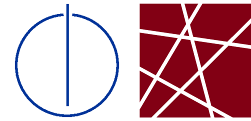
Technische Universität München
Fakultät für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste



A Penetration Testing Framework for the Munich Scientific Network

Interdisziplinäres Projekt

durchgeführt am
Lehrstuhl für Netzarchitekturen und Netzdienste
Fakultät für Informatik
Technische Universität München

von

Omar Tarabai

Aufgabensteller:	Prof. Dr.-Ing. Georg Carle
Betreuer:	Dipl.-Inform. Ralph Holz
	Dipl.-Math. Felix von Eye (LRZ)
Tag der Abgabe:	17. Januar 2014

Contents

1 Introduction **1**

1.1 Background 1

1.2 Requirements 1

1.3 Related Work 2

2 Overall Design **3**

2.1 Architecture 3

2.2 Components Overview 4

2.2.1 Server Components 4

2.2.1.1 Scheduler 4

2.2.1.2 Interface 5

2.2.2 Scanner Components 5

2.2.2.1 Agent 5

2.2.2.2 Interface 5

2.2.3 Client Interfaces 5

2.2.3.1 Command Line Interface 5

2.2.3.2 Web Interface 6

2.3 Technologies and Languages 6

3 Implementation Details **7**

3.1 Components Details 7

3.1.1 Server Components 7

3.1.1.1 Scheduler 7

3.1.1.2 Interface 8

3.1.2 Scanner Components 10

3.1.2.1 Agent 10

3.1.2.2 Interface 10

3.1.3 Client Interfaces 11

3.1.3.1 Command Line Interface 11
3.1.3.2 Web Interface 12
3.2 Configurations 12
3.2.1 Server Configuration 12
3.2.2 Scanner Configuration 13
3.2.3 Client Configuration 13
3.3 Communication Specifications 14
3.4 Permissions System 14
3.4.1 Subnet Permissions 14
3.4.2 Scan Type Permissions 14
3.4.3 Scan Permissions 15
3.5 Database 16
3.5.1 Database Schema Diagram 16
3.5.2 Description 17
3.5.2.1 Scantype 17
3.5.2.2 Subnet 17
3.5.2.3 User 17
3.5.2.4 Scan 18
3.5.2.5 Scanner 19
3.6 Implemented Scan Types 19
3.6.1 Nmap Scans 19
3.6.2 Nmap NSE 20
3.6.3 SSH Basic 20
3.6.4 HTTP Server Version 20
3.6.5 Joomla! Version 20
3.7 Scan Type Weights 20
3.8 Adding New Scan Types 21
3.9 Startup Scripts 22

4 Installation, Administration and Usage **23**

4.1 Installation 23
4.1.1 Installing Server 23
4.1.1.1 Dependencies 23
4.1.1.2 Installation Steps 23
4.1.2 Installing Scanner 24

4.1.2.1 Dependencies . . . 24
4.1.2.2 Installation Steps . . . 24
4.1.3 Installing Web Interface . . . 25
4.1.3.1 Dependencies . . . 25
4.1.3.2 Installation Steps . . . 25
4.2 Administration . . . 26
4.2.1 Add Subnet . . . 26
4.2.2 Add User . . . 26
4.2.3 Enabling a Disabled Scanner . . . 27
4.3 Usage . . . 28
4.3.1 Command Line Interface . . . 28
4.3.1.1 Configuration . . . 28
4.3.1.2 View Allowed Subnets . . . 28
4.3.1.3 View Allowed Scan Types . . . 28
4.3.1.4 Issue New Scan . . . 28
4.3.1.5 Get Summary of Recent Scans . . . 28
4.3.1.6 Cancel Scan . . . 29
4.3.1.7 Get Scan Results . . . 29
4.3.1.8 Change Password . . . 29
4.3.1.9 Output to File . . . 29

5 Evaluation **31**
5.1 MWN SSH Scan . . . 31
5.2 Joomla! Scan . . . 32

6 Future Work **35**

References **37**

1. Introduction

1.1 Background

The Leibniz Supercomputing Center (LRZ) is mainly responsible for providing computational resources and operating the network backbone of the Munich Scientific Network (MWN).

The MWN connects institutes inside the Technical University of Munich, Ludwig Maximilian University of Munich, Munich University of Applied Sciences, Weihenstephan-Triesdorf University of Applied Science and the Bavarian Academy of Sciences. Institute networks are administrated locally by each institute personnel.

Since abuse complaints received from external networks are forwarded by the LRZ to the responsible local administrators, it is desired to allow the administrators to perform regular and on-demand network/security scans of their networks to reduce risks and minimize the administrative overhead performed by the LRZ.

1.2 Requirements

The goal was to design and implement a centralized scanning framework that can perform scans on multiple targets once or periodically and save the results for later viewing while sending a notification email to the scan owner.

The following types of scans should be implemented, the framework should be easily extensible with additional scans.

- Nmap scans with different options specified
- Nmap vulnerability scan using Nmap Scripting Engine (NSE)
- Basic SSH scan
- HTTP server version detection
- Joomla! version detection

The framework should support multiple remote scanning machines, communication between the framework and the scanning machines must be secured using SSL. The performed scans should be distributed as equally as possible on the defined scanning machines.

The framework should enforce a permission system that restricts the user to scanning only specific subnets, reading scan results for specific subnets, issuing scans of specific types and limiting the minimum interval at which a user can run a periodic scan. A user of type 'Administrator' will have unlimited capabilities on the system.

The framework should be accessible to external users through both a command line interface and a web interface. Both interfaces should support issuing of new scans, viewing previous scans status and results, querying for allowed subnets and scan types and changing the user's password. The interface must restrict the user's capabilities and views according to his/her defined permissions. All communication between the interfaces and the framework must be secured using SSL.

1.3 Related Work

Other network/security scanners that perform similar functions are available, we present some of the popular ones.

Nessus [Secu] is a proprietary vulnerability scanner. It uses plugins to define scans with a database of more than 59,000 plugins that can scan for known vulnerabilities, misconfiguration, weak passwords, signs of malware, sensitive information leakages and other types.

Metasploit Framework [Rapi] is an open-source platform for developing and executing exploits. Defined exploits (known as modules) are written in ruby and are plugged into the framework. Modules are freely developed and shared by the community.

Other popular scanners include GFI Languard [GFI], a vulnerability scanner with support for patch management, and Retina [beyo], a vulnerability and risk analysis platform.

Our solution is unique because in addition to generic scanning functionality with plugins support which is implemented by other scanners such as Nessus and Metasploit, it can run on multiple remote scanning machines, enforces user permissions, allows periodic and one-time scans with notifications. Implementing a separate solution also helps against relying on a third-party for development and support of a proprietary solution.

2. Overall Design

In this chapter we describe the system architecture, design decisions and an overview of the function and technologies for each system component.

2.1 Architecture

The framework is designed in a star topology with the **Server** at the center connected to one or more **Scanner**s, the scanner does not necessarily have to be on a separate computer, one or more scanners can be deployed on the same computer as the server. The server acts as a control and command center, sending scan requests to scanner machines and periodically querying for results. For organisation and speed of access purposes, a PostgreSQL database on the server is used to store system data including scan results.

The **Scanner** is divided into two main components, the **Scanner Interface** listens for connections and commands from the server and the **Agent** runs the actual scans. For simplicity, files are used as data store.

The **Server** is divided into two main components, a **Scheduler** for core scanning functionality and an **Interface** that achieves user accessibility functionality such as issuing new scans and retrieving scan results while enforcing user permissions. A command line interface (CLI) or a web interface can be used by the user to communicate with the **Server Interface**, the user can modify the CLI for personal purposes as long as it confronts to the communication specifications accepted by the **Server Interface**.

