Secure Coding Lab Experiment - 12
Rule 09. Input Output (FIO)

FIO30-C. Exclude user input from format strings

- *Never call a formatted I/O function with a format string containing a tainted value.*
- *An attacker who can control the format string can crash the process, view stack contents, view memory content, or write to an arbitrary memory location.*
- *The attacker can execute arbitrary code with the permissions of the vulnerable process.*
- *Formatted output functions are particularly dangerous because many programmers are unaware of their full capabilities.*
- *Formatted output functions can be used to write an integer value to a specified address using the %n conversion specifier.*

Noncompliant Code Example

The incorrect_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user. This is an exemplar of untrusted data that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard fprintf() function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password_noncompliant(const char *user) {
   int ret;
   /* User names are restricted to 256 or fewer characters */
   static const char msg_format[] = "%s cannot be authenticated.\n";
   size_t len = strlen(user) + sizeof(msg_format);
   char *msg = (char *)malloc(len);
   if (msg == NULL) {
      fprintf(stderr, "Memory allocation failed.\n");
      return;
   }
   ret = snprintf(msg, len, msg_format, user);
   if (ret < 0) {
      fprintf(stderr, "Error formatting message.\n");
      free(msg);
      return;
   } else if (ret >= len) {
      fprintf(stderr, "Truncated output.\n");
      free(msg);
      return;
   }
   fprintf(stderr, msg);
```

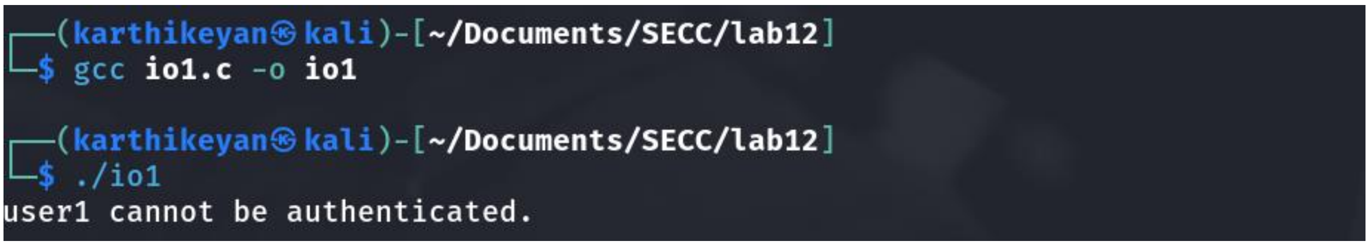
```
    free(msg);
}

int main() {
    const char *username = "user1";
    incorrect_password_noncompliant(username);
    return 0;
}
```

The incorrect_password() function calculates the size of the message, allocates dynamic storage, and then constructs the message in the allocated memory using the snprintf() function. The addition operations are not checked for integer overflow because the string referenced by user is known to have a length of 256 or less. Because the %s characters are replaced by the string referenced by user in the call to snprintf(), the resulting string needs 1 byte less than is allocated. The snprintf() function is commonly used for messages that are displayed in multiple locations or messages that are difficult to build. However, the resulting code contains a format-string vulnerability because the msg includes untrusted user input and is passed as the format-string argument in the call to fprintf().

Compliant Solution (fputs())

This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr without evaluating its contents:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password_fputs(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        return;
    }
    ret = snprintf(msg, len, msg_format, user);
    if (ret < 0) {
        fprintf(stderr, "Error formatting message.\n");
        free(msg);
        return;
    } else if (ret >= len) {
        fprintf(stderr, "Truncated output.\n");
        free(msg);
        return;
    }
    fputs(msg, stderr);
    free(msg);
}

int main() {
    const char *username = "user1";
    incorrect_password_fputs(username);
    return 0;
}
```

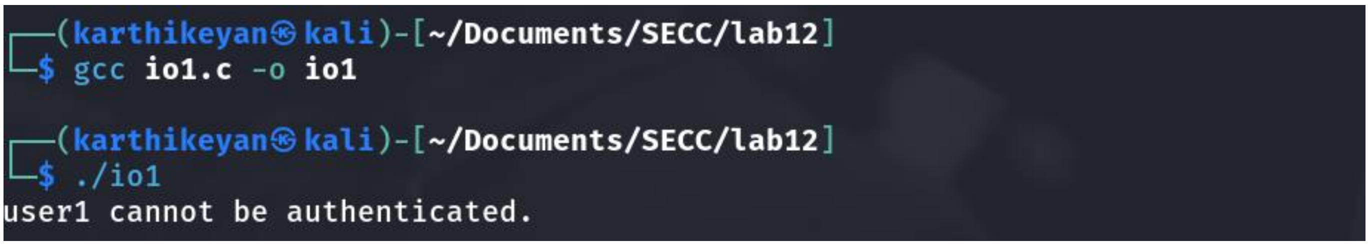


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io1.c -o io1

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io1
user1 cannot be authenticated.
```

Compliant Solution (fprintf())

This compliant solution passes the untrusted user input as one of the variadic arguments to fprintf() and not as part of the format string, eliminating the possibility of a format-string vulnerability:

```
#include <stdio.h>

void incorrect_password_compliant(const char *user) {

    static const char msg_format[] = "%s cannot be authenticated.\n";

    fprintf(stderr, msg_format, user);

}


int main() {

    const char *username = "example_user";

    incorrect_password_compliant(username);

    return 0;

}
```



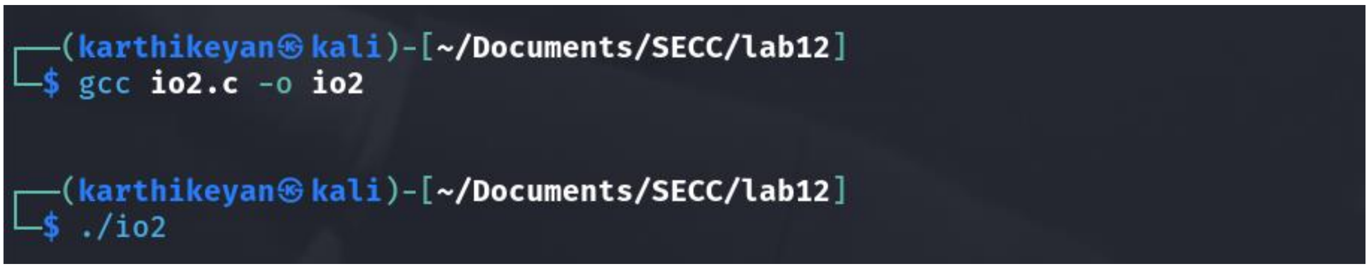
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io1.c -o io1

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io1
user1 cannot be authenticated.
```

Noncompliant Code Example (POSIX)

