Document: WG 14/N1619

Text for comment WFW-1 of N1618

Introduction

Background

An essential element of secure coding in the C programming language is a set of well-documented and enforceable coding rules. The rules specified in this Technical Specification apply to analyzers, including static analysis tools and C language compiler vendors that wish to diagnose insecure code beyond the requirements of the language standard. All rules are meant to be enforceable by static analysis.

The application of static analysis to security has evolved in an ad hoc manner. This is useful from the point of view of exploring a new market to see what works. However, it has resulted in a fragmented market, with different vendors addressing different security issues and no way for a purchaser to specify the minimum requirements of a static analysis tool. Now that the shape of security needs is becoming clearer, there is a need for a specification that says, "For an analysis tool to conform to this specification, it must be able to do at least *this much*," where *this much* is well specified. By imposing a floor on analysis capabilities rather than circumscribing them completely, a specification can allow for continued improvements while still giving customers a way to know what they are buying.

The largest underserved market in security is ordinary, non-security-critical code. The security-critical nature of code depends on its purpose rather than its environment. The UNIX finger daemon (`fingerd`) is an example of ordinary code, even though it may be deployed in a hostile environment. A user runs the client program, `finger`, which sends a user name to `fingerd` over the network, which then sends a reply indicating whether the user is logged in and a few other pieces of information. The function of `fingerd` has nothing to do with security. However, in 1988, Robert Morris compromised `fingerd` by triggering a buffer overflow, allowing him to execute arbitrary code on the target machine. The Morris worm could have been prevented from using `fingerd` as an attack vector by preventing buffer overflows, regardless of whether `fingerd` contained other types of bugs.

By contrast, the function of `/bin/login` is purely related to security. A bug of any kind in `/bin/login` has the potential to allow access where it was not intended. This is security-critical code.

Similarly, in safety-critical code, such as software that runs an X-ray machine, any bug at all could have serious consequences. In practice, then, security-critical and safety-critical code have the same requirements.

There are already standards that address safety-critical code, and therefore security-critical code. The problem is that because they must focus on preventing essentially all bugs, they are required to be so strict that most people outside the safety-critical community do not want to use them. This leaves ordinary code like `fingerd` unprotected.

This Technical Specification has two major subdivisions:

preliminary elements (clauses 1-4) and

secure coding rules (clause 5).

Annexes provide additional information. A bibliography lists documents that were referred to during the preparation of the standard.

The rules documented in this Technical Specification rely only on non-annotated source files and not upon assumptions of programmer intent. However, a conforming implementation may take advantage of annotations to inform the analyzer. The rules, as specified, are reasonably simple, although complications can exist in identifying exceptions. Additionally, there are significant differences in rules that are intended primarily for evaluating new code versus legacy code. Because security is the primary concern, these rules are intended first and foremost for evaluating new code and secondarily for evaluating legacy code. Consequently, the application of these rules to legacy code may result in false positives. However, legacy code is generally less volatile, and many static analysis tools provide methods that eliminate the need to research each diagnostic on every invocation of the analyzer. The implementation of such a mechanism is encouraged but not required.

Completeness and soundness

To the greatest extent possible, an analyzer should be both complete and sound with respect to enforceable rules. An analyzer is considered sound (with respect to a specific rule) if it does not give a false-negative result, meaning it is able to find all violations of a rule within the entire program. An analyzer is considered complete if it does not issue false-positive results, or false alarms. The possibilities for a given rule are outlined in Table 1.

Table 1-Completeness and soundness

	False positives		
False negatives		Y	N
	N	Sound with false positives	Complete and sound
	Y	Unsound with false positives	Unsound

The analyzer shall report a diagnostic for at least one program that contains a violation of each rule.

There are many tradeoffs in minimizing false positives and false negatives. It is obviously better to minimize both, and there are many techniques and algorithms that do both to some degree. However, once an analysis technology reaches the efficient frontier of what is possible without fundamental breakthroughs, it must select a point on the curve trading off these two factors (and others, such as scalability and automation). For automated tools on the efficient frontier that require minimal human input and that scale to large code bases, there is often tension between false negatives and false positives.

It is easy to build analyzers that are in the extremes. An analyzer can report all of the lines in the program and have no false negatives at the expense of large numbers of false positives. Conversely, an analyzer can report nothing and have no false positives at the expense of not reporting real defects that could be detected automatically. Analyzers with a high false-positive rate waste the time of developers, who can lose interest in the results and therefore miss the true bugs that are lost in the noise. Analyzers with a high number of false negatives miss many defects that should be found. In practice, tools needs to strike a balance between the two.

The degree to which conforming analyzers minimize false-positive diagnostics is a quality of implementation issue. In other words, quantitative thresholds for false positives and false negatives are outside the scope of this Technical Specification.

