NetBSD Wifi Browser
Software Requirements Specification

Kevin McGrane, Stephen Loudiana, Dylan Roy

November 22, 2021

Version: 1.0

Ver.	Date	Who	Change

1 Introduction

1.1 Purpose

The purpose of this document is to introduce the core features of our API. Moreover, we will demonstrate the efficiency of having a streamlined process for connecting to Wi-Fi through NetBSD, as well as, functional and non-functional features of our API, security and performance aspects, as well as product description and interfaces. Each feature of our project is explicitly described in detail. While our vision and scope document appends the schedule and requirements for our project, this document describes what we want our API to achieve, and any limitations that might exist.

1.2 Document Conventions

Any time the text like `this` is used, this refers to a function call or a NetBSD program.

1.3 Intended Audience and Reading Suggestions

The intended audience includes developers, customers/stakeholders, quality control, and documentation.

1.4 Project Scope

Vision and scope adheres to the requirements of our API. We plan to implement each core feature over the course of five milestones. Deliverables for our project include the API, command line interface and terminal user interface. Each milestone envelops a specific goal. Milestone one we expect to complete within' the first five weeks of development, in which a skeleton of our project will be implemented. Milestone two and three will consist of the correct implementation of our product, including the CLI and TUI features. Milestone four and five will consist of security testing, in addition to meeting with stakeholders and customers as needed. We expect each milestone to be equally distributed in the amount of time it takes to complete. Individual responsibilities are equally distributed among team members. Project scope is only defined by the time it takes for each milestone to be completed. Resources are minimal, API development will be handled directly on NetBSD operating system.

2 Overall Description

The NetBSD Wifi Browser is to fill a void that exists within the NetBSD operating system: the need for a streamlined method to connect to wifi. At the current moment, wifi connection requires a lot of manual effort and configuration to achieve a connection. The NetBSD Wifi Browser will simplify this process by automating some of the manual configuration bits, significantly reducing the effort and time required of the user.

2.1 Product Perspective

This product will be particularly beneficial when a user needs to connect to a new network, a common occurrence with fresh installs of NetBSD and when the computer with NetBSD has moved to a new location.

2.2 Product Features

The NetBSD Wifi Browser will be comprised of two components: an API and a user interface. The API will handle the interactions with `wpa_supplicant` and be the source of automation of network configuration, with the only required input from the user being the network (chosen from a list typically) and the passkey or username and password for that network. The UI then will utilize the API to allow a user to take advantage of the features the API will contain.

2.3 User Classes and Characteristics

General User The general user will have a desire to easily connect to wifi networks and remove known networks.

Developer The developer may use the API in their own development projects or in an alternative user interface.

2.4 Operating Environment

The expected operating environment is the NetBSD operating system. We are constructing an application program interface to interact with this specific type of operating system. This is not an API that can run on mobile devices, and is a standard desktop API used for making the process of connecting to existing Wi-Fi connections much simpler.

2.5 Design and Implementation Constraints

The intended environment for the NetBSD Wifi Browser is on a NetBSD desktop, initially only within a terminal as the only provided user interfaces will be a CLI or TUI.

2.6 Assumptions and Dependencies

The NetBSD Wifi Browser API will heavily depend on `wpa_supplicant` and its provided interfaces. The API will be designed and implemented under the assumption that `wpa_supplicant` and its interfaces will not change its policies.

3 Features

1. The API shall interface with `wpa_supplicant` and `ioctl`

2. The API shall hash passkeys and passwords provided to it

3. The API shall provide a manual process for connecting to a wifi network
4. The API shall provide a semi-automatic process for connection to a wifi network
5. The API shall provide a way to remove known/configured networks
6. The CLI/TUI shall provide an interface to the API for the user

3.1 The API shall interface with `wpa_supplicant` and `ioctl`

3.1.1 Description

In NetBSD, `wpa_supplicant` and `ioctl` are utilized to determine available wifi networks and the specific details of a particular wifi network. The information from these are used to construct a `wpa_supplicant` configuration file, with which the `wpa_supplicant` daemon is then run, connecting the wifi interface to the wifi network. The API would streamline this process, grabbing all of the necessary information from `wpa_supplicant` and ioctl and putting it into a configuration file upon a network selection, requiring minimal intervention from the user.

3.1.2 Priority

The feature is of high priority as it is the core of the purpose of the existence of this product.

3.1.3 Stimulus and Response

This feature will be utilized every time a user needs to connect to a new network. Upon it's use, the API will present the available (visible) networks and a network will then need to be selected from that list. The API will then handle the configuration file and restarting `wpa_supplicant` for the connection to take affect.

