FIO34-C. Distinguish between characters read from a file and EOF or WEOF

The EOF macro represents a negative value that is used to indicate that the file is exhausted and no data remains when reading data from a file. EOF is an example of an in-band error indicator. In-band error indicators are problematic to work with, and the creation of new in-band-error indicators is discouraged by ERR02-C. Avoid in-band error indicators.

The byte I/O functions `fgetc()`, `getc()`, and `getchar()` all read a character from a stream and return it as an `int`. (See STR00-C. Represent characters using an appropriate type.) If the stream is at the end of the file, the end-of-file indicator for the stream is set and the function returns EOF. If a read error occurs, the error indicator for the stream is set and the function returns EOF. If these functions succeed, they cast the character returned into an `unsigned char`.

Because EOF is negative, it should not match any unsigned character value. However, this is only true for implementations where the `int` type is wider than `char`. On an implementation where `int` and `char` have the same width, a character-reading function can read and return a valid character that has the same bit-pattern as EOF. This could occur, for example, if an attacker inserted a value that looked like EOF into the file or data stream to alter the behavior of the program.

The C Standard requires only that the `int` type be able to represent a maximum value of +32767 and that a `char` type be no larger than an `int`. Although uncommon, this situation can result in the integer constant expression EOF being indistinguishable from a valid character; that is, `(int)(unsigned char)65535 == -1`. Consequently, failing to use `feof()` and `ferror()` to detect end-of-file and file errors can result in incorrectly identifying the EOF character on rare implementations where `sizeof(int) == sizeof(char)`.

This problem is much more common when reading wide characters. The `fgetwc()`, `getwc()`, and `getwchar()` functions return a value of type `wint_t`. This value can represent the next wide character read, or it can represent WEOF, which indicates end-of-file for wide character streams. On most implementations, the `wchar_t` type has the same width as `wint_t`, and these functions can return a character indistinguishable from WEOF.

In the UTF-16 character set, `0xFFFF` is guaranteed not to be a character, which allows WEOF to be represented as the value `-1`. Similarly, all UTF-32 characters are positive when viewed as a signed 32-bit integer. All widely used character sets are designed with at least one value that does not represent a character. Consequently, it would require a custom character set designed without consideration of the C programming language for this problem to occur with wide characters or with ordinary characters that are as wide as `int`.

The C Standard `feof()` and `ferror()` functions are not subject to the problems associated with character and integer sizes and should be used to verify end-of-file and file errors for susceptible implementations [Kettlewell 2002]. Calling both functions on each iteration of a loop adds significant overhead, so a good strategy is to temporarily trust EOF and WEOF within the loop but verify them with `feof()` and `ferror()` following the loop.

Noncompliant Code Example

This noncompliant code example loops while the character c is not EOF:

```
#include <stdio.h>

void func(void) {
  int c;

  do {
    c = getchar();
  } while (c != EOF);
}
```

Although EOF is guaranteed to be negative and distinct from the value of any unsigned character, it is not guaranteed to be different from any such value when converted to an `int`. Consequently, when `int` has the same width as `char`, this loop may terminate prematurely.

Compliant Solution (Portable)

This compliant solution uses `feof()` to test for end-of-file and `ferror()` to test for errors:

```
#include <stdio.h>

void func(void) {
  int c;

  do {
    c = getchar();
  } while (c != EOF);
  if (feof(stdin)) {
    /* Handle end of file */
  } else if (ferror(stdin)) {
    /* Handle file error */
  } else {
    /* Received a character that resembles EOF; handle error */
  }
}
```

Noncompliant Code Example (Nonportable)

This noncompliant code example uses an assertion to ensure that the code is executed only on architectures where `int` is wider than `char` and `EOF` is guaranteed not to be a valid character value. However, this code example is noncompliant because the variable `c` is declared as a `char` rather than an `int`, making it possible for a valid character value to compare equal to the value of the `EOF` macro when `char` is signed because of sign extension:

```
#include <assert.h>
#include <limits.h>
#include <stdio.h>

void func(void) {
  char c;
  static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation");

  do {
    c = getchar();
  } while (c != EOF);
}
```

Assuming that a `char` is a signed 8-bit type and an int is a 32-bit type, if `getchar()` returns the character value `'\xff` (decimal 255), it will be interpreted as `EOF` because this value is sign-extended to `0xFFFFFFFF` (the value of `EOF`) to perform the comparison. (See STR34-C. Cast characters to unsigned char before converting to larger integer sizes.)