All external communications (Client $\leftrightarrow$ Server, Server $\leftrightarrow$ Scanner) are SSL-encrypted, certificates are pre-distributed on installation for identity authentication.

The following is a high-level architecture diagram:

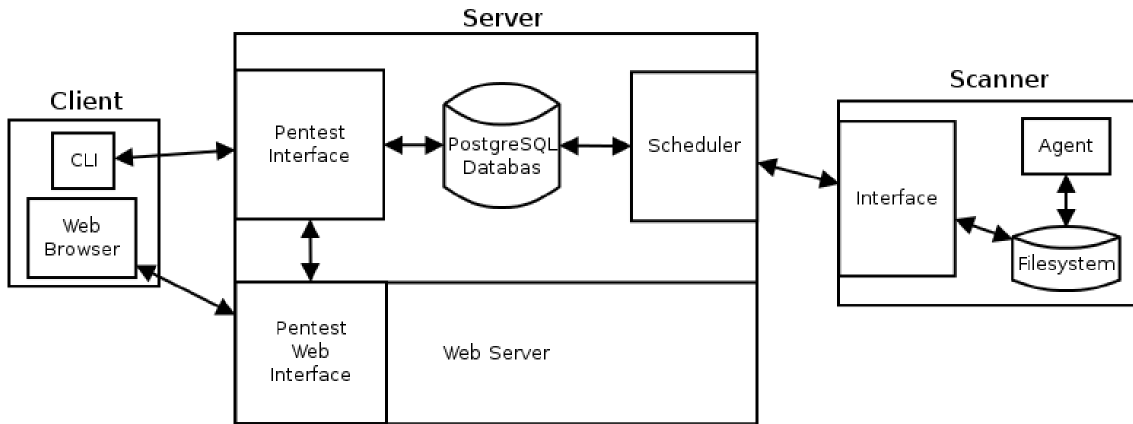


Figure 2.1: Architecture diagram

2.2 Components Overview

In this section, we present an overview of each system component, describing its main functions and the technologies/languages used. More details about the implementation of each component can be found in Component Details Section 3.1.

2.2.1 Server Components

We present an overview of the components running on the framework's central server, namely the *Scheduler* and *Interface* components.

2.2.1.1 Scheduler

The scheduler component is constantly running as a daemon. A sequence of operations is performed in series before sleeping for a period specified in the configuration file (default: 60 seconds), then repeating the same sequence again.

The first stage is querying the database for any 'cancel scan' commands issued by a user for a scan that is currently running, if exists, a corresponding 'cancel scan' command is sent to the responsible scanner.

Secondly, for scans that are currently running, the responsible scanners are queried for results, if the scanner responds with the scan results, it is written to the database, the scan is then marked as complete and an email is sent to the scan owner notifying him/her of scan completion. Only after the previous operations are successful, the scanner is instructed to delete the results.

Lastly, the scheduler checks for any scans that are due to start running. In case of a 'one-time' scan, this means that the 'scheduled time' has passed, in case of a 'periodic' scan, this means that the time resulting from adding the scan 'period' and the last 'finished on' time of the scan is less than the current time. For each of the due scans found, the next "free" scanner is chosen. If no "free" scanners available, the scanner with the earliest expected finishing time is chosen, this is calculated as a function of the currently running scans on this scanner, the number of targets per scan and the estimated runtime for this scan type per target.

In case the scheduler fails to connect to a scanner to issue a new scan or collect results, the scanner is marked in the database as 'down' and a notification email is sent to the system administrator, the scanner is never queried for operations again until the administrator sets its status back to 'up'.

2.2.1.2 Interface

The server interface component is constantly running and listening for incoming SSL connections, for each received connection, a separate thread is dispatched to handle it. The thread receives the user command, verifies the validity of the command structure, authenticates the user credentials (username/password), then runs the appropriate command handler. Results from the command handler including any error messages are sent back to the user and the connection is terminated immediately.

In case of an invalid message structure/command/arguments, an error describing the problem is returned to the user and the connection is terminated.

2.2.2 Scanner Components

We present an overview of the components running on the individual scanner machines, namely the *Agent* and *Interface* components.

2.2.2.1 Agent

The agent is responsible for executing the scans instructed by the server, similar to the server scheduler, it is constantly running, performs a sequence of operations in series then sleeps for a period specified in the configuration file (default: 60 seconds) before repeating the sequence again. All server commands are represented as files in directories defined in the configuration.

The first operation checks for any 'cancel scan' commands, if exists and the scan is running at the moment, the scan process is terminated, all data corresponding to the scan is deleted.

The second operation checks for any 'new scan' commands, if exists, a separate process is spawned to handle the scan, this allows the scan to be cancelled by killing the corresponding process without affecting the agent itself. The new process parses the scan header to retrieve the scan type and any scan parameters, it dynamically creates an object of the scan handler responsible for this scan type, this helps in extending the system with a new scan handler with minimal effort. Each scan target spawns a new thread that runs the scan handler on this target, the number of threads running at one time is limited by the 'parallelism' configuration parameter. Results of each scan thread is written to the scan results file.

2.2.2.2 Interface

The scanner interface constantly listens for incoming SSL connections from the server scheduler, it uses the locally defined SSL certificate to verify the server identity. Server commands are passed to the appropriate command handler and the result message is returned the server. The interface does not terminate the connection until its closed from the server side.

2.2.3 Client Interfaces

We present an overview of the components used by end users to access the framework's functionality, namely the *command line interface* and the *web interface*.

2.2.3.1 Command Line Interface

The CLI component implements the communication specifications required to communicate with the server interface, it provides a parameter-based interface to the user to issue commands to the server while providing user credentials. It implements all commands

supported by the user interface and uses the locally defined SSL certificate to verify the server identity.

The CLI acts as a thin client to the server interface, it does not validate user arguments in terms of input size and correctness nor verify user permissions to perform requested operations. It relies on the server interface to perform validation and verification and outputs any error messages returned by the server to the user.

2.2.3.2 Web Interface

The web interface provides an easier method of accessing the server functionality. Similar to the CLI, it implements the required communication specifications and all commands supported by the server interface. The interface can be deployed on the same computer as the server or on a separate computer, it communicates with the server interface in the same SSL-encrypted manner as the CLI. The user can use any web browser to access the web interface.

Similar to the CLI, the web interface acts as a thin client to the server interface, it does not validate user arguments in terms of input size and correctness nor verify user permissions to perform requested operations. It relies on the server interface to perform validation and verification and outputs any error messages returned by the server to the user.

2.3 Technologies and Languages

Python is the main programming language used for development, it is chosen for its ease of use, readability and suitability for developing a large-scale project. Server, scanner and interface components are developed using Python.

The web interface is built using Python-Django framework for easy integration with other Python framework components, Apache web server is used to host the web interface, it supports direct interfacing with python web pages using the *mod_python* [mpyt] module.

PostgreSQL is used as database server for its extensive features and ability to host large amounts of data with a minimum performance penalty.

3. Implementation Details

3.1 Components Details

In this section we describe in details the inner working of each system component, all file paths are relative to the system source directory.

3.1.1 Server Components

We present a detailed description of the components running on the framework's central server, namely the *Scheduler* and *Interface* components.

3.1.1.1 Scheduler

This component is defined in the file *scheduler.py*.