Analyzers are trusted processes, meaning that developers rely on their output. Consequently, developers must ensure that this trust is not misplaced. To earn this trust, the analyzer supplier should, ideally, run appropriate validation tests. Although it is possible to use a validation suite to test an analyzer, no formal validation scheme exists at this time.

Security focus

The purpose of this Technical Specification is to specify analyzable secure coding rules that can be automatically enforced to detect security flaws in C-conforming applications. To be considered a security flaw, a software bug must be triggered by the actions of a malicious user or attacker. An attacker may trigger a bug by providing malicious data or by providing inputs that execute a particular control path that in turn executes the security flaw. Implementers are required to distinguish violations that involve tainted values from those that do not involve tainted values.

Taint analysis

Tainted values and tainted sources

Document: WG 14/N1619

Text for comment WFW-1 of N1618

Certain operations and functions have a domain that is a subset of the type domain of their operands or parameters. When the actual values are outside of the defined domain, the result might be either undefined or at least unexpected. If the value of an operand or argument may be outside the domain of an operation or function that consumes that value, and the value is derived from any external input to the program (such as a command-line argument, data returned from a system call, or data in shared memory) that value is tainted, and its origin is known as a tainted source. A tainted value is not necessarily known to be out of the domain; rather, it is not known to be in the domain. Note also that only values, and not the operands or arguments, can be tainted; in some cases the same operand or argument can hold tainted or untainted values along different paths.

Tainted sources include

- parameters to the `main` function,
- the returned values from `localeconv`, `fgetc`, `getc`, `getchar`, `fgetwc`, `getwc`, and `getwchar`, and
- the input values or strings produced by `getenv`, `fscanf`, `vfscanf`, `vscanf`, `fgets`, `fread`, `fwscanf`, `vfwscanf`, `vwscanf`, `wscanf`, and `fgetws`.

Restricted sinks

Operands and arguments whose domain is a subset of the domain described by their types are called restricted sinks. Any pointer arithmetic operation involving an integer operand is a restricted sink for that operand. Certain parameters of certain library functions are restricted sinks because these functions perform address arithmetic with these parameters, or control the allocation of a resource, or pass these parameters on to another taintededness sink. All string input parameters to library functions are restricted sinks because those strings are required to be null-terminated, with the exception of `strncpy` and `strncpy_s`, which explicitly allow the source argument not to be null-terminated. Loop bounds are not restricted sinks.

Propagation

The taintness of values is propagated through operations from operands to results unless the operation itself imposes constraints on the value of its result that subsume the constraints imposed by restricted sinks. In addition to operations that propagate the same sort of taint, there are also operations that propagate taint of one sort of an operand to taint of a different sort for their results, the most notable example of which is `strlen` propagating the taint of its argument with respect to string length to the taint of its return value with respect to range.

Approaches to analysis

By definition, any tainted value flowing into a restricted sink is a security issue, so all such cases must be diagnosed. Diagnosing these violations requires some form of data flow analysis. In its most basic form, such an analysis operates intraprocedurally to determine which local tainted sources flow into local restricted sinks. Intraprocedural analysis is limited and can be extended by interprocedural analysis. Interprocedural analysis can be accomplished by top-down or bottom-up approaches that follow global data flow more than they follow the call graph. For example, in a bottom-up analysis, the parameters identified in the first step as flowing into restricted sinks would themselves be treated as restricted sinks at all of their function's call sites, recursively. In addition, function return values can be identified as tainted sources and treated accordingly at each call site. This description ignores such details as recursion and programs such as libraries with multiple call graph roots. It also ignores the large issue of tainted data escaping into the heap or into global or static variables.

Sanitization

For a tainted value to cease being tainted, it must be *sanitized* to ensure that it is in the defined domain of any restricted sink into which it flows. Sanitization is performed by *replacement* or *termination*. In replacement, out-of-domain values are replaced by in-domain values, and processing continues using an in-domain value in place of the original. In termination, the program logic terminates the path of execution when an out-of-domain value is detected, often simply by branching around whatever code would have used the value.

In general, sanitization cannot be recognized exactly using static analysis. Analyzers that perform taint analysis usually provide some extralinguistic mechanism to identify sanitizing functions that sanitize an argument (passed by address) in place, return a sanitized version of an argument, or return a status code indicating whether the argument is in the required domain. Because such extralinguistic mechanisms are outside the scope of this specification, this Technical Specification uses a set of rudimentary definitions of sanitization that is likely to recognize real sanitization but might cause nonsanitizing or ineffectively sanitizing code to be misconstrued as sanitizing. The following definition of *sanitization* presupposes that the analysis is in some way maintaining a set of constraints on each value encountered as the simulated execution progresses: a given path through the code sanitizes a value with respect to a given restricted sink if it restricts the range of that value to a subset of the defined domain of the restricted sink type. For example, sanitization of signed integers with respect to an array index operation must restrict the range of that integer value to numbers between zero and the size of the array minus one.

