items to the developer. Note that it will not try to determine what plugins are in the **:sev:** type above. If the developer wants to know that info, they can use the `getPluginsBySeverity` API call. An example of using the described API method is shown below.

```
// Profile ID as found in the profile_list database table
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

$client->query('jobs.getSpecialProfileItems', $profile_id);
$items = $client->getResponse();
```

## Reset a canceled, running, scans' status

This method will reset the status of a canceled scan after the scan has been canceled by the scan-maker. If this method is not called by the scan-maker, then the scan will remain in a permanently canceled state. This inconsistency would likely be cleared up the next time the scan is run, because the scan-maker will see the canceled bit, stop the scan, and reset the profile to be *not ready to run* and *not canceled*.

```
// Profile ID as found in the profile_list database table
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

$client->query('jobs.getSpecialProfileItems',
              _CLIENT_KEY,
              $profile_id
            );
$items = $client->getResponse();
```

## Set a scan's current progress

During the time that the scan is running, it's progress is continually being updated in the database. This method takes care of updating the status of the scan so that other software can read that progress and report it back to the end user. This method requires a profile ID to be updated as well as two (2) values; portscan progress and attack progress.

Nessus reports it's progress in two ways. A portscan is done before the actual attacks are started, and this progress is shown . This progress counter is very unreliable because the number of ports scanned cannot be determined, absolutely, ahead of time. Also, several portscans may be done during the course of a scan. This leads to further inconsistency because a portscan progress can very likely jump from 12 to 40 to 90 percent and then back down to 20 percent.

Attack progress has been observed to be more stable. Attack plugins tend to only be run once. As such, the progress back in the nessquik UI only shows attack progress. Example portscan and attack progress output are shown below.

- portscan|131.225.82.83|100|4481
- attack|131.225.82.83|2|12248

The fields in the output are pipe ( | ) delimited. The value in each field is described below

- Type of scan

- Target of the particular scan

- The current,

portscan - port being scanned

attack - plugin being run

- The total,

portscan – number of ports to scan

attack – number of plugins to use

An example of this API method being used is shown below.

```
// The profile being updated
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

// The total portscan progress, in percent.
$portscan = "50";

// The total attack progress, in percent
$attack = "3";

$client->query('jobs.setProgress',
              _CLIENT_KEY,
              $profile_id,
              $portscan,
              $attack
            );
```

## Change a scan's status

This method will change a scan's status from one status to another. You must know the current status before you can change to the new status. The status that is being set, and the status being set

from, should be one of the four (4) valid status'.

1. **N** – Note Ready to Run
2. **P** – Pending
3. **R** – Running
4. **F** – Finished

```
// The profile being updated
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

// The status you're coming from
$from = "P";

// The status you're going to
$to = "R";

$client->query('jobs.setStatus',
              _CLIENT_KEY,
              $profile_id,
              $from,
              $to
            );
```

## Set a scan's finish date

After a scan has finished running, it's finish date should probably be reported back to the database so that the web UI will update with the proper finish date and time. The format of the datetime string should be a valid MySQL datetime format. This can be accomplished using the PHP strftime function as shown in the example below.

```
// The profile being updated
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";
```

```

// Format the time correctly
$time = time();
$formatted_date = strftime("%Y-%m-%d %T", $time);

$client->query('jobs.setFinished',
              _CLIENT_KEY,
              $profile_id,
              $formatted_date
            );

```

## Save a scan report

If the results of a scan need to be saved back to the database once the scan has been run, then this API method should be used. It will add a new record to the results table that contains the output generated by Nessus. By convention, the NBE output is used, so that output is what should be saved back to the database.

I am not 100% sure if this API method can handle very large scan results. I refer to “very large” as being in the 50-70 meg range. Since nessquik has not been tested on datasets this large, I cannot say for certain, what will happen.

```

// The profile being updated
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

// Format the time correctly
$time = time();
$saved_on = strftime("%Y-%m-%d %T", $time);

// The saved results in NBE format
$results = "timestamp|12:....";

$client->query('jobs.saveReport',

```

```
        _CLIENT_KEY,  
        $profile_id,  
        $saved_on,  
        $results  
    );
```