The scheduler is responsible for managing scans including issuing new scans, collecting results and cancelling scans. This requires establishing connections to defined scanner instances, all connections are SSL-secured and each scanner is identified by a certificate that is exchanged on installation of a new scanner, scanner certificates are added to a file that is defined in the server configuration (see Configurations Section 3.2). If any connection to a scanner fails, the scanner is marked as 'down' in the database and a notification email is sent to the system administrator, no more connections to the scanner are attempted until the administrator sets its status back to 'up'. All communications follow the specifications defined in 3.3.

On startup, the scheduler tries to parse the configuration file "server-config.ini", it then moves into the main loop which performs three main operations before sleeping for a period defined in configuration (default: 60 seconds) then repeating until the scheduler is externally terminated.

The first operation checks for any pending cancel operations, it does so by querying the database for any cancelled scans (scan.active=0) that are marked as currently running (history.status=1). For all such scans, a connection to the responsible scanner is established, a 'cancel_scan' command is sent and an 'OK' response is expected, the scan status is then set as cancelled (history.status=3). If a different response is received from the scanner, it is logged as an error and no changes are applied. The connection is terminated after one exchange.

The second operation tries to collect results for scans currently running, the database is queried for running scans (history.status=1), for all such scans, a connection to the responsible scanner is established and a 'get_result' command and the scan-id are sent. The response is read line-by-line with each line corresponding to one target results, if the first line contains only the value '0', this signifies that results are not ready, in this case the connection is terminated and the scan is skipped. Otherwise, results are written to database, the scan status is set to 'completed' (history.status=2). A 'delete_result' command and the scan-id are sent to the scanner, the connection is terminated and a notification email is sent to the scan owner.

The final operation is checking for new scans to perform, it starts by querying the database for one-time (scheduling_method=1) with no history records (no previous runs) and the 'scheduled_time' less than the current time, the results are sorted ascending by 'scheduled_time' to preserve the order of running. If no pending 'one-time' scans found, the database is queried for 'periodic' (scheduling_method=2) scans where the current time is between 'valid_after' and 'valid_before' values and no history records (no previous runs). If no such scans pending, the database is queried for 'periodic' scans where the current time is between 'valid_after' and 'valid_before' values and the last running time (history.started_on) added to the scan interval (scan.period) is less than the current time.

For each pending scan found, a new scanner is chosen from the scanner pool, the first choice is any idle scanner available, if all scanners are busy, the one with the earliest expected finishing time is chosen, this is calculated using the following equation:

```
SCANNER_AVAIL = MAX(SCAN_START_TIME + SCAN_WEIGHT * TARGETS_COUNT)
```

The maximum is calculated over all scans that are currently assigned to each scanner and the scanner with the smallest SCANNER_AVAIL is picked (see Scan Types Weights Section 3.7).

A connection is then established to the chosen scanner and 'new_scan' command and the scan metadata are sent, the targets are then sent one per line and at the end, an 'OK' response is expected from the scanner. If 'OK' received, the scan history is updated and the connection terminated, otherwise, the received message is logged as error and no changes are applied.

3.1.1.2 Interface

This component is defined in the file *interface.py*.

On startup, the interface tries to parse the configuration file "server-config.ini" (see Configurations Section 3.2), opens a socket listening on all machine addresses and port number defined in configuration. A received connection dispatches a new handler thread (we will call it connection-handler) and the main thread goes back to accepting more connections.

The connection-handler starts by creating an SSL socket wrapper for the received connection, it acts as the server side and uses the server certificate and key files defined in configuration. TLSv1 is used as the encryption protocol. All exchanged messages uses JSON and the communication specifications defined in 3.3, if a format error in a received message is detected, the connection is terminated immediately.

The connection-handler waits for a message from the client, the message is then passed to a command handler instance, which is expected to return either a response message or an error message, this is then encoded as per the communication specifications and passed back to the client over the established channel and the connection is terminated.

The received message is expected to be in the form of one or more (key,value) pairs, one of the keys should be a valid command, its value depends on the command itself and can be irrelevant in some cases. Some commands requires other (key,value) pairs to be present in the message, this is validated and an error message is returned otherwise. The expected data type is validated as well.

The command-handler first checks for a valid username and password in the message by querying the database for the given username and the SHA512 hash of the given password, an error message is returned otherwise. Then it searches for a valid command in the user message and handles it, if none is found, an error message is returned.

The **allowed_subnets** command queries the database for the list of subnets the user is allowed to access (readonly or read/write) and returns them. If the user is an administrator, all subnets defined in the database are returned.

The **change_password** command is expected to contain a username value, a **new_password** key is also expected. If the current user is an administrator, the password can be changed for any username, otherwise, the username should be the same as the current user.

The **recent_scans** command is expected to contain an integer value between 0 and 50 as the number of scans to return, the database is queried for the most recently executed scans that the user is allowed to view (see Permissions System 3.4). Only scan information and history are returned, individual scan results can be retrieved using the **result** command and the scan-id returned here.

The **target_count** command is expected to contain a valid scan-id. The current user is checked for at least readonly permissions to the given scan. If so, the database is queried for the number of targets for the specified scan and is returned to the user. If not, an error message is returned.

The **result** command is expected to contain a valid scan-id. The current user is checked for at least readonly permissions to the given scan. If so, the database is queried for the results of the specified scan and is returned to the user. If the user is allowed access to only a subset of the scan targets, only the results of these targets are returned.

The **cancel** command is expected to contain a valid scan-id. Only an administrator or the issuer of the scan are allowed to perform this action. The current status of the scan is queried, if it has already been cancelled (active=0) or not currently running, the scan is deleted entirely, otherwise, the active flag is set to 0.

The **allowed_scan_types** command queries the database for the list of scan types the current user is allowed to run, it returns them along with other scan-type permission values (see Permissions System 3.4).

The **new** command is used to issue a new scan. It starts by checking for the existence of other mandatory arguments (**target**, **scan_type** and (**once** or **periodic**)).

It prepares the list of targets by validating the format of the target list, resolves any hostnames, checks that the current user has enough permissions to scan the given targets and expands any subnets to individual IP addresses.

If the **once** argument is present, it is expected to contain a valid timestamp which is the running time of the scan. If the **periodic** argument is present, it is expected to contain a value in the format of an integer (between 0 and 999) followed by 'd', 'h' or 'm' corresponding to 'day', 'hour' and 'month' respectively. In case of **periodic** scheduling, two more optional arguments can be present, **from** containing a valid timestamp which is the start datetime for running the scan, if not found, the current datetime is used, and **to**

containing a valid timestamp which is the expiry datetime for the scan, if not found, it is set for a year after the current datetime.

The **scantype** argument is checked for a valid scantype, user permissions are queried to ensure that the user has access to this scantype, and in case of a periodic scan, that the user can run this scan periodically with the specified period (see Permissions System 3.4).

At this point, the scan is ready to be added, a new scan record and a record for each scan target are inserted into the database, a ”New scan issued” message is returned.

3.1.2 Scanner Components

We present a detailed description of the components running on the individual scanner machines, namely the *Agent* and *Interface* components.

3.1.2.1 Agent

The agent is defined in the file *scanner/agent.py*.

On startup, the agent tries to parse the configuration file ”scanner-config.ini” (see Configurations Section 3.2), it then moves into the main loop which performs two main operations before sleeping for a period defined in configuration (default: 60 seconds) then repeating until the agent is externally terminated.

The first operation scans the ’cancel’ directory (defined in configuration) for ’cancel_scan’ commands, which are represented by empty files with the name as the scan-id to cancel. If found, the agent searches for the given scan-id in an inner dictionary of (scan-id, process-instance) that contains the currently running scans and terminates the corresponding process. The agent then removes all traces of the scan from the ’scans’ and ’results’ directories and the running-scans dictionary.

