CS 486 – Capstone Project

Coding Standards

(Revision 1.0)

Submitted to

Dr. Doerry

By

Team Fugu:

Erik Wilson

Ben Atkin

Nauman Qureshi

Thad Boyd

On

March 3, 2004

Table of Contents

1. Executive Summary

On the 5th of January 2004 the United States Geological Survey (USGS) approached Northern Arizona University's (NAU) Capstone Project with an idea of developing OS tools for OpenBSD. Team Fugu (*http://www.cet.nau.edu/~fugu/*) was formed in order to develop this project. USGS is a world leader in the natural sciences through their scientific excellence and responsiveness to society's needs. The USGS Astrogeology Program uses OpenBSD due to its renowned security but the costs in time for installation and maintenance of the operating system are a big drawback.

On the 16th of January 2004 Team Fugu was chosen to develop an automated installer and an automated patcher for OpenBSD which when developed would alleviate the USGS of their problems. During the course of developing this product each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make our programs easier to read, understand, and maintain. The purpose of this document is to provide those coding standards that will be beneficial to legibility, but in addition will improve our ability to test, debug, and integrate during the development process.

2. File Naming Conventions

File names will be lowercase, with words separated by underscores (e.g. file_name.sh). The extension of a file will be indicative of the type of script which is being run (e.g.: .sh, .pl, .awk), but if a script resides within the executable path of a user it is okay to omit the extension (e.g.: "tepatche" instead of "tepatche.pl", because it resides in */usr/local/sbin*). The names of files should be mnemonic, in other words they will be descriptive without being excessively wordy. For example, the following are *bad* ways to name a file such as the one which would parse configuration files for the automated installer:

`parser.awk` – *This is too short, what is it parsing?*
`automated_configuration_parsing_program.awk` – *Obviously too descriptive.*

However, some *good* examples are as follows:

`auto_conf_parser.awk` – *Understandable what auto means and conf is configuration.*
`install_parser.awk` – *The parser is used in the context of an install, this is okay.*

As a general rule of thumb if the name is shorter than 8 characters or longer than 24 it might be a good idea to try and find a better name, however sometimes this may be unavoidable.

3. Prologue Standards

We will be using prologues, for the purpose of better understanding each file and each module (function or procedure), and so the source will be easier to browse. Revision histories will be omitted from the prologues, as this will be maintained by the version control software, please see *Section 7* for more details. The following two sections describe prologues for files and for modules in greater detail:

3.1. File Prologue Format

The file prologue will immediately follow the shebang line (“#!/usr/bin/sh”, “#!/usr/bin/perl”, or “#!/usr/bin/awk”), where applicable, and will use the following format:

```
#!/usr/bin/sh

# file:           file_name.sh
# author:         Joe Slinger
# last modified:  2004.01.14
# description:
# A one or two sentence description will go here. Longer
# introductions will follow the license if needed.
```

As stated above the revised BSD license will be included with every file, an example of this license can be found at *http://www.cet.nau.edu/~fugu/*.

3.2. Module Prologue Format

Modules are smaller parts of files such as functions or procedures, and therefore will have descriptive features as to the inputs and return values, in example:

```
# name:           log_error
# author:         Joe Slinger
# input:          text of log, date
# return value:   0 for failure, 1 for success
# description:
# A one or two sentence description will go here.
```

Name, author, and revision history will also go in the CVS changelogs for each file. These will be backed up and preserved.

4. Symbol Naming Standards

Variables will also be lowercase with underscores separating words. Global variables will be descriptive but should also be mnemonic and not excessively verbose. For example, we want to create a global variable which will indicate if the log file has been locked or not, some *bad* variable declarations are as follows:

`int log_file_lock_indicator_for_log_error = 0;` – *Obviously too descriptive.*
`int lflifle = 0;` – *Meaningless to other developers, always avoid using acronyms.*

However some *good* examples are:

`int log_lock = 0;` – *Other developers understand it is related to logging and locking.*
`int locked_log_file = 0;` – *Even more descriptive yet not excessively.*

Comments better describing global variables will be included, see the next section for more detail. Local variables will have short but sensible names, with optional comments for clarity. Variables which are very short lived, such as counters for loops, may be very terse (e.g., the variable *i*), as their purpose is obvious.

5. Commenting Standards

Functions and lengthy control structures will be preceded by a comment describing what they do, and their purpose. Comments will be added for clarity, but excessive commenting will be avoided. Any comment should also avoid simply repeating the code, but instead be more suggestive as to the purpose of that code. The following is a *bad* example of how to comment a global variable:

```
# Integer variable to lock the log
int log_lock = 0;
```

The previous example is too terse and does not provide any additional insight to that variable.

A much better example is:

```
# log_lock is used by 'flock' in write_log to avoid race conditions
#   Initializing to 0 assumes the log file is not locked
int log_lock = 0;
```

The previous example describes exactly what context the variable is used in and why it needs to be initialized to that value.

6. Whitespace Standards

Whitespace, while not necessary in the languages we're using, makes it easier for humans to read code. Spaces before the open-parentheses and after commas make a program easier to read, but more importantly rules for indenting and using blank lines need to be specifically adhered to. For this reason we are making the following requirements for indentation and blank lines in our code:

6.1. Indentation

Perl and manual pages have indentation guidelines, that we chose to adhere to. For the shell scripts used in the installer, and the Makefile for the patcher, we chose to use the indentation that is used for the code we are deriving from, for consistency's sake. Here are the indentation standards for each type of source file:

File Type	***Indentation Characters***	***Reasoning***
Perl Scripts (patcher)	4	Perl standard, used in current Tepatche source.
Shell scripts (installer)	8	Used in interactive install. Common for shell scripts and Makefiles.
Manual pages	8	Standard for UNIX man pages.

These guidelines will be strictly adhered to. Anyone using an editor with configurable indentation may need to customize their indentation to get it working. This is easy to do on most advanced text editors, such as *vi* and *emacs*.

6.2. Blank Lines

Blank lines should be used between functions and major control sections to improve readability. Excessive use of blank lines should be avoided simply because this tends to reduce the legibility, therefore the number of blank lines used should be limited to three.

7. Version Control and Organization

We have set up a CVS repository, and sent out basic CVS posting instructions. We will set up WebCVS for source browsing, and a CVS server (which is not necessary to use CVS, but makes it more convenient).

All source files will be committed to our CVS version-controlled repository. This will allow us to track the revisions of our code, and will allow us to go back and look at old code when needed. It is required that with each commit to CVS, the person posting it write a short description of the changes that were made.