2014-12-12

Trusted Computing Exemplar: Software Development Standards

Clark, Paul C.

Monterey, California. Naval Postgraduate School

NPS-CAG-14-007

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Trusted Computing Exemplar: Software Development Standards

by

Paul C. Clark, Cynthia E. Irvine,
Thuy D. Nguyen, and David Shifflett

12 December 2014

Approved for public release; distribution is unlimited

Prepared for: United States Navy, OPNAV N2/N6

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Ronald A. Route
President

Douglas A. Hensler
Provost

The report entitled "Trusted Computing Exemplar: Software Development Standards" was prepared for United States Navy, OPNAV N2/N6 and funded in part by United States Navy, OPNAV N2/N6.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Paul C. Clark
Research Associate

Cynthia E. Irvine
Distinguished Professor

Thuy D. Nguyen
Research Associate

David Shifflett
Research Associate

Reviewed by:

Cynthia E. Irvine, Chair
Cyber Academic Group

Released by:

Jeffrey D. Paduan
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From-To)*
12-12-2014	Technical Report	Nov 2013 to Nov 2014

4. TITLE AND SUBTITLE
Trusted Computing Exemplar: Software Development Standards

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Paul C. Clark, Cynthia E. Irvine, Thuy D. Nguyen, and David Shifflett

5d. PROJECT NUMBER
W4C05

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CAG-14-007

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rhonda Onianwa
OPNAV, N2N6 F13
rhonda.onianwa@navy.mil

LT David Rivera
OPNAV, N2/N6F1
david.j.rivera4@navy.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for pubic release; distribution is unlimited

13. SUPPLEMENTARY NOTES
The view expressed in this report are those of the authors and do not reflect the official policy or position of the Department of Defense of the U.S. Government.

14. ABSTRACT

This document describes the Life Cycle Management Plan for the development of a high assurance secure product. A high assurance product is one for which its users have a high level of confidence that its security policies will be enforced continuously and correctly. Such products are constructed so that they can be analyzed for these characteristics. Lifecycle activities ensure that the product reflects the intent to ensure that the product is trustworthy and that vigorous efforts have been made to ensure the absence of unspecified functionality, whether accidental or intentional.

This document provides policy and process for developing and approving software-related Configuration items (CIs), giving more detail than was covered in the Life Cycle Management Plan (LCMP). This document does not replace the LCMP, it expands on the principles and processes the LCMP defined, and should not conflict with the LCMP in any way. Other documents will describe the standards for hardware development.

15. SUBJECT TERMS
Machinery control systems, MCS, life cycle security, high assurance, system security, trustworthy systems

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Cynthia E. Irvine
Unclassified	Unclassified	Unclassified	UU	19	**19b. TELEPHONE NUMBER** ***(include area code)*** (831) 656 2461

Standard Form 298 (Rev. 8-98)

Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

CYBER ACADEMIC GROUP
NAVAL POSTGRADUATE SCHOOL

NPS-CAG-14-007

Trusted Computing Exemplar: Software Development Standards

Paul C. Clark
Cynthia E. Irvine
Thuy D. Nguyen
David Shifflett

December 2014

ATTRIBUTION REQUEST

December 2014

The Cyber Academic Group (CAG) and the Center for Information Systems Security Studies and Research (CISR) at the Naval Postgraduate School (NPS) wish to facilitate and encourage the development of highly robust security systems.

To further this goal, the NPS CAG and NPS CISR ask that any derivative products, code, writings, and/or other derivative materials, include an attribution for NPS CAG and NPS CISR. This is to ensure that the public has a full opportunity to direct questions about the nature and functioning of the source materials to the original creators.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the following organizations for providing support toward the development of this work: OPNAV N2/N6 F1.

The material presented here builds upon work supported in previous years by the Office of Naval Research.

A portion of the material presented here is based upon work supported by the National Science Foundation under Grant No. CNS-0430566 and CNS-0430598. This document does not necessarily reflect the views of the National Science Foundation.