The second operation checks for new scans to run in the ’scans’ directory, from the sorted (by scan-id) list of scans, it retrieves the next scan that is not currently running (not in running-scans dictionary). The first line in the file is read, it corresponds to the scan metadata as key-value pairs, a new process is spawned to handle the new scan and an entry in the running-scans dictionary is created for the scan.

The new process creates an instance of the specific scan handler according to the scan-type found in the scan metadata then creates a new results file with its name as the scan-id in the ’results’ directory. The scan file is then read line-by-line with each line corresponding to a single target definition, for each target a new thread is spawned to run the scan handler on it, the number of threads running at the same time is limited by a ’parallelism’ parameter defined in the configuration. To avoid problems arising from multiple threads writing results to the same file simultaneously, each thread writes the results of the scan to a thread-safe queue structure, the main thread reads results from this structure and writes to the results file. When all targets are scanned successfully, the scan file is removed from the ’scans’ directory.

3.1.2.2 Interface

The scanner interface is defined in the file *scanner/scanner-interface.py*.

On startup, the interface tries to parse the configuration file ”scanner-config.ini” (see Configurations Section 3.2), opens a socket listening on all machine addresses and port number defined in configuration. The connection is SSL-wrapped and the scanner interface uses its key and certificate files defined in configuration, it uses the server certificate file setup on installation to verify incoming connections since only the server is allowed to establish a connection to the scanner.

Once a connection is received from the server, the scanner interface waits for a command, processes it and sends a response back then waits for another command and so on until the connection is terminated from the server side. All exchanged messages are encoded according to the communication specifications (see 3.3).

The **new_scan** command creates a new file in the 'scans' directory (path defined in configuration), the file name is the scan-id, content is written as received from server, it is expected that the first line contains the scan metadata and each subsequent line contains one target definition. On success, 'OK' message is returned.

The **get_result** command expects an 'id' item containing the scan-id being requested. If a file exists with its name as the scan-id in the 'results' directory (defined in configuration) and no file exists with the same name in the 'scans' directory, this signifies that the scan is complete, the contents of the results file are sent back to the server, it is expected to contain the target results one per line. If the scan is not yet finished, '0' is returned instead.

The **delete_result** command expects an 'id' item containing the scan-id being deleted. The file with the name as the scan-id in the 'results' directory is deleted and an 'OK' response is returned.

The **cancel_scan** command expects an 'id' item containing the scan-id being cancelled. An empty file with its name as the scan-id is created in the 'cancel' directory (defined in configuration) and an 'OK' response is returned.

The **get_pending_scans** command reads the 'scans' directory and returns a sorted list of scan-ids that are not yet complete.

3.1.3 Client Interfaces

We present a detailed description of the components used by end users to access the framework's functionality, namely the *Command line interface* and the *Web interface*.

3.1.3.1 Command Line Interface

The CLI is defined in the *pentest-cli.py* file.

On startup, the CLI tries to parse the configuration file "client-config.ini" (see Configurations Section 3.2), commandline arguments supplied by the user are then parsed into a dictionary structure (see CLI Usage section 4.3.1 for description of supported arguments).

For collecting user credentials, the environment variables 'LRZSCANUSER' and 'LRZSCANPWD' corresponding to username and password respectively are used, if not defined, the user is prompted to enter his/her username and password.

In case the user requests that results are written to an output file instead of stdout (–output-file argument), the specified filename is opened and the output-file argument is removed from the arguments dictionary.

If the user requests a password change, he/she is prompted for the new password and password confirmation which are checked for equality.

The arguments dictionary is prepared for sending by removing any keys with empty values, converting datetime values to timestamps and adding user credentials. The dictionary is then encoded according to the communication specifications (see 3.3), an SSL-encrypted connection is opened to the server (server host/port and certificate file specified in configuration) and the encoded dictionary is sent over the established connection. The CLI then waits for a response from the server and writes it to stdout or output file if requested so by the user.

3.1.3.2 Web Interface

The web interface is programmed using python-django framework [Foun], definition under the *pentest_www/* directory.

The web interface acts as a thin-client to the server interface with no defined database of its own, it performs the same functionality as the CLI with the adition of two main features. It provides a virtual 'login' where the user credentials are stored in a session variable on the web server and sent to the framework server with each request. It also provides the option to download scan results as a .tar.gz archive with results separated into files, one for each target.

The website skeleton is created automatically by the django framework which follows the Model-View-Controller (MVC) architecture.

In the *pentest_www/pentest/views.py* file, we define a view for each page as a single function, each view performs login validations, prepares the user message, passes it to an instance of a 'Client' model (defined in *pentest_www/pentest/models.py*) which communicates the message to the server (server host/port and certificate file specified in configuration file: *pentest_www/pentest/client-config.ini*) and sends the response back to the view which presents it to the user appropriately.

The views use HTML templates defined in *pentest_www/pentest/templates/* directory.

3.2 Configurations

In this section we describe the configuration parameters used by server, scanner and client components.

3.2.1 Server Configuration

Server configuration is defined in *server-config.ini*.

It is divided into three sections: **interface**, **scheduler** and **default**. Following is a description of each configuration parameter.

Parameter Name	Description
interface.port	port number that the server interface listens on
interface.max_msg_len	maximum message length accepted by the server interface from a client to protect against malicious clients
interface.logfile	path to the interface log file
interface.loglevel	interface logging level (1 - DEBUG , 2 - INFO , 3 - WARNING , 4 - ERROR)

Parameter Name	Description
scheduler.sleep_time	time in seconds that the scheduler sleeps for between performing its regular operations
scheduler.logfile	path to the scheduler log file
scheduler.loglevel	scheduler logging level (1 - DEBUG , 2 - INFO , 3 - WARNING , 4 - ERROR)

Parameter Name	Description
default.server_certfile	path to the server certificate file
default.server_keyfile	path to the server private key file
default.scanners_certfile	path to the file containing defined scanners' certificates
default.admin_email	administrator email address
default.db_name	name of PostgreSQL database instance
default.db_user	username for accessing the database
default.db_pass	password for accessing the database

3.2.2 Scanner Configuration

Server configuration is defined in *scanner/scanner-config.ini*.

It is divided into three sections: **interface**, **agent** and **default**. Following is a description of each configuration parameter.

Parameter Name	Description
interface.keyfile	path to the scanner private key file
interface.certfile	path to the scanner certificate file
interface.server_cert	path the server certificate file
interface.port	port number that the scanner interface listens on
interface.logfile	path to the interface log file
interface.loglevel	interface logging level (1 - DEBUG , 2 - INFO , 3 - WARNING , 4 - ERROR)

Parameter Name	Description
agent.sleep_time	time in seconds that the agent sleeps for between performing its regular operations
agent.logfile	path to the agent log file
agent.parallelism	defines the number of targets to scan in parallel
agent.loglevel	agent logging level (1 - DEBUG , 2 - INFO , 3 - WARNING , 4 - ERROR)

Parameter Name	Description
default.scans_dir	path to the directory that stores scan commands
default.results_dir	path to the directory that stores scan results
default.cancel_dir	path to the directory that stores cancel commands

3.2.3 Client Configuration

Required by the command line interface and web interface clients, it is defined in *client-config.ini* which should be in the same directory as *pentest-cli.py* for the CLI and under *pentest_www/pentest/* for the web interface.

It contains one section: **default**. Following is a description of each configuration parameter.

Parameter Name	Description
default.server_host	hostname or IP address of server to connect to
default.server_port	port number that the server interface is listening on
default.server_certfile	path to the server certificate file

3.3 Communication Specifications

All communications between system components follow the same specifications.