This noncompliant code example is similar to the first noncompliant code example but uses the POSIX function syslog() [IEEE Std 1003.1:2013] instead of the fprintf() function. The syslog() function is also susceptible to format-string vulnerabilities.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <syslog.h>
void incorrect_password_syslog_noncompliant(const char *user) {
   int ret;
   /* User names are restricted to 256 or fewer characters */
   static const char msg_format[] = "%s cannot be authenticated.\n";
   size_t len = strlen(user) + sizeof(msg_format);
   char *msg = (char *)malloc(len);
   if (msg == NULL) {
      syslog(LOG_ERR, "Memory allocation failed.");
      return;
  }
   ret = snprintf(msg, len, msg_format, user);
   if (ret < 0) {
      syslog(LOG_ERR, "Error formatting message.");
      free(msg);
      return;
   } else if (ret >= len) {
      syslog(LOG_ERR, "Truncated output.");
      free(msg);
      return;
   }
   syslog(LOG_INFO, msg);
   free(msg);
}
int main() {
   const char *username = "user1";
   incorrect_password_syslog_noncompliant(username);
   return 0;
}
```



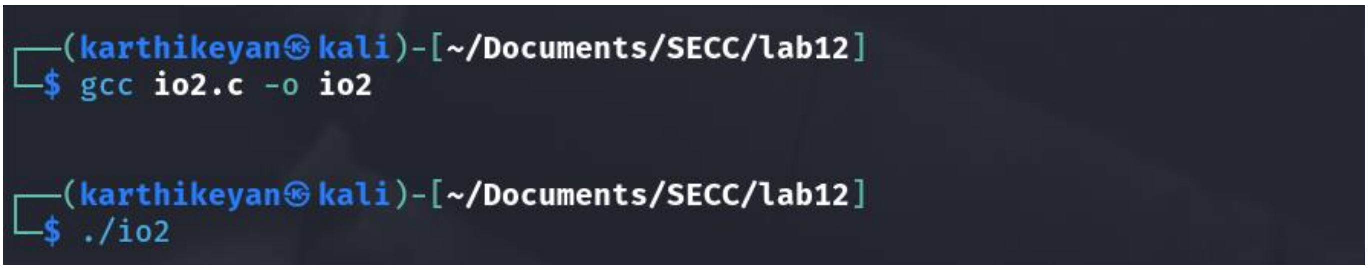
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io2.c -o io2

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io2
```

The syslog() function first appeared in BSD 4.2 and is supported by Linux and other modern UNIX implementations. It is not available on Windows systems.

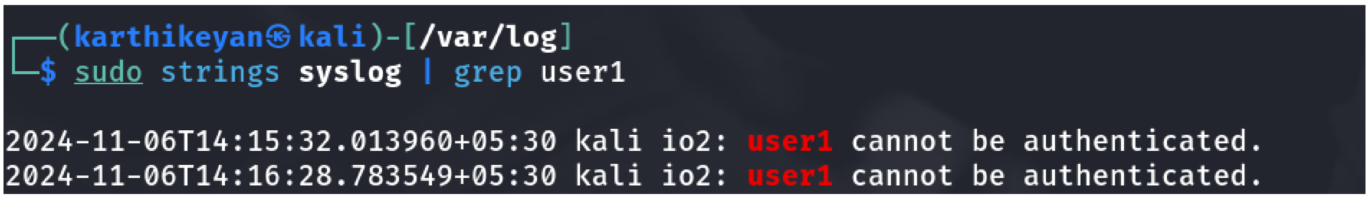
Compliant Solution (POSIX)

This compliant solution passes the untrusted user input as one of the variadic arguments to syslog() instead of including it in the format string:

```
#include <syslog.h>
void incorrect_password_syslog_compliant(const char *user) {
   static const char msg_format[] = "%s cannot be authenticated.\n";
   syslog(LOG_INFO, msg_format, user);
}
int main() {
   const char *username = "user1";
   incorrect_password_syslog_compliant(username);
   return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io2.c -o io2

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io2
```



```
┌──(karthikeyan㉿kali)-[/var/log]
└─$ sudo strings syslog | grep user1

2024-11-06T14:15:32.013960+05:30 kali io2: user1 cannot be authenticated.
2024-11-06T14:16:28.783549+05:30 kali io2: user1 cannot be authenticated.
```

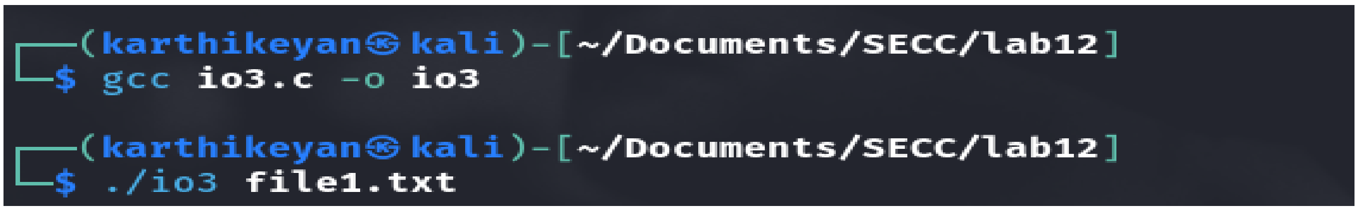
FIO32-C: Do Not Perform Operations on Devices That Are Only Appropriate for Files

- *File names on many operating systems, like Windows and UNIX, can access special files (devices).*
- *Operations intended for regular files on device files can cause crashes and denial-of-service attacks.*
- *For example, improperly handling device names can lead to invalid resource access and system crashes.*
- *Access to certain device files, such as /dev/kmem, can allow attackers to manipulate process attributes or crash systems.*
- *On Linux, opening certain devices can lead to application locks, creating vulnerabilities.*

Noncompliant Code Example

In this noncompliant code example, the user can specify a locked device or a FIFO (first-in, first-out) file name, which can cause the program to hang on the call to fopen():

```
#include <stdio.h>
#include <stdlib.h>

void func(const char *file_name) {
   FILE *file;
   if ((file = fopen(file_name, "wb")) == NULL) {
      perror("Error opening file");
      exit(EXIT_FAILURE); // Handle error
   }

   // Operate on the file

   if (fclose(file) == EOF) {
      perror("Error closing file");
      exit(EXIT_FAILURE); // Handle error
   }
}

int main(int argc, char *argv[]) {
   if (argc != 2) {
      fprintf(stderr, "Usage: %s <file_name>\n", argv[0]);
      return EXIT_FAILURE; // Handle error
   }
   func(argv[1]);
   return EXIT_SUCCESS;
}
```