3.1.4 Functional Requirements

- The API shall require a call to it to gather available network information
- The API shall provide a list of available networks to connect to
- The API shall handle `wpa_supplicant` configuration
- The API shall restart the `wpa_supplicant` service (to read in new/appended
- The API shall not require root privileges from the user config file)

3.2 The API shall hash passkeys and passwords

3.2.1 Description

When using `wpa_supplicant`, the default method of providing a passkey (to connect to a network) or a password (for connecting to a network with a specific user account) is be storing them in plain text in the configuration file. However, `wpa_supplicant` supports hashed passkeys and passwords in the configuration file, providing a small additional layer to security. For passkeys, the `wpa_supplicant` package provides a hasher program. For passwords, piping a password through `iconv` and `openssl` provides a hashed password.

3.2.2 Priority

This feature is important as it further protects a user's personal data and prevents potentially malicious actors from having an easier entrance into a network.

3.2.3 Stimulus and Response

This feature will be triggered any time the user inputs a passkey or a password. Upon receiving a passkey, the API will run the passkey through the provided `wpa_supplicant` hashing program before storing it in the configuration file. For passwords, `wpa_supplicant` will pipe the provided password through `iconv` and `openssl` before storing the password in the configuration file.

3.2.4 Functional Requirements

- The API shall hash passkeys and passwords before storage in the configuration file

3.3 The API shall provide a manual process for connecting to a network

3.3.1 Description

In using this product, there may be a time that a user will want to manually configure a network connection. Instead of writing a separate `wpa_supplicant` configuration file, the user will be able to provide the network information to the API and the API will handle the actual interfacing with `wpa_supplicant` and append it to the API's configuration file.

3.3.2 Priority

This feature is not of the highest priority as it does not actively work towards streamlining the interaction with `wpa_supplicant` and ioctl in a significant way, but should be easily implementable with the existing infrastructure since the user will be providing the same information that might be provided by `wpa_supplicant` and `ioctl`, and thus can be processed similarly.

3.3.3 Stimulus and Response

This feature will occur when the user invokes the call specifying there will be manual input from the user. The user will provide at minimum the necessary information, as well as any extra information for connecting to a network. If there is cause for more information to be provided in the configuration, the API will grab the remaining necessary information through a query with `wpa_supplicant` or `ioctl`.

3.3.4 Functional Requirements

- The API will provide a library call for manual configuration
- The API will fill in any necessary information not provided by the user (the minimum provision being an SSID)
- The API will add the configuration information to its internal `wpa_supplicant` configuration file

3.4 The API will provide a semi-automatic process for connecting to a wifi network

3.4.1 Description

In NetBSD, much of the time spent connecting to a wifi network is spent identifying the necessary information for the `wpa_supplicant` configuration file. The API can circumvent this by gathering the information automatically for the user and automatically updating the internal `wpa_supplicant` configuration file.

3.4.2 Priority

The priority of this high as this is the main feature that provides the streamlined experience of connecting to a wifi network in NetBSD.

3.4.3 Stimulus and Response

This feature will occur when the call for semi-automatic configuration is made on the API. Upon invocation, the API will gather the SSIDs of the available (visible) networks and return them. The API will then require a network to be selected from the list and the passkey or password associated with the connection before gathering all the necessary information of the network from `wpa_supplicant` and `ioctl`, appending it into the API's internal `wpa_supplicant` configuration file.

3.4.4 Functional Requirements

- The API shall return a list of available (visible) networks when requested
- The API shall handle the configuration of a `wpa_supplicant` configuration file

3.5 The API shall provide a way to remove known/configured networks

3.5.1 Description

There are many situations in which it makes sense to remove a network from the list of known/configured networks: don't need it anymore, contains potentially sensitive information, or looking for minimal ways to save space. Whatever the reason is, if a configured network can be added to the configuration file, then a configured network can be removed.

3.5.2 Priority

This is of moderate priority, but is as simple as removing the lines that the network occupies in the configuration profile.

3.5.3 Stimulus and Response

Deletion of a network will be triggered when the user makes the call on the API. At that point, the API will locate the network configurations position in the configuration file and remove the lines that it occupies. When it is finished, it will pass the modified configuration file to `wpa_supplicant`.

3.5.4 Functional Requirements

- The API will have the ability to remove a network configuration from the configuration file.