If one side requires a message to be sent, firstly the message should be encoded using JSON, the length of the resulting JSON-encoded message is sent followed by a new line then the JSON-encoded message is sent. The receiving end will parse the first line to know the size of the message to expect.

In some cases, the message size is unknown or difficult to calculate beforehand. For example, when the server is reading scan results from the database and sending them to the client, it is infeasible for reasons of memory constraints to read the results of a large scan and calculate its size before sending. In such cases a length of -1 can be sent instead. The receiving side will keep accepting data until an empty line is detected which signifies end-of-message. This is only accepted by the receiving side if the server is the sending side since it is the only side trusted to not act maliciously.

3.4 Permissions System

Users are divided into two types:

1. Admin: Full-access to all subnets, scan types and issued scans.

2. Normal User: Permissions have to be explicitly set.

3.4.1 Subnet Permissions

User can be assigned permission for a specific network subnet. The user can read results of any scan performed on hosts within the specified subnet (in case the scan spans multiple subnets, the user will only see results for subnets he/she is assigned permissions for) and run his/her own scans on targets within the specified subnet.

If the **readonly** flag is set, the user is not allowed to perform new scans on targets within the specified subnet.

3.4.2 Scan Type Permissions

User can be allowed to issue a scan type with the following parameters defined:

- periodically_allowed flag: if set, the user can perform a periodic scan of this scan type.

- min_periodic_time: in case the periodically_allowed flag is set, this puts a limit on the minimum interval for the issued periodic scan (e.g. 1d, 12h..etc).

3.4.3 Scan Permissions

Scan permissions are not explicitly set but are inferred from the user's subnet permissions. Two types of permissions are possible per a pair (user, scan). **Full** permissions means the user can read all scan results and cancel/delete scan, this applies only for users of type 'Administrator' and the scan issuer. **Read** permissions means the user can read all or part of the scan results depending on the scan target subnets and the user's subnet permissions, for example, a scan targeting '192.168.0.1' and '10.0.0.1', a user A can only read results for the target '192.168.0.1' if he has read permissions for subnet '192.168.0.0/16' but no permissions for subnet '10.0.0.0/8'.

3.5 Database

The server uses a PostgreSQL database for data storage, in this section we present the database schema diagram and a description of each database table grouped by logical entities.

3.5.1 Database Schema Diagram

The following is the database schema diagram showing table fields and foreign key relations between tables. Primary key fields are underlined.

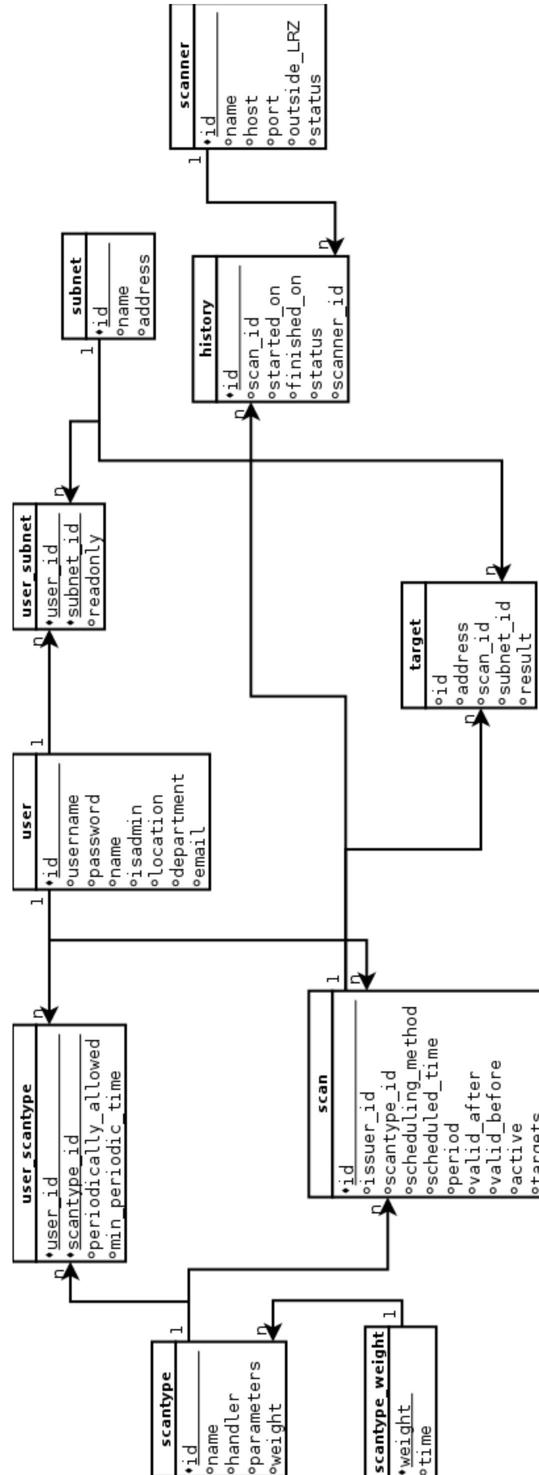


Figure 3.1: Database diagram

3.5.2 Description

The following is a description of each logical entity stored in the database and the underlying database tables.

3.5.2.1 Scantype

Describes the types of scans supported by the framework. Scantype information are stored in two different tables, table *scantype* stores core scantype information.

Column	Type	Description
id	integer	Primary key
name	varchar	Name assigned to scan type
handler	varchar	Name of the python handler class
parameters	varchar	General parameters that can be passed to handler
weight	integer	Foreign key referencing scantype_weight(id)

scantype table references *scantype_weight* table which stores possible weight classes (see Scan Type Weights section 3.7).

Column	Type	Description
weight	integer	Primary key
time	interval	Estimated time of this weight class

3.5.2.2 Subnet

Describes network subnets that can be scanned by the framework, subnet information are stored in one table: *subnet*.

Column	Type	Description
id	integer	Primary key
name	varchar	Name assigned to the subnet
address	varchar	Address in CIDR notation

3.5.2.3 User

Describes individual users configured to use the framework and the user permissions. User information are stored in three different tables. Table *user* stores core user information.

Column	Type	Description
id	integer	Primary key
username	varchar	Username used for login
password	varchar	SHA512 encrypted password
name	varchar	User's full name
isadmin	integer	Flag (is user an administrator?)
location	varchar	User's address
department	varchar	User's department or institute within the MWN
email	varchar	User's email address

Table *user_scantype* stores information about user permissions in regard to supported scantypes.

Column	Type	Description
user_id	integer	Foreign key referencing user(id)
scantype_id	integer	Foreign key referencing scantype(id)
periodically_allowed	integer	Flag
min_periodic_time	varchar	Minimum periodic interval allowed

Table *user_subnet* stores information about user permissions in regard to supported subnets.

Column	Type	Description
user_id	integer	Foreign key referencing user(id)
subnet_id	integer	Foreign key referencing subnet(id)
readonly	integer	Flag

3.5.2.4 Scan

Describes instances of scans performed by the framework. Scan information are stored in three different tables. Table *scan* stores main scan information as supplied by the user.

Column	Type	Description
id	integer	Primary key
issuer_id	integer	Foreign key referencing user(id)
scantype_id	integer	Foreign key referencing scantype(id)
scheduling_method	integer	1: once, 2: periodic
scheduled_time	double	Timestamp for when to run (one-time scans)
period	interval	In case of periodic scan
valid_after	double	Timestamp for when to start doing periodic scan
valid_before	double	Timestamp for when to stop doing periodic scan
active	integer	0 or 1
targets	varchar	List of comma-seperated targets

Table *target* stores individual target results for each scan.