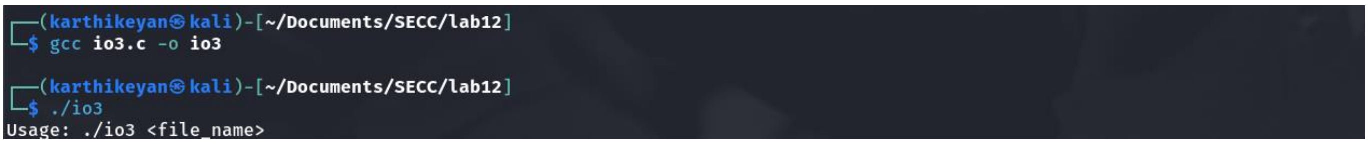


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io3.c -o io3

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io3 file1.txt
```

Compliant Solution (POSIX)

- *Use the O_NONBLOCK flag with open() to avoid delays in operations.*
- *Opening a FIFO with O_NONBLOCK allows immediate return for reading and fails for writing if no reader is present.*
- *For special files, the O_NONBLOCK behavior is device-specific.*
- *Verify file types with lstat() and fstat() to ensure proper handling.*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#ifdef O_NOFOLLOW
  #define OPEN_FLAGS O_NOFOLLOW | O_NONBLOCK
#else
  #define OPEN_FLAGS O_NONBLOCK
#endif

void func(const char *file_name) {
    struct stat orig_st;
    struct stat open_st;
    int fd;
    int flags;

    if ((lstat(file_name, &orig_st) != 0) || (!S_ISREG(orig_st.st_mode))) {
        perror("Error with lstat or not a regular file");
        exit(EXIT_FAILURE); // Handle error
    }

    // Race window

    fd = open(file_name, OPEN_FLAGS | O_WRONLY);
    if (fd == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE); // Handle error
    }

    if (fstat(fd, &open_st) != 0) {
        perror("Error with fstat");
        exit(EXIT_FAILURE); // Handle error
    }

    if ((orig_st.st_mode != open_st.st_mode) ||
        (orig_st.st_ino  != open_st.st_ino) ||
        (orig_st.st_dev  != open_st.st_dev)) {
        fprintf(stderr, "The file was tampered with\n");
        close(fd);
        exit(EXIT_FAILURE); // Handle error
    }
    // Optional: drop the O_NONBLOCK now that we are sure this is a good file.
    if ((flags = fcntl(fd, F_GETFL)) == -1) {
        perror("Error getting file flags");
        exit(EXIT_FAILURE); // Handle error
    }

    if (fcntl(fd, F_SETFL, flags & ~O_NONBLOCK) == -1) {
        perror("Error setting file flags");
        exit(EXIT_FAILURE); // Handle error
    }

    // Operate on the file

    if (close(fd) == -1) {
        perror("Error closing file");
        exit(EXIT_FAILURE); // Handle error
    }
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file_name>\n", argv[0]);
        return EXIT_FAILURE; // Handle error
    }
    func(argv[1]);
    return EXIT_SUCCESS;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io3.c -o io3

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io3
Usage: ./io3 <file_name>
```

This code has a TOCTOU race condition where an attacker can alter the file after checking with lstat() but before open()

FIO34-C: Do Not Rely on EOF or WEOF as a Valid Return Value

The EOF (End-of-File) macro is utilized in C to signify that no more data is available for reading from a file. While EOF is negative and distinct from valid character values, complications can arise when dealing with character data streams. These complications are particularly significant on platforms where the int type is the same width as char, leading to potential ambiguities when reading characters.

Issues with In-Band Error Indicators

- *Ambiguity: An attacker might insert a value in a data stream that has the same bit pattern as EOF, causing logical errors in the application.*

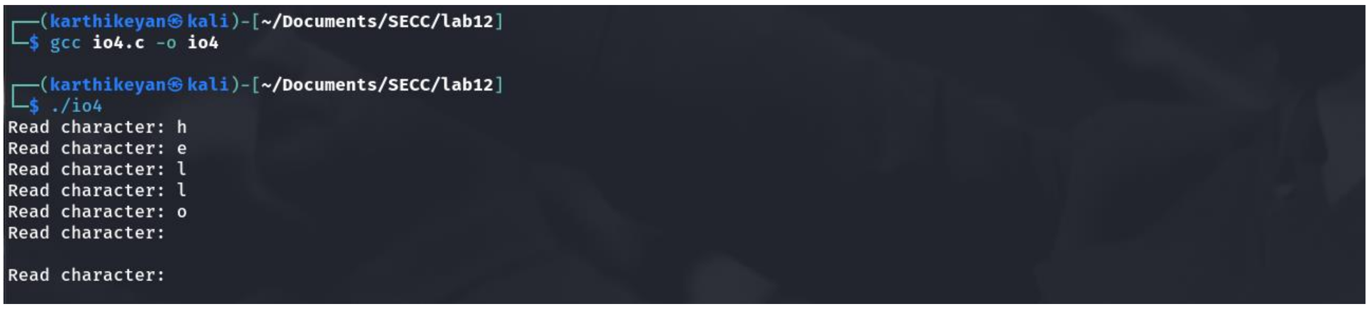
- *Standard Compliance: The C Standard requires only that int can represent values up to +32767, which might create situations where EOF can be indistinguishable from a valid character.*
- *Wide Character Reading: Similar issues arise when reading wide characters, where WEOF might conflict with valid character values.*

To correctly handle EOF and prevent security vulnerabilities, it is essential to use the feof() and ferror() functions after an I/O operation, rather than relying solely on EOF or WEOF.

Noncompliant Code Example

The following code example demonstrates improper handling of EOF:

```
#include <stdio.h>

void func(void) {
   int c;
   FILE *fp = fopen("input.txt", "r");
   if (fp == NULL) {
      printf("Failed to open file.\n");
      return;
   }
   do {
      c = fgetc(fp);
      if (c != EOF) {
         printf("Read character: %c\n", c);
      }
   } while (c != EOF || (!feof(fp) && !ferror(fp)));
   fclose(fp);
}

int main() {
   func();
   return 0;
}
```