3.6 The CLI/TUI shall provide an interface to the API for the user

3.6.1 Description

The API by itself is only useful if using as a library in source code of the same language. Thus, a CLI/TUI will be provided as a way for a user to actually interface with the API and its services. Through the interface, the user will be able to take advantage of the API's features.

3.6.2 Priority

The priority of at least one of these interfaces is high as this will serve as the face of the product. Not many people will want to build their own interface on our API library, so providing one that supplies all the necessary functionality is a must.

3.6.3 Stimulus and Response

The CLI/TUI will be used any time a user needs to manage their network connections, new or current. The interface will provided the tools necessary for interacting with those networks and manage (most of) the heavy lifting for the user.

3.6.4 Functional Requirements

- The CLI/TUI will provide the user with the ability to directly interface with the API
- The CLI/TUI will provide formatted presentation of the user's options
- The CLI/TUI will present formatted output from the API

4 External Interface Requirements

4.1 User Interfaces

- A computer running the NetBSD operating system

4.2 Hardware Interfaces

- A Wifi card or dongle attached to the host computer
- A Wifi network to connect to

4.3 Software Interfaces

- `wpa_supplicant` and its included programs
- Wifi driver, supporting different wifi protocols

4.4 Communication Interfaces

- Wifi protocols for connecting to the wifi network (mostly handled by the driver with details provided by `wpa_supplicant`)

5 Other Nonfunctional Requirements

Location of Configuration File The `wpa_supplicant` configuration file will be stored locally per user. This makes it so that root access is not required to modify the network configurations. With this, a central file will exist in conjunction containing ”generic” versions of configured networks on the machine so that other users can utilize those configured networks, potentially with their own information (like with a WPA-EAP network).

5.1 Performance Requirements

- In the API, automatic configuration should not take more than 1 second past the time it takes for the user to choose and network and enter the passkey or a username and password.
- After changing a configuration file, the configuration should not take more than 5-10 seconds to take effect in the `wpa_supplicant` process.

5.2 Security Requirements

Hashing of Passkeys and Passwords Passkeys will be hashed with the passkey hasher provided in the `wpa_supplicant` package, and passwords will be hashed via a pipe through `iconv` to change the encoding to utf16 and piped to `openssl` to encrypt in an md4 hash. The purpose of this is to prevent storing passkeys and password in plain text, and offer an extra (albeit extremely thin) layer of security.

5.3 Software Quality Attributes

- The API will accurately determine and place configuration information in the `wpa_supplicant` configuration file for a given network.
- The CLI/TUI will encapsulate all of the functionality that the API offers and way to easily interact with that functionality.

A Glossary

API Application Programming Interface. Used in programming, provides a library of functions that can simplify the efforts of a programmer or provide a simpler interface to one or more services/libraries.

CLI Command Line Interface. A style of user interface that centralizes around using the command line and commands provided by the interface.

TUI Terminal User Interface. A style of user interface that utilizes the terminal to

wpa_supplicant `wpa_supplicant` is a program and daemon used for connecting to wifi networks. It utilizes a configuration profile containing information about networks to connect to, informing `wpa_supplicant` about how to connect to different networks.

B Analysis Models

B.1 Use Cases

Use Case 1: Connect to Wi-Fi using our API
Actor: User

Use Case Overview: The user of the system wants to connect to Wi-Fi using our API.
Trigger: The system has no current Wi-Fi connection and the user wishes to implement one.
Precondition 1: There is a valid internet connection available.
Basic Flow: The user wishes to connect to Wi-Fi semi–automatically using our API. The user selects an interface to our API and then selects a connection from the list of available Wi-Fi connections. The user then connects to the Wi-Fi connection they selected. Our API processes the passkeys and passwords in the configuration file. The user successfully connects to Wi-Fi.
Alternative Flow 1: The user wishes to connect to Wi-Fi manually using our API. The user selects set up Wi-Fi connection. The user manually inputs the network information required to establish a connection. The user selects establish connection. Our API processes the required information and successfully connects to Wi-Fi.
Termination Outcome: The user has successfully connected to Wi-Fi using our API.

Use Case 2: Disconnect from Wi-Fi using our API
Actor: User
Use Case Overview: The user of the system wants to disconnect from Wi-Fi using our API.
Trigger: The system has a valid Wi-Fi connection and the user wishes to disconnect.
Precondition 1: A valid internet connection is established within' the system.
Basic Flow: The user selects disconnect from Wi-Fi on our interface to the API. The system successfully disconnects from Wi-Fi.
Termination Outcome: The user has successfully disconnected from Wi-Fi using our API.

C Issue List

I don't believe there are any outstanding requirement issues for our API.