Column	Type	Description
id	integer	Primary key
address	varchar	Target hostname or IP address
scan_id	integer	Foreign key referencing scan(id)
subnet_id	integer	Foreign key referencing subnet(id)
result	text	Result text for scanning for this target

Table *history* stores information about the execution history of each scan.

Column	Type	Description
id	integer	Primary key
scan_id	integer	Foreign key referencing scan(id)
started_on	double	Timestamp of start datetime
finished_on	double	Timestamp of end datetime
status	integer	1: started, 2: finished, 3: canceled
scanner_id	integer	Foreign key referencing scanner(id)

3.5.2.5 Scanner

Describes configured scanner machines. Information are stored in one table: *scanner*.

Column	Type	Description
id	integer	Primary key
name	varchar	Name assigned to scanner
host	varchar	Hostname used to connect to scanner
port	integer	Port that the scanner interface is listening on
outside_LRZ	integer	Flag
status	integer	0: Down, 1: Up

3.6 Implemented Scan Types

The following scan types are already implemented into the project. All scans are and should be defined inside the *scanner/handlers/* directory.

3.6.1 Nmap Scans

Nmap scans are defined in the file: *nmap_handler.py*.

It performs regular nmap scans with different command line parameters, therefore, nmap should be installed on all scanner machines.

Three different nmap scans are defined, each with different scan parameters. Since they all use the same scan handler, more can be easily added by adding a record in the 'scantype' database table with the same values but different parameters.

- Nmap quick: -Pn -sS -T4 -F
- Nmap normal: -Pn -sS -sV
- Nmap full: -Pn -sS -p1-65535 -A

Following is a description of each parameter used, taken from [Nmap].

- -Pn: Treat all hosts as online – skip host discovery
- -sS: TCP SYN scan
- -T<0-5>: Set timing template (higher is faster)
- -F: Fast mode - Scan fewer ports than the default scan
- -sV: Probe open ports to determine service/version info
- -p <port ranges>: Only scan specified ports
- -A: Enable OS detection, version detection, script scanning, and traceroute

3.6.2 Nmap NSE

Defined in the file: *nmap_nse_handler.py*.

It uses Nmap Scripting Engine (NSE) to do a basic vulnerability scan of open parts detected using nmap. The scan scripts are defined in the *scanner/nmap-nse/* directory.

3.6.3 SSH Basic

Defined in the file: *ssh_handler.py*. Modified OpenSSH client under *scanner/ssh/*

This scanner tries to connect to the default SSH port (if no port number supplied by user). Based on the SSH scanner developed by Gasser et al.[GaHC14], it uses a slightly modified OpenSSH client to perform and log a complete SSH handshake with the target, the OpenSSH client is modified to log SSH host keys used by the server. Useful information collected include server and protocol versions, supported cipher suites and public keys.

3.6.4 HTTP Server Version

Defined in the file: *httpversion_handler.py*.

Sends a HEAD request to a HTTP server on the default HTTP port (if no port number supplied by user) and returns the value of the 'Server' header which by default contains the HTTP server version.

3.6.5 Joomla! Version

Defined in the file: *jversion_handler.py*.

Connects to the HTTP server and performs a series of defined checks to try and detect the installed Joomla! version. Certain files served by the Joomla! website are searched for an occurrence of a specific string or regular expression. For regular expression searches (2,3 and 4), the value between brackets is matched as the version number. These checks are not guaranteed to be correct.

The following is the list of checks performed:

File	String/Regex	Joomla! version
/	'Joomla! - Copyright'	1.0
/	'Joomla! (1.[567])'	1.[567]
/administrator/manifests/files/joomla.xml	'<version>([0-9\.]*)</version>'	[0-9\.]*
/language/en-GB/en-GB.xml	'<version>([0-9\.]*)</version>'	[0-9\.]*
/language/en-GB/en-GB.ini	'# $Id: en-GB.ini 11391 2009-01-04 13:35:50Z ian $'	1.5.26
/language/en-GB/en-GB.ini	'; $Id: en-GB.ini 20196 2011-01-09 02:40:25Z ian $'	1.6.0
/language/en-GB/en-GB.ini	'; $Id: en-GB.ini 20990 2011-03-18 16:42:30Z infograf768 $'	1.6.5 - 1.7.1
/language/en-GB/en-GB.ini	'; $Id: en-GB.ini 22183 2011-09-30 09:04:32Z infograf768 $'	1.7.3 - 1.7.5

3.7 Scan Type Weights

Each scan type created is assigned a weight class, a weight class defines the expected running time of the scan for one target. This is used to estimate the time at which a scanner machine will finish its current assigned scans to maximize load balancing.
The following weight classes are defined by default:

Weight	Time
1	1 second
2	1 minute
3	10 minutes
4	30 minutes
5	1 hour
6	12 hours
7	1 day
8	7 days
9	1 month
10	1 year

3.8 Adding New Scan Types

The framework is designed to be easily extensible by adding new scan types.
To do that, two main steps need to be done:

1. Write the scan handler class:
   - Under the 'scanner/handler' directory, create a new python file. The naming convention used is 'XXX_handler.py'
   - The class needs to have the following method defined:

```
# target: one target ip/hostname
# port: port number if supplied by user, ignore if not applicable
# parameters: additional parameters defined in server database,
    ignore if not applicable
# Returns: scan result string
def run(self, target, port, parameters):
```

2. Add scan type to server database:
   - Add a new record to the 'scantype' table:

```
INSERT INTO scantype (name, handler, parameters, weight)
VALUES
(
'<name>',
'<handler>',
'<parameters>',
<weight>
);

<name>: used by users when creating new scans
<handler>: name of the handler class created in the previous
    step without the '.py'
<parameters>: parameters that will be passed to the handler,
    this enables the user to create multiple scan types that used
    the same handler but with different parameters
<weight>: expected weight class (see 'Scan Type Weights' section
    for more details)
```

3.9 Startup Scripts

Both server and scanner installations can be started using defined bash startup scripts, server script in *bin/server* and scanner script in *scanner/bin/scanner*. Both scripts accept one the following arguments: start, stop, restart. They can be used in combination with crontab to ensure that the server/scanner is always running.

Both scripts use the start-stop-daemon utility to start/stop its components, component PID's are stored in files under the same directory as the script.

4. Installation, Administration and Usage

In this chapter we present detailed installation steps for all framework components, description of possible administrative operations and end-user's usage instructions.

4.1 Installation

We describe the installation procedure for all system components, all steps were tested on a Debian 7.0 system. The user is assumed to have shell access and root permissions on the target system.

4.1.1 Installing Server

We list the dependencies and installation steps for server components.

4.1.1.1 Dependencies

- python
- postgresql
- python-psycopg2
- Python package: netaddr

4.1.1.2 Installation Steps

PostgreSQL database installation:

```
cd lrz_pent/
sudo -u postgres createuser -s pentest
# Replace password in the next command appropriately.
sudo -u postgres psql -c "ALTER USER pentest WITH PASSWORD 'pentest'"
sudo -u postgres psql -c "CREATE DATABASE pentest WITH OWNER = pentest"
echo -e "local \t pentest \t pentest \t\t md5" | sudo cat -
   /etc/postgresql/9.1/main/pg_hba.conf > temp && sudo mv temp
   /etc/postgresql/9.1/main/pg_hba.conf
sudo /etc/init.d/postgresql restart
psql -U pentest -d pentest -f db/schema.sql
psql -U pentest -d pentest -f db/data.sql
```

Generate server keys:

```
cd auth/
touch scanners.crt
openssl req -x509 -newkey rsa:2048 -keyout self.key -out self.crt -nodes
cd ..
editor server-config.ini
# Change the configurations accordingly, most importantly:
    server_certfile, server_keyfile, admin_email
```

At this point, the server is installed but not running. You can set it to run automatically using crontab:

```
sudo crontab -e
```

and adding the following to the crontab file:

```
# Replace /<path>/<to>/lrz_pent appropriately
* * * * * /<path>/<to>/lrz_pent/bin/server start
```

Note that some scans such as Nmap with operating system detection need to run as root, we are using sudo crontab.