Table of Contents

1 Introduction 1

2 Coding Standards 1

2.1 Language 1
2.2 Commenting and Readability 1
2.3 Constants and Macros 1
2.4 Scope 2
2.5 Curly Braces 2
2.6 Switch Statements 3
2.7 Types and Storage Classes 4
2.8 File Style 5
2.9 Functions 5
2.10 C Constructs 6
2.11 Pointers 6
2.12 Naming Conventions 6
2.13 Code Correspondence 7
2.14 Assembly Language 7
2.15 Peer Review 7

3 Testing Standards 7

References 8

[THIS PAGE IS INTENTIONALLY BLANK]

1 Introduction

This document has been written in support of a research project to publicly demonstrate and document how a high assurance product can be developed and distributed. A high assurance product is one for which its users have a high level of confidence that its security policies will be enforced continuously and correctly. Such products are constructed so that they can be analyzed for these characteristics. Lifecycle activities ensure that the product reflects the intent to ensure that the product is trustworthy and that vigorous efforts have been made to ensure the absence of unspecified functionality, whether accidental or intentional.

This document provides policy and process for developing and approving software-related Configuration items (CIs), giving more detail than was covered in the Life Cycle Management Plan (LCMP) [1]. This document does not replace the LCMP, it simply expands on the principles and processes the LCMP defined, and should not conflict with the LCMP in any way. Other documents will describe the standards for hardware development.

2 Coding Standards

This section describes the programming standards.

2.1 Language

An ANSI-C compliant language shall be used when programming trusted code, except in those rare circumstances when assembly code must be used (e.g., during initialization). The preferable language for untrusted code is also the C language, but other alternatives will be considered (e.g., shell scripts), depending on the situation. However, variations from the C language must be approved by the Configuration Item (CI) Leader.

2.2 Commenting and Readability

Comments are encouraged and shall use the "//" syntax because they lead to fewer mistakes than the "/* */" syntax. The "/* */" syntax is allowed when a comment is made in a "#define" statement to avoid potential errors in the pre-processing stage of compilation.

Tab characters shall **not** be used for white space, due to the inconsistent presentation of the amount of space a tab receives across applications. White space shall be introduced with the space character. Indentation of code blocks within a source file shall be four spaces.

Lines in a source file shall not be so long that they will wrap around to the next line when printed on an 8-1/2" x 11" piece of paper.

2.3 Constants and Macros

Constants shall only be used when associated with a #define construction. In other words, symbolic constants are the only allowable use of constants.

Macros shall only be used to implement code if the code is small, uncomplicated, and there is a concern about the impact on performance if it was implemented as a function. Use of macros shall be approved by the CI Leader, and such code shall be critically reviewed.

2.4 Scope

Variables are not allowed to be accessible outside the source file they are declared in. A variable can have scope across a source file if it is considered a "database" managed by the associated module. See Section 2.7.3.

2.5 Curly Braces

When curly braces are used to bracket a function, the beginning curly brace shall be on a line by itself, and the ending curly brace shall be followed by a comment that identifies the name of the function being terminated, as shown below:

```
int foo(void)
{
    // body

} // foo()
```

Otherwise, beginning curly braces are put at the end of the first line of a code block. The ending curly brace for such blocks may be followed by an optional comment, as shown below:

```
while (temp < BOILING) {
    // body

} // while
```

All statements that follow a condition or loop statement shall be contained within curly braces, even if it is only one statement that could syntactically be done without braces. For example, the following shall **not** be used:

```
if (temp < BOILING) temp++; // This is not allowed
```

Instead, the following syntax shall be used:

```
if (temp < BOILING) {
    temp++;
}
```

If, then, else statements shall be written in the following style, with the else statement being on the same line as the previous ending curly brace and its own beginning curly brace, as shown below:

```
if (temp) < FREEZING) {
    // body
} else if (temp < BOILING) {
    // body
} else {
    // body
}
```

2.6 Switch Statements

The following style shall apply for switch statements:

```
switch (color) {
    case RED:
        // statements
        break;
    case GREEN:
    case BLUE:
        // statements
        break;
    default:
        // statements
        break;
} // switch
```

If a case shall purposefully "fall" through to the next case (i.e., no "break" statement is used), then it must be commented in the code, unless two or more cases are adjacent, as shown with the GREEN and BLUE cases above.