Compliant Solution (Nonportable)

This compliant solution declares `c` to be an `int`. Consequently, the loop will terminate only when the file is exhausted.

```
#include <assert.h>
#include <stdio.h>
#include <limits.h>

void func(void) {
  int c;
  static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation");

  do {
    c = getchar();
  } while (c != EOF);
}
```

Noncompliant Code Example (Wide Characters)

In this noncompliant example, the result of the call to the C standard library function `getwc()` is stored into a variable of type `wchar_t` and is subsequently compared with `WEOF`:

```
#include <stddef.h>
#include <stdio.h>
#include <wchar.h>

enum { BUFFER_SIZE = 32 };

void g(void) {
  wchar_t buf[BUFFER_SIZE];
  wchar_t wc;
  size_t i = 0;

  while ((wc = getwc(stdin)) != L'\n' && wc != WEOF) {
    if (i < (BUFFER_SIZE - 1)) {
      buf[i++] = wc;
    }
  }
  buf[i] = L'\0';
}
```

This code suffers from two problems. First, the value returned by `getwc()` is immediately converted to `wchar_t` before being compared with `WEOF`. Second, there is no check to ensure that `wint_t` is wider than `wchar_t`. Both of these problems make it possible for an attacker to terminate the loop prematurely by supplying the wide-character value matching `WEOF` in the file.

Compliant Solution (Portable)

This compliant solution declares c to be a `wint_t` to match the integer type returned by `getwc()`. Furthermore, it does not rely on `WEOF` to determine end-of-file definitively.

```
#include <stddef.h>
#include <stdio.h>
#include <wchar.h>

enum {BUFFER_SIZE = 32 }

void g(void) {
  wchar_t buf[BUFFER_SIZE];
  wint_t wc;
  size_t i = 0;

  while ((wc = getwc(stdin)) != L'\n' && wc != WEOF) {
    if (i < BUFFER_SIZE - 1) {
      buf[i++] = wc;
    }
  }

  if (feof(stdin) || ferror(stdin)) {
   buf[i] = L'\0';
  } else {
    /* Received a wide character that resembles WEOF; handle error */
  }
}
```

Exceptions

FIO34-C-EX1: A number of C functions do not return characters but can return `EOF` as a status code. These functions include `fclose()`, `fflush()`, `fputs()`, `fscanf()`, `puts()`, `scanf()`, `sscanf()`, `vfscanf()`, and `vscanf()`. These return values can be compared to `EOF` without validating the result.

Risk Assessment

Incorrectly assuming characters from a file cannot match `EOF` or `WEOF` has resulted in significant vulnerabilities, including command injection attacks. (See the *CA-1996-22 advisory.)

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO34-C	High	Probable	Medium	P12	L1

Automated Detection

Tool	Version	Checker	Description
Axivion Bauhaus Suite	6.9.0	**CertC-FIO34**	
CodeSonar	5.2p0	**LANG.CAST. COERCE**	Coercion alters value
Compass /ROSE			
Coverity	2017.07	**CHAR_IO**	Identifies defects when the return value of `fgetc()`, `getc()`, or `getchar()` is incorrectly assigned to a `char` instead of an `int`. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary
ECLAIR	1.2	**CC2.FIO34**	Partially implemented
Klocwork	2018	**CWARN.CMPCHR. EOF**	
LDRA tool suite	9.7.1	**662 S**	Fully implemented
Parasoft C /C++test	10.4.2	**CERT_C-FIO34-a**	Avoid implicit conversions from wider to narrower types
Polyspace Bug Finder	R2019b	CERT C: Rule FIO34-C	Checks for character values absorbed into EOF (rule partially covered)
PRQA QA-C	9.7	**2676, 2678**	
PRQA QA-C++	4.4	**2676, 2678, 3001, 3010, 3051, 3137, 3717**	
Splint	3.1.1		

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT C Secure Coding Standard	STR00-C. Represent characters using an appropriate type	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT C Secure Coding Standard	INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT Oracle Secure Coding Standard for Java	FIO08-J. Use an int to capture the return value of methods that read a character or byte	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TS 17961:2013	Using character values that are indistinguishable from EOF [chreof]	Prior to 2018-01-12: CERT: Unspecified Relationship
CWE 2.11	CWE-197	2017-06-14: CERT: Rule subset of CWE

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-197 and FIO34-C

Independent(FLP34-C, INT31-C) FIO34-C = Subset(INT31-C)

Therefore: FIO34-C = Subset(CWE-197)

Bibliography

[Kettlewell 2002]	Section 1.2, "<stdio.h> and Character Types"
[NIST 2006]	SAMATE Reference Dataset Test Case ID 000-000-088
[Summit 2005]	Question 12.2