4.1.2 Installing Scanner

The scanner can be installed on the same machine as the server, the framework requires at least one scanner installed to function correctly. Same instructions can be followed for each scanner machine required.

4.1.2.1 Dependencies

- python
- nmap

4.1.2.2 Installation Steps

Generate scanner keys:

```
cd scanner/auth/
openssl req -x509 -newkey rsa:2048 -keyout scanner.key -out scanner.crt -
    nodes
cd ..
```

At this point, the server and scanner need to exchange certificates.
Copy the scanner public key to the server and into the 'auth/scanners.crt' file.
Copy the server public key to the scanner auth folder. If the scanner is on the same machine as the server:

```
cat auth/scanner.crt >> ../auth/scanners.crt
```

Change values in scanner-config.ini accordingly.

Configure the scanner to start automatically using crontab:

```
sudo crontab -e
```

Then add the following:

```
# Replace /<path>/<to>/lrz_pent appropriately
* * * * * /<path>/<to>/lrz_pent/scanner/bin/scanner start
```

On the server, connect to the database by running:

```
psql -U pentest -d pentest
```

Then issue the following SQL statement:

```
INSERT INTO scanner (name, host, port, "outside_LRZ")
VALUES
(
'<name>',
'<host>',
'<port>',
'0|1'
);
```

Replace values appropriately, the port number is the one configured in the scanner configuration file.

4.1.3 Installing Web Interface

You can choose to install the web interface on the server, it is optional since all the main functionality are provided by the CLI.

4.1.3.1 Dependencies

- django
- apache2
- libapache2-mod-wsgi

4.1.3.2 Installation Steps

Apache site configuration:

```
sudo editor /etc/apache2/sites-enabled/pentest
```

Add the following to the file:

```
WSGIScriptAlias /pentest
    /<path>/<to>/lrz_pent/pentest_www/pentest_www/wsgi.py
WSGIPythonPath /<path>/<to>/lrz_pent/pentest_www

<Directory /<path>/<to>/lrz_pent/pentest_www/pentest_www>
<Files wsgi.py>
Order deny,allow
Allow from all
</Files>
</Directory>
```

Replace '/pentest' with the path you HTTP path you wish to use for the web interface.
Replace '/path/to/lrz_pent' with the correct path.
Copy cert file to web interface authentication folder:

```
cp auth/self.crt pentest_www/pentest/auth/
```

Restart apache:

```
sudo /etc/init.d/apache2 restart
```

4.2 Administration

Most administration tasks will require performing SQL statements directly on the database. To first connect to the postgresql database, issue the following command on the server:

```
pentest -U pentest -d pentest
```

When prompted for the password, enter the password defined in 4.1.1.

4.2.1 Add Subnet

Run the following SQL statement on database:

```
INSERT INTO subnet (name, address)
VALUES (
'<name>',
'<address>'
);
```

Replace the values accordingly.
'name' is symbolic and is not used for address resolution.
'address' should be a valid IP network (e.g. 127.0.0.1, 192.168.0.0/16)

4.2.2 Add User

1. Generate password hash:
   Run the following command on a system with python installed:

   ```
   python -c "import hashlib; print
       hashlib.sha512('<pass>').hexdigest()"
   ```

2. Add user record:
   Run the following SQL statement on the database:

   ```
   INSERT INTO public.user (username, password, name, isadmin,
       location, department, email)
   VALUES
   (
   '<username>',
   '<pass-hash>',
   '<name>',
   '0|1',
   '<location>',
   '<department>',
   '<email>'
   ); SELECT LASTVAL();
   ```

Replace the statement values with the appropriate values.
(location, department, email) are optional, can be replaced with NULL (without any quotes).
The statement will output the ID of the new user, take note of it for future operations.

3. Assign scan type permissions:
This step is not necessary if the user is an administrator (i.e. 'isadmin' flag set).
Run the following SQL statement on the database to get all available scan types:

```
SELECT * FROM scantype;
```

Note the IDs of the scan types you want to assign to the new user, then run the following SQL statement for each scan type:

```
INSERT INTO user_scantype (user_id, scantype_id,
    periodically_allowed, min_periodic_time)
VALUES
(
'<user_id>',
'<scantype_id>',
'0|1',
'<min_periodic_time>'
);
```

Replace values accordingly.
For the min_periodic_time, '0' is a valid value, otherwise specify the timing in terms of minutes, hours or days (e.g. 30m, 8h, 1d).
If the user is not allowed periodic scanning (i.e. periodically_allowed = 0), you can specify an empty value for min_periodic_time, but not NULL.

4. Assign subnet permissions:
This step is not necessary if the user is an administrator (i.e. 'isadmin' flag set), but a subnet has to exist in database for admin to be able to scan it.
Run the following SQL statement on the database to get all available subnets:

```
SELECT * FROM subnet;
```

Note the IDs of the subnets you want to assign to the new user, then run the following SQL statement for each subnet:

```
INSERT INTO user_subnet (user_id, subnet_id, readonly)
VALUES
(
'<user_id>',
'<subnet_id>',
'0|1'
);
```

Replace values accordingly.

4.2.3 Enabling a Disabled Scanner

In the case that the server is unable to connect to a scanner, the scanner is marked as 'down' in the database and an email is sent to the administrator (email address defined in server configuration file).
To re-enable the scanner after the issue has been fixed:

```
UPDATE scanner SET status = 1 WHERE host = '<hostname>';
```

4.3 Usage

In this section we describe usage instructions for end-users, users are created by an administrator by following the instructions in 4.2.2.

4.3.1 Command Line Interface

pentest-cli.py is a command line interface used to interact with the Pentest server.
Use

```
python pentest-cli -h
```

to view command details.

All CLI commands will prompt for a user/pass before communicating with the server.
To prevent the CLI from prompting for user/pass with each command, you can set the following environment variables with the user/pass:

```
export LRZSCANUSER='<username>'
export LRZSCANPWD='<password>'
```

4.3.1.1 Configuration

To configure the CLI to run correctly, you will need to have a copy of the server certificate file and the configuration file 'client-config.ini'.
Change the values in the configuration file to point to the correct server:port and the location of the server cert file.

4.3.1.2 View Allowed Subnets

```
python pentest-cli.py -A
```

4.3.1.3 View Allowed Scan Types

```
python pentest-cli.py -S
```

4.3.1.4 Issue New Scan

Some examples of a new scan command:

```
# Quick nmap scan of localhost every one day
python pentest-cli.py -n -t 'localhost' -s 'nmap-quick' -p 1d
# Full nmap scan of two IP addresses once
python pentest-cli.py -n -t '192.168.0.10,192.168.0.11' -s 'nmap-full' -o
```

4.3.1.5 Get Summary of Recent Scans

```
python pentest-cli.py -R
python pentest-cli.py -R 3 # Gets only the latest 3 scans
```

4.3.1.6 Cancel Scan

```
python pentest-cli.py -c 10 # Where 10 is the scan-id, retrieved with -R
```

4.3.1.7 Get Scan Results

```
python pentest-cli.py -r 10 # Where 10 is the scan-id, retrieved with -R
```

4.3.1.8 Change Password

```
python pentest-cli.py -cP
```

Will be prompted to enter and confirm new password.