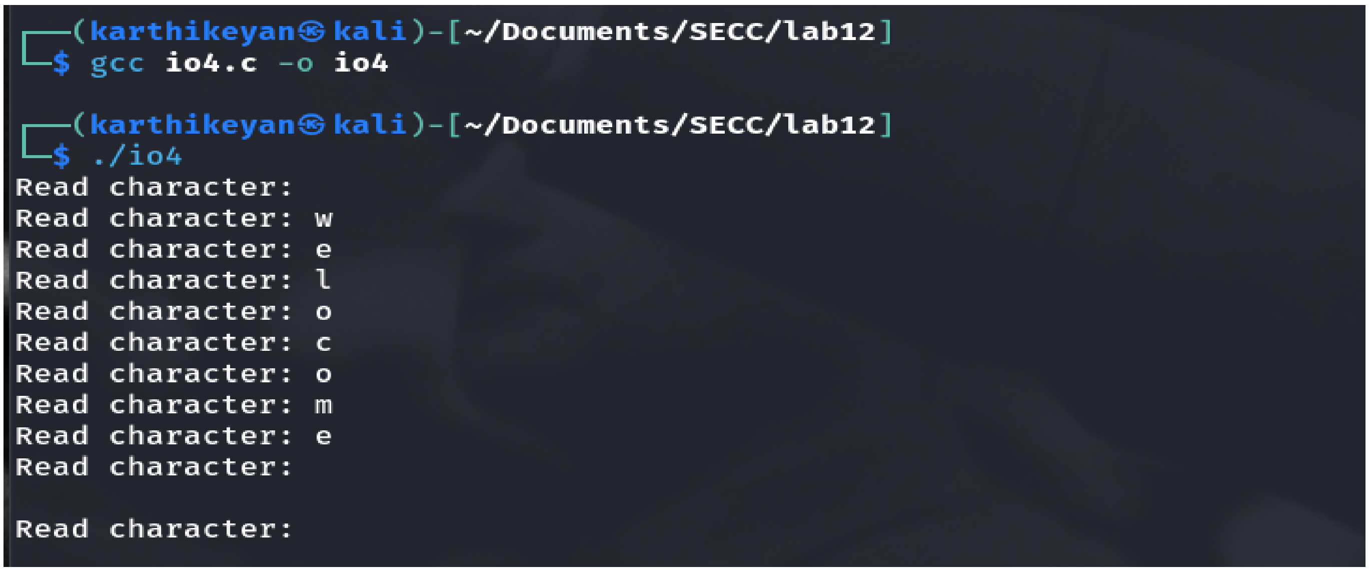


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io4.c -o io4

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io4
Read character: h
Read character: e
Read character: l
Read character: l
Read character: o
Read character: 

Read character: 
```

Compliant Solution (Portable)

The following compliant solution correctly verifies EOF by using feof() and ferror() after the loop:

```
#include <stdio.h>

void func(void) {
   int c;
   FILE *fp = fopen("input.txt", "r");
   if (fp == NULL) {
      printf("Failed to open file.\n");
      return;
   }
   do {
      c = fgetc(fp);
      if (c != EOF) {
         printf("Read character: %c\n", c);
      }
   } while (c != EOF || (!feof(fp) && !ferror(fp)));
   fclose(fp);
}

int main() {
   func();
   return 0;
}
```

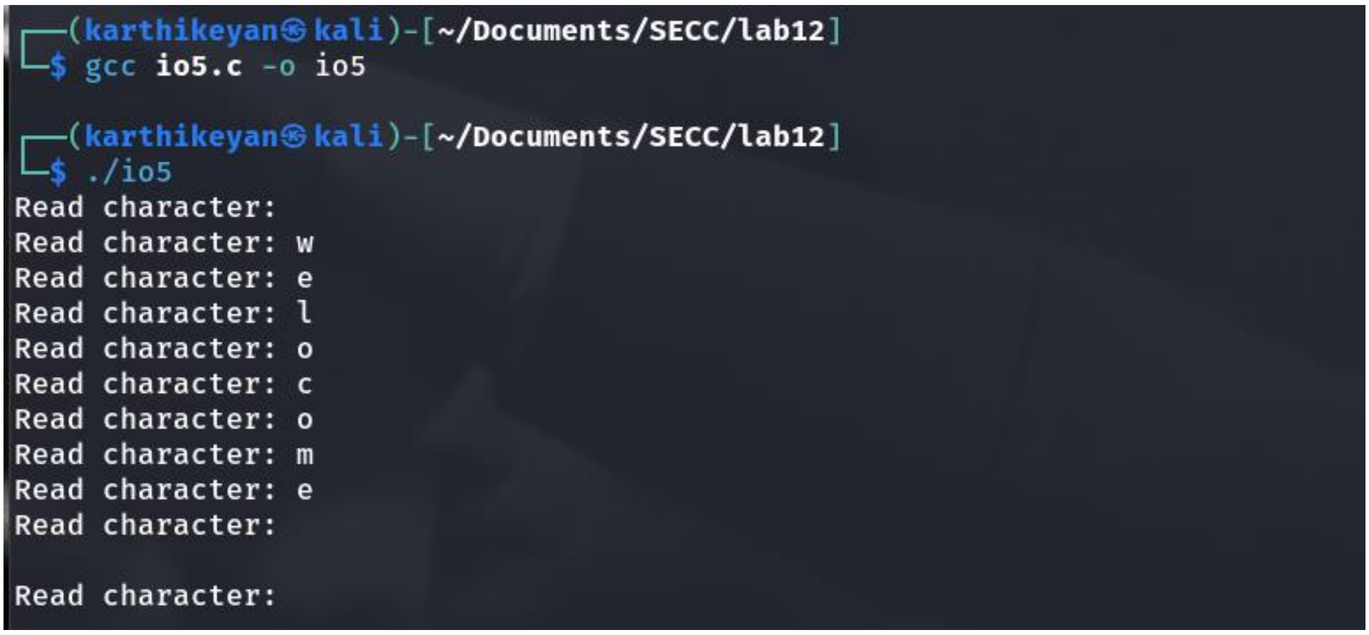


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io4.c -o io4

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io4
Read character:
Read character: w
Read character: e
Read character: l
Read character: o
Read character: c
Read character: o
Read character: m
Read character: e
Read character:

Read character:
```

Noncompliant Code Example (Nonportable)

In this example, an assertion is used to validate that int is wider than char, but it remains noncompliant due to potential value misinterpretation:

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

void func(void) {
   char c;
   FILE *fp = fopen("input.txt", "r");
   if (fp == NULL) {
      printf("Failed to open file.\n");
      return;
   }
   static_assert(UCHAR_MAX < UINT_MAX, "FIO34 -C violation");
   do {
      c = fgetc(fp);
      if (c != EOF) {
         printf("Read character: %c\n", c);
      }
   } while (c != EOF);
   fclose(fp);
}