2.7 Types and Storage Classes

2.7.1 Const Type Specifier

"The const type specifier prevents objects from having their value changed" [2]. If an input to a function is not expected to change, then the "const" type specifier shall be used in the corresponding function declaration, as shown below.

```
int isfrozen( const int freezingpoint, const int temp )
{
    int result = NO;

    if (temp <= freezingpoint) {
        result = YES;
    }

    return(result);
} // isfrozen()
```

2.7.2 Void Type Specifier

When a function does not have any arguments, the "void" type specifier shall be used to explicitly show it.

2.7.3 Static and Extern Storage Classes

Functions and variables declared outside of functions need to be explicitly declared as either the "static" or "extern" storage class. The "extern" storage class shall only be used when declaring exported functions.

The "static" storage class shall be used on all internal functions that are not to be exported by the linker, viz., all non-exported functions. "static" shall also be used on all variables that have file-level scope.

2.7.4 Type Conversion

The C language does not have strong type checking, which can introduce problems not easily identified during compilation time. Therefore, the policy in this section attempts to minimize such problems.

Type conversion shall not be used without an adjacent comment describing why it is used, and why it is safe. Extra special care shall be taken in the source code (e.g., range checks) when a type with a smaller memory size is receiving data from a bigger memory size, e.g., a 32-bit integer being assigned to a 16-bit integer. Peer Review shall inspect such code with extra care.

2.8 File Style

In general, the following order shall be used in files:

1. file header
2. ifndef statement (for header files)
3. include statements
4. define statements
5. variable definitions
6. function prototypes
7. function implementations (for source files)
8. endif statement (for header files)

Every source file will have a header with the same style, as shown below.

Describe here what the organization's standard file header will contain, such as licensing information, contact information, developer information, modification descriptions, etc.

Version numbers, such as a CI version, shall not be used in the modification description in a file header. In the event of a branching of a source tree, however, the description of the modification may have informal advisory information about what was changed with respect to a version, as an aid for potential merging of the branches.

Header files (i.e., files with a ".h" suffix), shall have the following after the header described above:

```
#ifndef _FILENAME_H_
#define _FILENAME_H_
```

Note that the syntax for the name definition is a leading and trailing underscore, with another underscore taking the place of the "." in the file name. Everything is in uppercase. Therefore, if the name of the header file is "inputs.h", then the line would look as follows:

```
#ifndef _INPUTS_H_
#define _INPUTS_H_
```

In addition, the last line of every header file shall have the matching endif, as shown below:

```
#endif //  FILENAME H
```

2.9 Functions

Function prototypes for functions only used internally to a source file shall be specified in full (e.g., no ellipses for the list of arguments) near the beginning of the file, and shall be

declared with the “static” storage class. (See Sections 2.7.3 and 0). All prototypes shall be identical to the function implementation. Input arguments specified with the “const” type specifier shall be listed first in a prototype. A variable number of function arguments must be approved by the CI Leader. Function pointers passed as arguments must be approved by the CI Leader.

All input parameters must be validated before they are actually used.

As a general rule, functions shall return a status value, i.e., a success or failure code, which is returned as a function result, not as an output argument. A function without a return value must be approved by the CI Leader. The caller of a function shall check the returned status before continuing, and handle any errors appropriately.

Functions shall have one entry point and one exit point. For example, there shall not be multiple “return” statements in a function.

Within the processing of a function, output variables shall only be used to track the value of a potential output, and shall not be used for other purposes.