## Email scan results to users

nessquik will, by default, always send your scan results via email (even if you also saved them to the database). This functionality is encapsulated in this method. The method will not only send the results to you, but will also send them to your list of additional recipients. The results that are sent should already be formatted as text or HTML. You have the ability to include a special subject line with the email too.

```
// The profile being updated  
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";  
  
// List of extra people to send to  
$recipients = array(  
    'joe@somewhere.com',  
    'jan@somewhere.com'  
);  
  
// Subject line to send with email  
$subject = "Nessus Scan Results";  
  
// Formatted Nessus results output  
// The $output variable in this case  
// would contain NBE that I am transforming  
// through the use of the output_html helper function  
$results = $nes->output_html($output);  
  
// Format of the email to send
```

```
$format = "html";

$client->query('jobs.emailResults',
    _CLIENT_KEY,
    $profile_id,
    $recipients,
    $subject,
    $results,
    $format
);
```

## Add entry to exemption table

### **This functionality only works in the Fermi release of nessquik**

An exemption is added for each scan that is performed by nessquik so that in the event that a user needs to request a border exemption, by virtue of scheduling a scan, the necessary pre-requisites will already be in place to fulfill the exemption. Note that currently nessquik does not check the type of scan you are running before it adds your exemption. Therefore, just scheduling a scan with a single plugin will generate an exemption. This will be addressed in a future release.

```
// The profile being updated
$profile_id = "68326b7e827cdaa105e8d0351d3fd9c1";

// The username to add the exemption for
$username = "joeuser";

// The duration, in seconds, of the scan
$duration = "600";

$client->query('jobs.addExemption',
    _CLIENT_KEY,
    $profile_id,
```

```
    $username,  
    $duration  
);
```

## ***SysOps API***

The *SysOps API* is included with nessquik so that software can be written to query the system status of nessquik clients and servers. This API is still in it's infancy. At this point in time, only a handful of methods are available. This API will be expanded in the future to include many more methods.

<i>Method Name</i>
sysops.markOffline
sysops.markOnline
sysops.whatsOnline
sysops.isOnline

## ***Using the SysOps API***

The *SysOps API* is not difficult to use. In the following examples, the IXR library for PHP will be used. For each example however, I've included an example of the XML that is returned by nessquik. This should be enough to explain how to talk the API using languages other than PHP.

## **Creating a new client object**

In PHP, before you can use the API, you need to create a client object that can talk the API. This is simple and can be accomplished with the following code. All future operations can be done through this new object.

```
// Required. Before any other API calls,  
  
// these two lines must be run  
  
require_once ( _ABSPATH.' /lib/IXR_Library.php' );  
  
$client = getIXRClient();
```

## Mark a nessquik client's scanner as being offline

There is always the possibility that a Nessus scanner can go offline for whatever reason. This call to the API will tell nessquik to mark the scanner as being offline. When a scanner is offline, it cannot have jobs scheduled on it, and the client that talks to it will not try to schedule any pending jobs on it. Finally, setting this marker will allow the nessquik web interface to show the administrator that the system is down on a remote machine.

At this point in time, this API method is not functional. It will become more relevant in the next release. To mark the Nessus scanner as offline, use the following API method.

```
// Marks the scanner as "offline"
$client->query('sysops.markOffline', _CLIENT_KEY);
```

## Mark a nessquik client's scanner as being online

If a system has been marked as offline for any reason, it will need to be set back to *online* status before it is usable again. The following API method will turn a scanner back on provided that the developer has the appropriate client key.

```
// Marks the scanner as "online"
$client->query('sysops.markOnline', _CLIENT_KEY);
```

## Typical Program Flow

In nessquik there is a general pattern of execution that all scripts take. For consistency, it's recommended that you follow this pattern.

Pages are loaded through the `index.php` page for general user related code, and the `admin.php` page for admin related code. Individual pages for each area of the site are not used at this point in time. AJAX functionality is contained in files placed in the `async` directory. Each file should be named after the area of the site where it is located, or the functionality that it provides. For instance, the asynchronous code that is called from the admin whitelist area is contained in this `whitelist.php` file located in the `async` directory.

When a request for a main page is made, the request will pass through the `index.php` page for general requests, or the `admin.php` page for admin requests. A switch statement in either page will direct a Smarty object to request a specific “main” template to be used. This template is then displayed to the user.

All subsequent calls on the page are through the AJAX interface. Functions exist in the javascript that is loaded for each page, that will query the backend for new content. The javascript functions should provide a “catch” function that will catch the output sent from the backend and insert it into the document as needed.