Tainted source macros

The function-like macros `GET_TAINTED_STRING` and `GET_TAINTED_INTEGER` defined in this section are used in the examples in this Technical Specification to represent one possible method to obtain a tainted string and tainted integer.

```
#define GET_TAINTED_STRING(buf, buf_size)     \
  do {                                        \
    const char *taint = getenv("TAINT");      \
    if (taint == 0) {                         \
      exit(1);                                \
    }                                         \
                                              \
    size_t taint_size = strlen(taint) + 1;    \
    if (taint_size > buf_size) {              \
      exit(1);                                \
    }                                         \
                                              \
    strncpy(buf, taint, taint_size);          \
  } while (0)

#define GET_TAINTED_INTEGER(type, val)            \
  do {                                            \
    const char *taint = getenv("TAINT");          \
    if (taint == 0) {                             \
      exit(1);                                    \
    }                                             \
                                                  \
    errno = 0;                                    \
    long tmp = strtol(taint, 0, 10);              \
    if ((tmp == LONG_MIN || tmp == LONG_MAX) &&   \
        errno == ERANGE)                          \
      ; // retain LONG_MIN or LONG_MAX            \
    val = tmp & ~(type)0;                         \
  } while (0)
```

1 Scope

This document specifies

- rules for secure coding in the C programming language and
- code examples.

This document does not specify

- the mechanism by which these rules are enforced or
- any particular coding style to be enforced. (It has been impossible to develop a consensus on appropriate style guidelines. Programmers should define style guidelines and apply these guidelines consistently. The easiest way to consistently apply a coding style is with the use of a code formatting tool. Many interactive development environments provide such capabilities.)

Each rule in this document is accompanied by code examples. Code examples are informative only and serve to clarify the requirements outlined in the normative portion of the rule. Examples impose no normative requirements.

Two distinct kinds of examples are provided:

— *noncompliant examples* demonstrating language constructs that have weaknesses with potentially exploitable security implications; such examples are expected to elicit a diagnostic from a conforming analyzer for the affected language construct; and
— *compliant examples* are expected not to elicit a diagnostic.

Examples are not intended to be complete programs. For the sake of brevity, they typically omit `#include` directives of C Standard Library headers that would otherwise be necessary to provide declarations of referenced symbols. Code examples may also declare symbols without providing their definitions if the definitions are not essential for demonstrating a specific weakness.

2 Conformance

In this Technical Specification, “shall” is to be interpreted as a requirement on an analyzer; conversely, “shall not” is to be interpreted as a prohibition.

Various types of programs (such as compilers or specialized analyzers) can be used to to check if the C source of a program contains any violations of the coding rules specified in this Technical Specification. In this Technical Specification all such checking programs are called *analyzers*. An analyzer can claim conformity with this Technical Specification, programs (i.e., C sources) that do not yield any diagnostic when applied to a conforming analyzer cannot claim conformity to this Technical Specification.

A conforming analyzer shall diagnose all violations of coding rules specified in this Technical Specification. These rules may be extended in an implementation-dependent manner.

A conforming analyzer shall issue at least one diagnostic for a strictly conforming C program containing one or more violations of the rules in this specification.

NOTE: This Technical Specification does not require an analyzer to produce a diagnostic message for any violation of any syntax rule or constraint specified by the C standard.

For each distinct rule in this Technical Specification, a conforming analyzer shall be capable of producing a distinct diagnostic.

NOTE The diagnostic message might be of the form:

```
Accessing freed memory in function abc, file xyz.c, line nnn.
```

Conformance is defined only with respect to source code that is visible to the analyzer. Binary-only libraries, and calls to them, are outside the scope of these rules.

The rules in this Technical Specification are formulated in the most general form, but the security aspects of a rule may depend on actual implementation-defined aspects of the target platform (examples are the sizes of the various integer types and the signedness of the type `char`). A conforming analyzer may be informed (in an analyzer implementation-defined way) about the C language implementation-defined aspects of the target platform that may have effect on the applicability of a rule; in such a case an conforming analyzer need not to report on those rules that do not create a security risk for this particular target platform.

EXAMPLE

```
long i;
printf("i = %d", i);
```

This example can produce a diagnostic, such as the mismatch between `%d` and `long int`. This mismatch might not be a problem for all target implementations, but it is a portability problem because not all implementations have the same representation for `int` and `long`.