int main() {
   func();
   return 0;
}
```

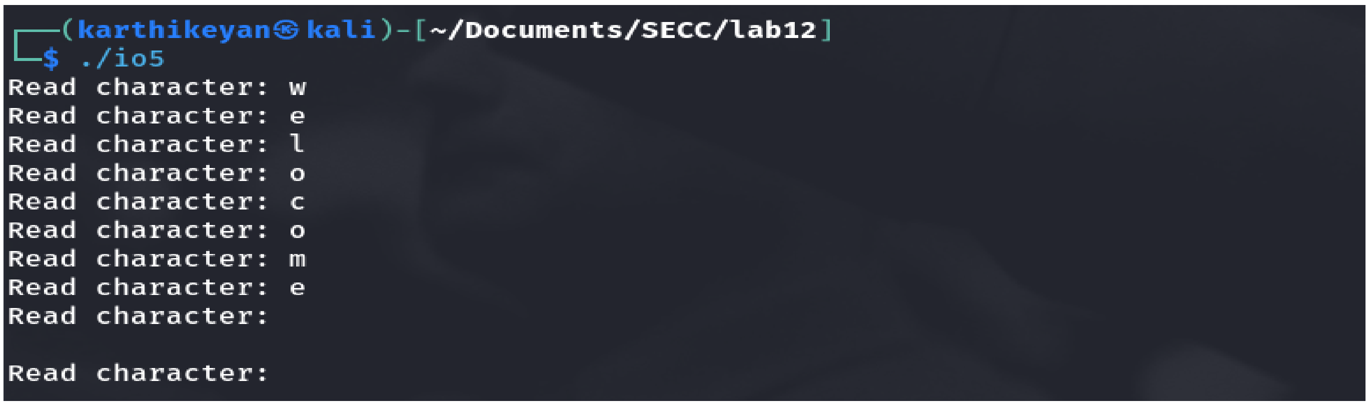


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io5.c -o io5

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io5
Read character: 
Read character: w
Read character: e
Read character: l
Read character: o
Read character: c
Read character: o
Read character: m
Read character: e
Read character: 

Read character: 
```

Compliant Solution (Nonportable)

To avoid the pitfalls mentioned above, c is declared as int, ensuring the loop only terminates upon true EOF:

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

void func(void) {
    int c;
    FILE *fp = fopen("input.txt", "r");
    if (fp == NULL) {
        printf("Failed to open file.\n");
        return;
    }
    static_assert(UCHAR_MAX < UINT_MAX, "FIO34 -C violation");
    do {
        c = fgetc(fp);
        if (c != EOF) {
            printf("Read character: %c\n", c);
        }
    } while (c != EOF);
    fclose(fp);
}

int main() {
    func();
    return 0;
}
```

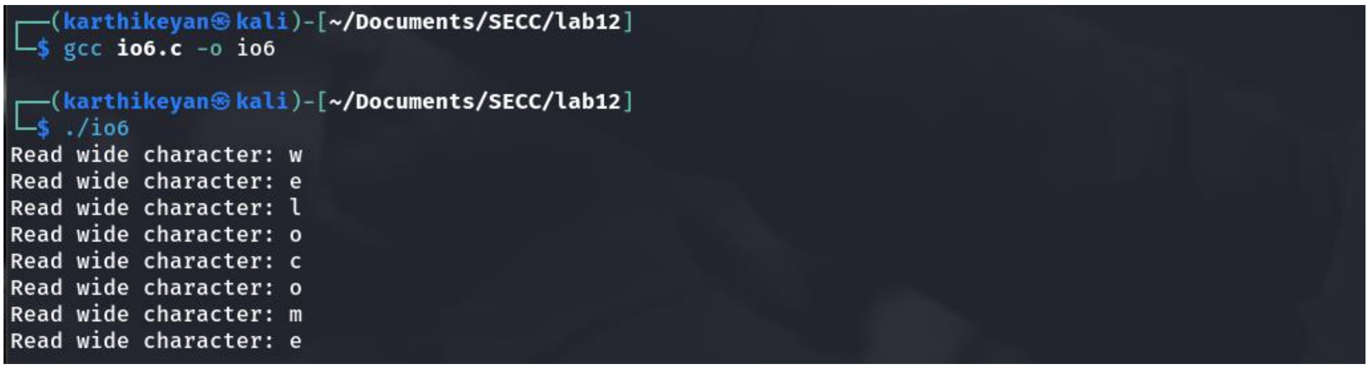


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io5
Read character: w
Read character: e
Read character: l
Read character: o
Read character: c
Read character: o
Read character: m
Read character: e
Read character: 

Read character:
```

Noncompliant Code Example (Wide Characters)

This example uses wchar_t for reading wide characters and fails to handle WEOF correctly:

```
#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;
   FILE *fp = fopen("input.txt", "r");
   if (fp == NULL) {
      printf("Failed to open file.\n");
      return;
   }
   while ((wc = getwc(fp)) != L'\n' && wc != WEOF) {
      if (i < BUFFER_SIZE - 1) {
         buf[i++] = wc;
         wprintf(L"Read wide character: %lc\n", wc);
      }
   }
   buf[i] = L'\0';
   fclose(fp);
}

int main() {
   g();
   return 0;
}
```

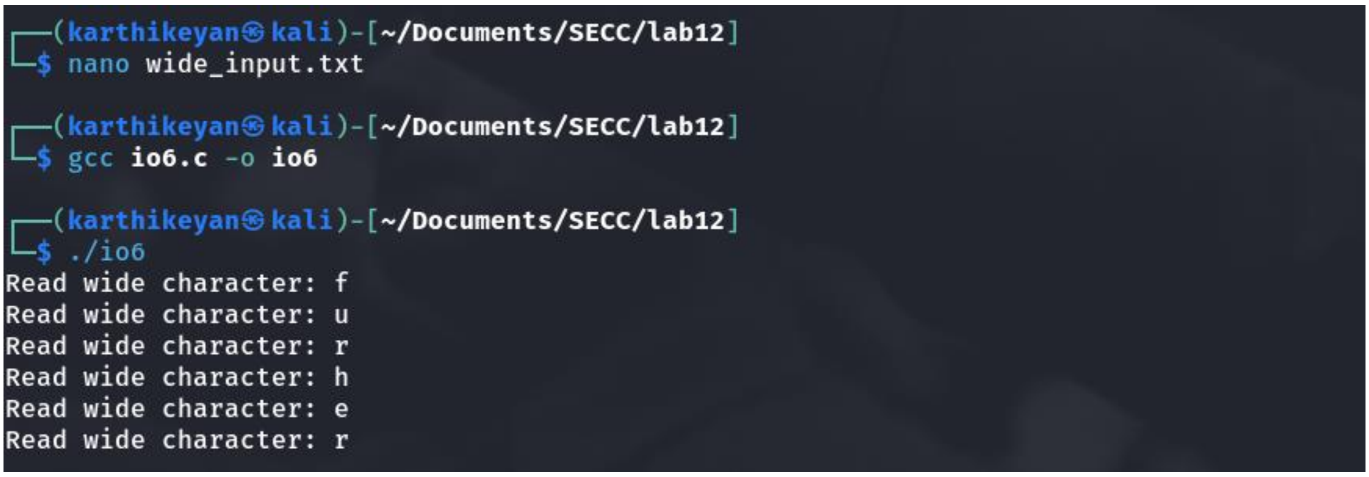


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io6.c -o io6

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io6
Read wide character: w
Read wide character: e
Read wide character: l
Read wide character: o
Read wide character: c
Read wide character: o
Read wide character: m
Read wide character: e
```

Compliant Solution (Portable)

In the compliant solution, wc is declared as wint_t to match the return type of getwc(). Additionally, checks for EOF are performed properly:

```
#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;
  FILE *fp = fopen("wide_input.txt", "r");
  if (fp == NULL) {
    printf("Failed to open file.\n");
    return;
  }
  while ((wc = getwc(fp)) != L'\n' && wc != WEOF) {
    if (i < BUFFER_SIZE - 1) {
      buf[i++] = wc;
      wprintf(L"Read wide character: %lc\n", wc);
    }
  }
  if (feof(fp) || ferror(fp)) {
    buf[i] = L'\0';
    printf("End of file or error encountered.\n");
  } else {
    printf("Received a wide character matching WEOF; handling error.\n");
  }
  fclose(fp);
}
int main() {
  g();
  return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ nano wide_input.txt

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io6.c -o io6

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io6
Read wide character: f
Read wide character: u
Read wide character: r
Read wide character: h
Read wide character: e
Read wide character: r
```

When dealing with character streams, especially in systems where data types may have overlapping values, always employ feof() and ferror() to verify EOF and error conditions after reading operations. This approach enhances robustness and prevents vulnerabilities due to ambiguities in character representations.

FIO37-C. Do not assume that fgets() or fgetws() returns a nonempty string when successful

When reading data using fgets() or fgetws(), it's important to recognize that:

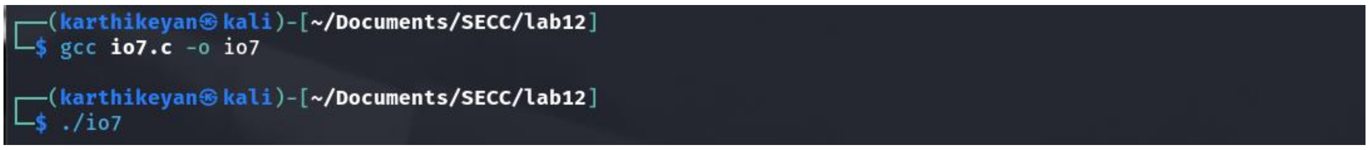
- *These functions return a pointer to the buffer if successful, or a null pointer if an error occurs or if the end of the file is reached without reading any characters.*
- *It is erroneous to assume that the buffer contains a non-empty string since it may include null characters, especially when reading binary data.*

Noncompliant Code Example

This code attempts to remove the trailing newline from input but can lead to a write-outside-array-bounds error if the first character read is a null character.

```
#include <stdio.h>
#include <string.h>

enum { BUFFER_SIZE = 1024 };

void func(void) {
   char buf[BUFFER_SIZE];

   // Simulating static input
   const char *input = "Hello , World!\n";
   strncpy(buf, input, sizeof(buf) - 1);
   buf[sizeof(buf) - 1] = '\0'; // Ensure null termination

   if (buf[0] == '\0') {
      /* Handle error */
      printf("Error: No input received.\n");
      return;
   }
```

```
    // Unsafe operation: assumes buf is a non-empty string
    buf[strlen(buf) - 1] = '\0'; // May lead to buffer overflow if buf starts with '\0'
}

int main() {
    func();
    return 0;
}
```



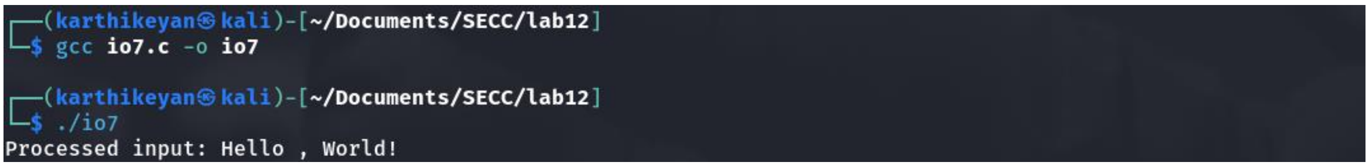
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io7.c -o io7

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io7
```

If fgets() reads a null character first, strlen(buf) returns 0, causing strlen(buf) - 1 to wrap around and potentially access out of bounds.

Compliant Solution

This compliant solution uses strchr() to safely replace the newline character with a null terminator if it exists.

```
#include <stdio.h>
#include <string.h>

enum { BUFFER_SIZE = 1024 };

void func(void) {
    char buf[BUFFER_SIZE];
    char *p;

    // Simulating static input
    const char *input = "Hello , World!\n";
    strncpy(buf, input, sizeof(buf) - 1);
    buf[sizeof(buf) - 1] = '\0'; // Ensure null termination

    if (buf[0] == '\0') {
        /* Handle error */
        printf("Error: No input received.\n");
        return;
    }

    // Use strchr to find and replace the newline character
    p = strchr(buf, '\n');
    if (p) {
        *p = '\0'; // Safely replace newline with null terminator
    }

    printf("Processed input: %s\n", buf); // Output the processed input
}

int main() {
    func();
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io7.c -o io7

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io7
Processed input: Hello , World!
```

By checking for a newline character explicitly, the code avoids the risk of accessing out of bounds in the buffer.

The solution can handle cases where binary data might be read, ensuring proper termination of the string without assumptions about its contents.

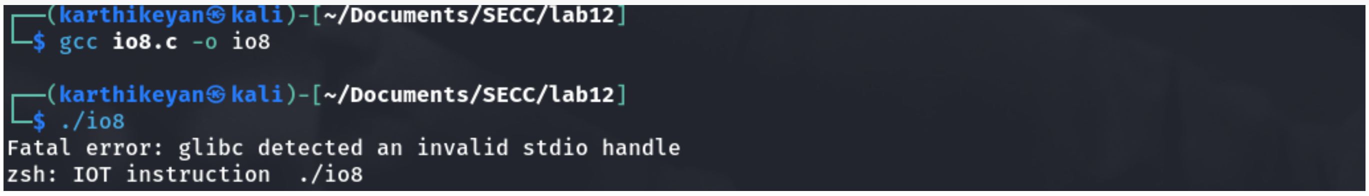
FIO38-C. Do not copy a FILE object

- *The address of the FILE object used to control a stream may be significant; a copy of a FILE object need not serve in place of the original.*
- *Do not copy a FILE object.*

Non-Compliant Code Example

This non-compliant code example can fail because a by-value copy of stdout is being used in the call to fputs():

```
#include <stdio.h>

int main(void) {
    FILE my_stdout = *stdout; // Incorrect: copying FILE object
    // Attempt to write using the copied FILE object
    if (fputs("Hello, World!\n", &my_stdout) == EOF) {
        /* Handle error */
        printf("Error writing to my_stdout.\n");
    }
    // Attempt to read back (not typical but illustrates potential issues)
    char buffer[100];
    if (fgets(buffer, sizeof(buffer), &my_stdout) == NULL) {
        printf("Error reading from my_stdout.\n");
    } else {
        printf("Read from my_stdout: %s", buffer);
    }
    return 0;
}
```