## Templates vs. Themes

nessquik does not use a theming concept. Templates are used instead to separate out the display code from the PHP code. Smarty is the template system that is used, due to my familiarity with it.

The template system is composed of three main components

- `lib/smarty/`
  - This directory contains all of the Smarty libraries. In addition to the classes packaged with Smarty, I have included a Smarty singleton class, `SmartyTemplate`, that should be used for instantiating a Smarty object.
- `templates/`
  - This directory contains all the of the templates used by nessquik. Templates should be named based on what they do. For example.  
  
`admin_scans.tpl`  
  
contains a template for the scans section in the admin area.
  - All templates must end with a `.tpl` extension
- `templates_c/`
  - This directory contains the Smarty cache of the templates. When a user surfs to a page, that page is generated by Smarty and stored temporarily in this directory. Subsequent calls to that page make use of the cache if it is still valid.
  - This directory must be writable by the web server

## Unit Testing

Unit testing is accomplished with the PHPUnit package. This code is available through PEAR for convenience. See the `tests` directory for the most up-to-date unit tests for nessquik.

By convention, there is a different unit test class for each class in nessquik. Classes are located in the `lib/` directory of the main nessquik installation. There are two main types of classes that are tested since there are two types of nessquik releases; General and Fermi. If a particular class is so common that it is included in both releases, then its unit test is prefixed with the word **Common\_**. Otherwise, the unit test is prefixed with the release it belongs to **Fermi\_** or **General\_**.

Each unit test can be manually run using standard PHP syntax

```
/path/to/php UnitTest
```

example:

```
/path/to/php Common_NmapTest.php
```

If the developer is interested in running all of the tests in one fell swoop, a shell script is provided that accomplishes this. There are also individual **AllTests** files that will run all of the tests in a particular category of unit tests.

In addition to the PHPUnit tests, there is interest in using the Selenium framework for testing the web UI. Selenium is an extension for Firefox that allows one to script browser usage. It's kind of like mechanize for Python. It provides an IDE-like environment where a developer can perform a single action on a page, save the results, and create a unit test from his actions.

nessquik will use this framework in the future for testing the functionality of the frontend.

## Build System

To help in managing new releases of the software and to make it easier to release specific “products” in the software, nessquik uses Phing for its build system. Phing is a PHP based build system that uses an XML file to hold instructions describing how to build your software. In addition to the XML file, it also uses the equivalent of a .ini file to store global variables that can be used in the XML configuration file.

Phing relies on two files to exist in the current working directory.

1. build.xml
2. build.properties

To build nessquik, these files must exist and be configured properly. For the most part, build.xml will not change, and you can ignore it. In the other file, build.properties, you must tweak several variables to ensure that you build nessquik (and/or any of its products) correctly. There are only a handful of variables though, and they are, for the most part, self explanatory. They are listed below for convenience.

- *nessquik.version*

The version number that will be assigned to the nessquik release

- *portscanmenow.version*

The version number that will be assigned to the portscan-me-now release

- *scanmenow.version*

The version number that will be assigned to the scan-me-now release

- *nessquikclient.version*

The version number that will be assigned to the nessquik-client release

- *nessquik.release*

Different files are deleted and moved based on the release of nessquik. This specifies which set of build instructions will be run so that you will have an installation that mimics that release.

The proper values for this variable are

- fermi
- general
- ovaltinefermi
- ovaltinegeneral

- *nessquik.product*

The type of product to build. This gives you fine grained control over which software you want to pack up. For instance, if the code for portscan-me-now is changed, there's no reason to go about rebuilding all of nessquik. You may only want to re-build portscan-me-now and package it up.

To build all the products, simply specify the value “nessquik” since the main nessquik release by default includes all the sub products.

The proper values for this variable are

- nessquik
- nessquik-client
- scanmenow
- portscanmenow

- *directory.main*

The main directory from where you want to start building nessquik. Note the the subversion repository will be checked out to this parent directory

- ex. /root/

- *directory.build*

The name you want to give to the subversion repository upon checkout.

- ex. /nessquik-build/

- *directory.products*

The directory where you want all the built products to be placed. They will be put in this directory as tarballs and will be properly labeled with a name and version number. If this directory does not exist, Phing will attempt to create it

- *svn.app*

The full path to the subversion svn binary

- *svn.repository*

The full URL to the repository that you want to check out.