4.3.1.9 Output to File

If the user needs to redirect the output of any of the above commands to file:

```
python pentest-cli.py -r 10 -f results.out # Outputs the results of scan
    10 to results.out
```

5. Evaluation

Multiple scans were performed to test and evaluate the performance of the framework. In this chapter we present some of the performed scans and a summary of the results.

5.1 MWN SSH Scan

A scan of type ssh-basic (see 3.6.3) targeting the entire Munich Scientific Network (MWN) of 1,640,192 IP addresses was performed. The scanner was configured with a parallelization parameter of 100 to speed up the scan. The scan was started on the 5th of December 2013 and was completed after approximately 46 hours, corresponding to approximately 6 hosts per minute for each scan process.

5,578 out of 1,640,192 hosts scanned had SSH server listening on port 22, figure 5.1 shows the distribution of SSH versions detected.

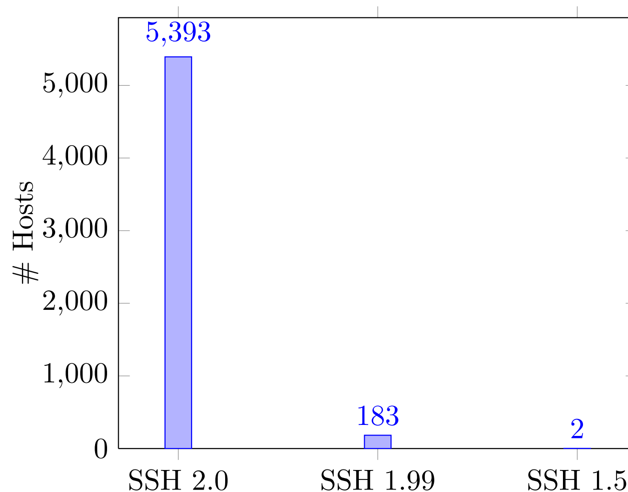


Figure 5.1: SSH Version

A total of 12,226 cryptographic keys were extracted from scanned hosts, 646 keys were shared between more than one host within the MWN with 1,607 hosts using a shared key.

Collected keys were also checked against keys encountered in the world-wide SSH scan performed by Gasser et al. [GaHC14] in the period from January to July 2013, results

show that 305 hosts within the MWN are using keys encountered outside the MWN, these hosts span 60 different MWN institutes, table 5.1 shows the top 5 institutes with duplicate keys. Figure 5.2 shows the distribution of the keys outside the MWN that were also encountered within the MWN. Table 5.2 shows the top 5 Autonomous Systems where such keys were found.

Institute	# Duplicate keys
LMU München,Maier-Leibnitz-Laboratorium der Universität und der Technischen Universität München	33
LMU München,Meteorologisches Institut	26
LMU München,Institut für Astronomie und Astrophysik mit Universitäts-Sternwarte	26
TUM,WWW & Online Services	23
TUM,Lehrstuhl für Genomorientierte Bioinformatik	16

Table 5.1: Duplicate keys top MWN institutes

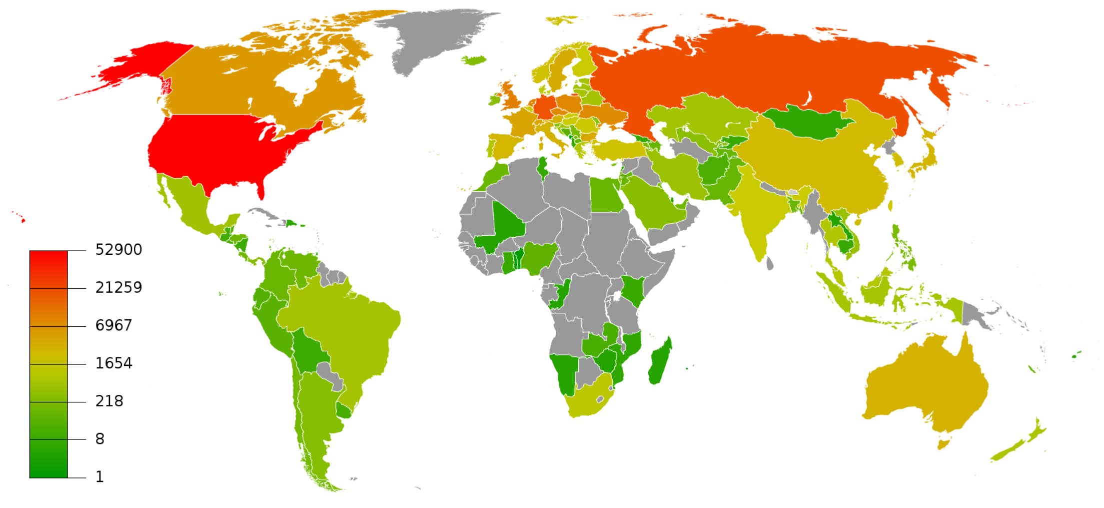


Figure 5.2: Duplicate keys map

ASN	# Hosts
6830	804
174	648
8220	546
5580	498
13768	470

Table 5.2: Duplicate keys top ASNs

Heninger et al. [HDWH12] describes a weakness in RSA keys caused by insufficient entropy during key generation, this causes some keys to share common factors which allows an attacker to easily retrieve the private key from the public key. Keys collected during the MWN scanned were checked and 1 key was found to be factorable using this method.

5.2 Joomla! Scan

To test that a Joomla! scan would correctly detect the running Joomla! version, two URLs were scanned in November 2013. The first URL *www.lrz.de* does not run Joomla! at all, the second URL *joomla.de* is expected to run a recent Joomla! version. Returned results are as follows.

URL	Scan Result
www.lrz.de	Failed to get Joomla version!
joomla.de	3.1.5

Joomla! version 3.1 was indeed the latest version at the time of scan, scan results were as expected.

6. Future Work

The framework collect scan results as a single block of text for each target scanned, this makes it difficult to extract meaningful information on a large scale. For example, how many hosts have port 22 open or which hosts run an old HTTP server version. More work needs to be done to parse scan results which will enable the implementation of features that can not be implemented at the current state of the system such as security alerts and risk analysis.

Additionally, the framework can be extended with more scan types to increase its usability. Examples of possible scans are: default or weak passwords scan, common vulnerabilities scan, testing possible exploits. The Joomla! scan can be improved by differentiating between hosts not running Joomla! at all and hosts that run Joomla! but we fail to detect its version number.

References

[beyo] beyondtrust. Retina CS Threat Management Console. http://www.beyondtrust.com/Products/RetinaCSThreatManagementConsole/.

[Foun] Django Software Foundation. The Web framework for perfectionists with deadlines | Django. https://www.djangoproject.com/.

[GaHC14] Oliver Gasser, Ralph Holz und Georg Carle. A deeper understanding of SSH: results from Internet-wide scans. Proc. 14th Network Operations and Management Symposium (NOMS), May 2014.

[GFI] GFI. GFI LanGuard. http://www.gfi.com/products-and-solutions/network-security-solutions/gfi-languard.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow und J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[mpyt] mod python. Apache / Python Intergration. http://modpython.org/.

[Nmap] Nmap. Nmap - Options Summary. http://nmap.org/book/man-briefoptions.html.

[Rapi] Rapid7. Metasploit Framework. https://github.com/rapid7/metasploit-framework.

[Secu] Tenable Network Security. Nessus Vulnerability Scanner. http://www.tenable.com/products/nessus.