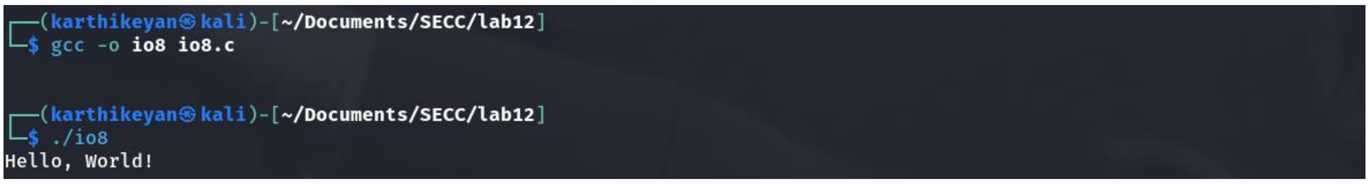
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io8.c -o io8

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io8
Fatal error: glibc detected an invalid stdio handle
zsh: IOT instruction  ./io8
```

The code copies stdout into my_stdout, which is a local FILE object. This can cause access violations or incorrect behavior when attempting to use my_stdout for file operations. Since my_stdout does not maintain the internal state of stdout, writing and reading from it can result in undefined behavior. The program may produce an error due to accessing an invalid FILE object.

Compliant Solution

In this compliant solution, a pointer to the FILE object is used instead of making a copy:

```
#include <stdio.h>
int main(void) {
    FILE *my_stdout = stdout; // Correct: using a pointer to FILE object
    if (fputs("Hello, World!\n", my_stdout) == EOF) {
        /* Handle error */
        printf("Error writing to stdout.\n");
    }
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc -o io8 io8.c

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io8
Hello, World!
```

By using a pointer to stdout, the program avoids copying the FILE object, preventing access violations and undefined behavior. The original FILE object retains its internal state, allowing for safe writing operations without risk of errors related to object state.

FIO39-C. Do not alternately input and output from a stream without an intervening flush or positioning call

Undefined behavior can occur in the following scenarios:

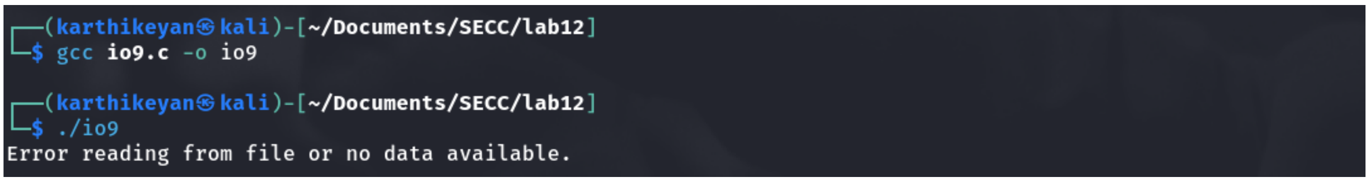
- *Receiving input from a stream immediately after outputting to it, without using fflush(), fseek(), fsetpos(), or rewind(), unless at end-of-file.*
- *Outputting to a stream directly after receiving input from it, also without an intervening call to fseek(), fsetpos(), or rewind(), unless at end-of-file.*

To avoid these issues, always use fflush(), fseek(), or fsetpos() between input and output operations on the same stream. For error checking, prefer fseek() over rewind().

Noncompliant Code Example

This noncompliant code example appends data to a file and then reads from the same file:

```
#include <stdio.h>
enum { BUFFERSIZE = 32 };

void initialize_data(char *data, size_t size) {
   for (size_t i = 0; i < size; i++) {
      data[i] = 'A' + (i % 26); // Initialize with alphabet letters
   }
}

void func(const char *file_name) {
   char data[BUFFERSIZE];
   char append_data[BUFFERSIZE];
   FILE *file;

   file = fopen(file_name, "a+");
   if (file == NULL) {
      printf("Error opening file.\n");
      return;
   }

   initialize_data(append_data, BUFFERSIZE);

   // Attempt to write to the file
   if (fwrite(append_data, sizeof(char), BUFFERSIZE, file) != BUFFERSIZE) {
      printf("Error writing to file.\n");
   }

   // Attempt to read from the same file (undefined behavior)
   if (fread(data, sizeof(char), BUFFERSIZE, file) < BUFFERSIZE) {
      printf("Error reading from file or no data available.\n");
   }

   fclose(file);
}

int main(void) {
   func("example.txt");
   return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io9.c -o io9

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io9
Error reading from file or no data available.
```

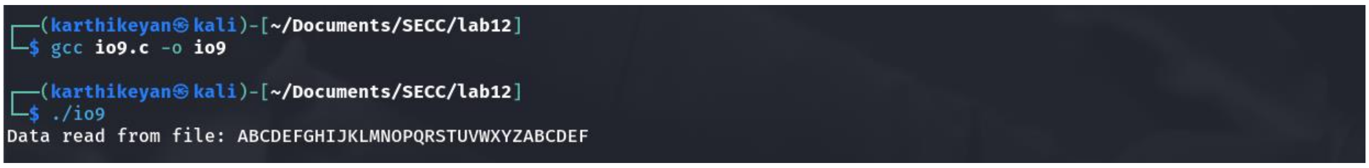
Because there is no intervening flush or positioning call between the calls to fread() and fwrite(), the behavior is undefined.

Compliant Solution

In this compliant solution, fseek() is called between the output and input, eliminating the undefined behavior:

```
#include <stdio.h>
enum { BUFFERSIZE = 32 };

void initialize_data(char *data, size_t size) {
   for (size_t i = 0; i < size; i++) {
       data[i] = 'A' + (i % 26); // Initialize with alphabet letters
   }
}

void func(const char *file_name) {
   char data[BUFFERSIZE];
   char append_data[BUFFERSIZE];
   FILE *file;

   file = fopen(file_name, "a+");
   if (file == NULL) {
       printf("Error opening file.\n");
       return;
   }

   initialize_data(append_data, BUFFERSIZE);

   // Write to the file
```

```
    if (fwrite(append_data, sizeof(char), BUFFERSIZE, file) != BUFFERSIZE) {
        printf("Error writing to file.\n");
    }

    // Move the file position indicator to the beginning of the file
    if (fseek(file, 0L, SEEK_SET) != 0) {
        printf("Error seeking in file.\n");
    }