Once you have tweaked the build properties sufficiently, you can run the phing command to build the particular products

```
[root@ovaltine ~]# phing
```

Phing will read the XML and properties file and will build what you specify. The files that are generated will be located in the path that you specified in *directory.products*.

In the future, the unit tests (PHPUnit), code coverage tests (PHPUnit), and generation of the API data via PHPDocumentor will likely be combined with Phing. The builds are run nightly via cron. This is to guarantee that a new packages are made available for to the community as often as possible.

## Bug Fixes, Enhancements, Patches in General

I welcome all submissions of code, but do not guarantee that any submitted code will be added to the development trunk. All submitted patches must follow the format below

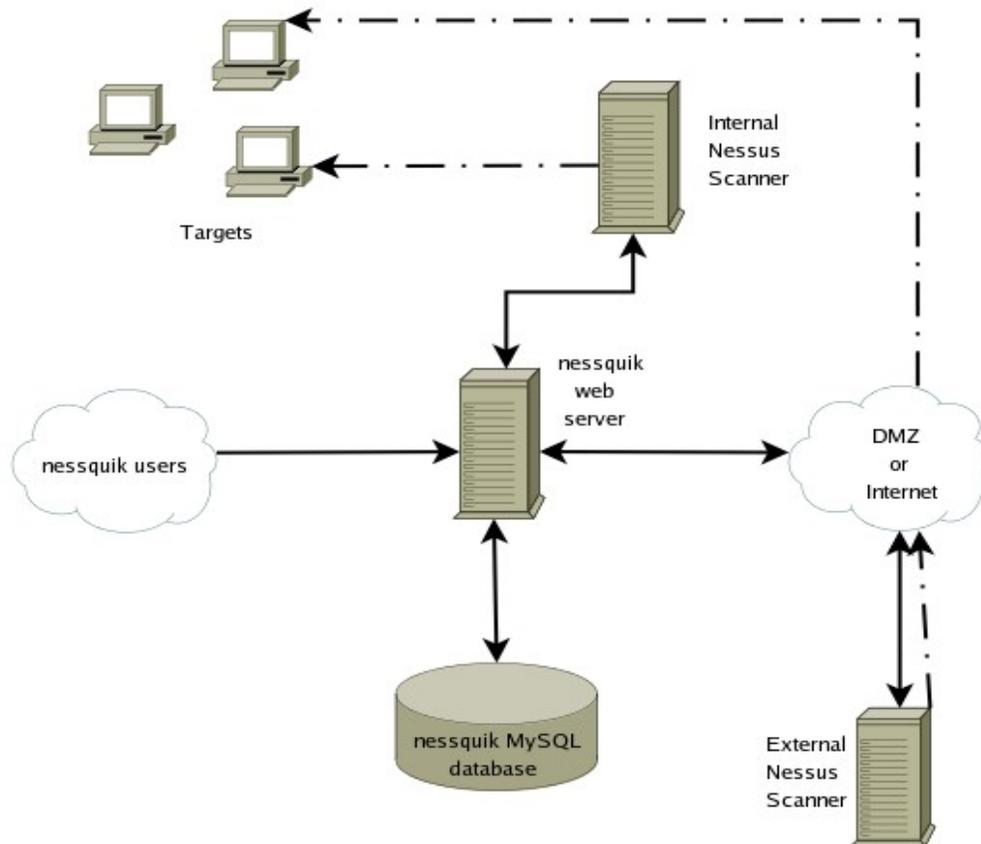
- Diff the patch against the most recent release of nessquik

- Use this diff command

```
diff -u original-source changed-source
```

- Email the patch to me as an attachment
- Include a short blurb in the email describing what the patch fixes

## Appendix A - nessquik Development Infrastructure



## Appendix B - References

I borrowed ideas from several existing coding standards documents to create this document.

- Gforge: PHP Coding Standards  
<http://gforge.org/docman/view.php/1/2/coding-standards.html>
- Edgeos Documentation for Nessus - “default” port range keyword  
[http://www.edgeos.com/nessuskb/details.php?option\\_id=76](http://www.edgeos.com/nessuskb/details.php?option_id=76)

# Appendix C - Special Plugin Profiles