As a general rule, functions should be less than 100 lines in length (excluding comments).

2.10 C Language Constructs

Switch statements shall have a default action, even if it seems like such a case will never be seen.

Goto statements shall not be used unless explicitly approved by the CI Leader. Even then, it is expected that they will rarely be used, if ever.

Conditional compilation shall only be used to separate debugging statements and CPU architecture differences. Of the two types of conditional statements (#ifdef and #if), #ifdef shall be used for consistency, unless a feature of the #if style is the only way to accomplish a desired compilation.

2.11 Pointers

The explicit use of pointers is seen as both an advantage and disadvantage of the C language. Inappropriate use can lead to undesired behavior. Peer Review shall carefully inspect all use of pointers.

If the value of a pointer cannot be assigned when the pointer is declared, then it shall be initialized with NULL. Pointers shall be compared to NULL before they are first used within a given scope.

2.12 Naming Conventions

Function and variable names should not be overly long.

Symbolic constants and macros shall be defined in all uppercase characters.

All function names shall only contain lower-case characters, underscores and numbers.

All non-global variable names shall start with a lower-case character.

All variables that are global to a source file shall start with an upper-case character, followed by all lower-case characters.

Compound names shall be separated by an underscore.

2.13 Code Correspondence

This subsection needs to describe the requirements on software developers that will support the organization's approach for code correspondence.

2.14 Assembly Language

As stated in Section 2.1, assembly language shall be used on a limited case-by-case basis. When it is used, the assembly code shall be placed in a C source file as inline code. Exceptions shall be approved by the CI Leader.

2.15 Peer Review

Prior to CCB submission, all code shall be peer reviewed by a person of similar skill level as the author of the item under review. The Peer Review shall not be performed until the code has been completed and the unit tests have been successfully performed, which shall be noted with their dates of completion in the review evidence. The reviewer is responsible for ensuring that the item conforms to all coding standards..

The peer review of a source file shall not be done by the author of the file. Because a CI may consist of many files that were authored by many people, the following shall be clearly noted in the review evidence: the author(s) of each file, and the peer reviewer(s) of each file.

3 Testing Standards

Testing strategies and test cases shall cover the following:

- Positive behavior

  Testing needs to show that all required functionality works as specified.

- Negative behavior

  Testing needs to show that obvious undesired behavior is not present. For example, it is not enough to test whether an authorized subject can access an object; the testing must also show that an unauthorized subject cannot access an object.

Where possible, all error conditions shall be tested to ensure that the condition is detected, and that the specified reaction is seen (e.g., the proper error code is returned).

The results of all test cases shall be documented.

It is acceptable for the author of a source code representation of a module to write and administer the unit tests. This allows the module to be tested before other dependent modules are written. Because the size of the development group is assumed to be small, it shall also be acceptable for the higher-level tests to be written and administered by someone who wrote some of the modules comprising the subsystem and product. In such a case, a peer review of the higher-level test code shall judge whether the tests are complete.

References

[1] P. C. Clark, C. E. Irvine, T. Levin, and T. D. Nguyen, "Trusted Computing Exemplar: Lifecycle management plan," Naval Postgraduate School, Monterey, CA, Tech. Rep. NPS-CAG-14-002, Dec. 2014.

[2] S. P. Harbison, G. L. Steele Jr., *C: A Reference Manual*, 2nd ed., Englewood Cliffs, NJ, Prentice-Hall, 1987.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Ft. Belvoir, Virginia

2. Dudley Knox Library, Code 013 2
   Naval Postgraduate School
   Monterey, California 93943

3. Research Sponsored Programs Office, Code 41 1
   Naval Postgraduate School
   Monterey, California 93943

4. Paul C. Clark 1
   Naval Postgraduate School
   Monterey, California 93943

5. Dr. Cynthia E. Irvine 1
   Naval Postgraduate School
   Monterey, California 93943

6. Thuy D. Nguyen 1
   Naval Postgraduate School
   Monterey, California 93943