    // Read from the same file
    if (fread(data, sizeof(char), BUFFERSIZE, file) < BUFFERSIZE) {
        printf("Error reading from file or no data available.\n");
    } else {
        printf("Data read from file: %.*s\n", BUFFERSIZE, data);
    }

    fclose(file);
}

int main(void) {
    func("example.txt");
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io9.c -o io9

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io9
Data read from file: ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEF
```

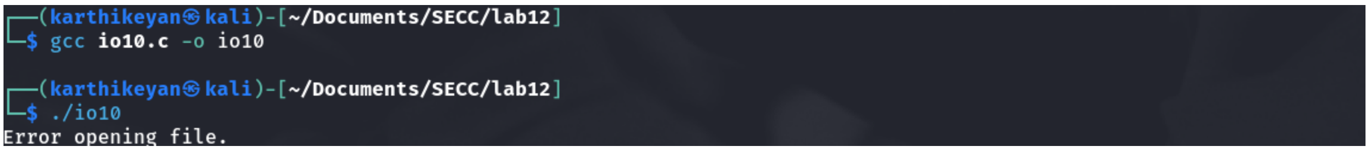
FIO40-C. Reset strings on fgets() or fgetws() failure

When using the C Standard library functions fgets() or fgetws(), it's essential to handle failures properly. If these functions fail, the contents of the buffer may be indeterminate, leading to undefined behavior. This assignment demonstrates how to reset strings when these functions fail.

Noncompliant Code Example

In this noncompliant code example, an error flag is set if fgets() fails. However, buf is not reset and has indeterminate contents:

```
#include <stdio.h>
#include <string.h>
enum { BUFFER_SIZE = 1024 };

void func(FILE *file) {
   char buf[BUFFER_SIZE];
   // Simulate reading from a file
   if (fgets(buf, sizeof(buf), file) == NULL) {
      // Set error flag and continue without resetting buf
      printf("Error reading from file.\n");
   }
   // Print the buffer contents, which may contain garbage values
   printf("Buffer contents (non-compliant): '%s'\n", buf);
}

int main(void) {
   FILE *file = fopen("non_existent_file.txt", "r"); // Attempt to open a non-existent file
   if (file == NULL) {
      printf("Error opening file.\n");
      return 1;
   }
   func(file);
   fclose(file);
   return 0;
}
```

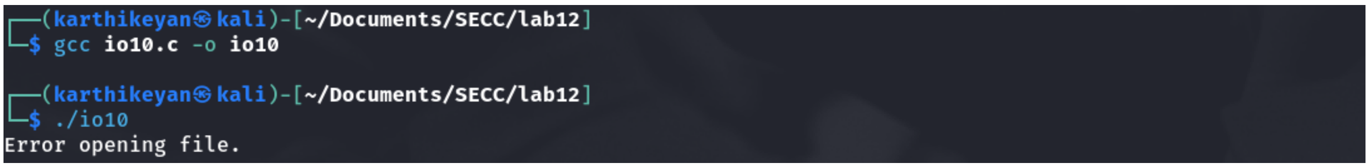


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io10.c -o io10

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io10
Error opening file.
```

Compliant Solution

In this compliant solution, buf is set to an empty string if fgets() fails. The equivalent solution for fgetws() would set buf to an empty wide string.

```
#include <stdio.h>

enum { BUFFER_SIZE = 1024 };


void func(FILE *file) {

   char buf[BUFFER_SIZE];

   // Attempt to read from the file

   if (fgets(buf, sizeof(buf), file) == NULL) {
       // Set error flag and reset buf
       printf("Error reading from file.\n");
       buf[0] = '\0'; // Reset buffer to an empty string
   }
   // Safe usage of buf
   printf("Buffer contents (compliant): '%s'\n", buf);
}

int main(void) {
   FILE *file = fopen("non_existent_file.txt", "r"); // Attempt to open a non-existent file
   if (file == NULL) {
       printf("Error opening file.\n");
       return 1;
   }
   func(file);
   fclose(file);
   return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io10.c -o io10

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io10
Error opening file.
```

FIO41-C. Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects

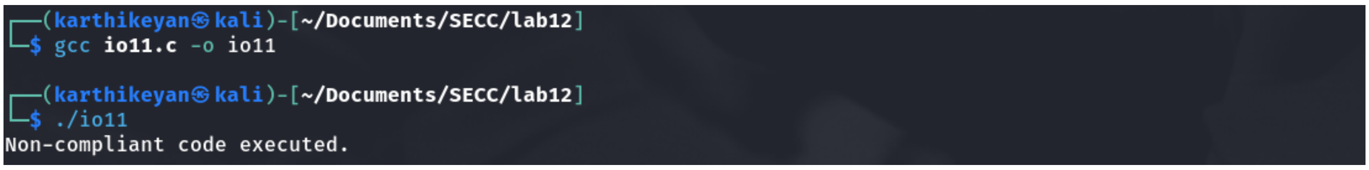
Do not invoke getc() or putc() or their wide-character analogues getwc() and putwc() with a stream argument that has side effects. The stream argument passed to these macros may be evaluated more than once if these functions are implemented as unsafe macros.

This rule does not apply to the character argument in putc() or the wide-character argument in putwc(), which is guaranteed to be evaluated exactly once.

Noncompliant Code Example (getc())

This noncompliant code example calls the getc() function with an expression as the stream argument. If getc() is implemented as a macro, the file may be opened multiple times.

```
#include <stdio.h>

void func_non_compliant(const char *file_name) {
    FILE *fptr;
    int c = getc(fptr = fopen(file_name, "r")); // Potential multiple calls to fopen
    if (fptr == NULL) {
        printf("Error: Unable to open the file.\n");
        return;
    }
    if (feof(fptr) || ferror(fptr)) {
        printf("Non-compliant Error: End of file reached or a read error occurred.\n");
    }
    if (fclose(fptr) == EOF) {
        printf("Error: Unable to close the file.\n");
    }
    printf("Non-compliant code executed.\n");
}

int main() {
    func_non_compliant("example.txt");
    return 0;
}
```



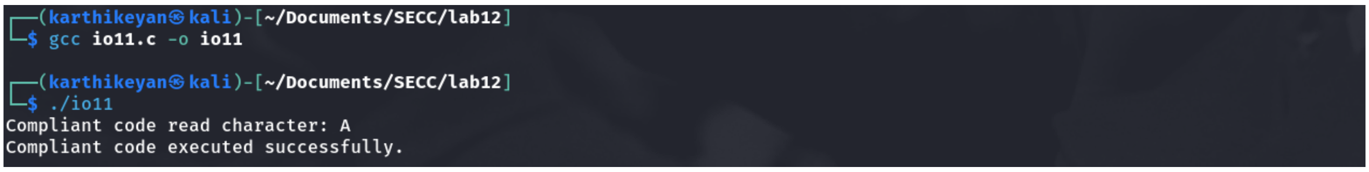
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io11.c -o io11

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io11
Non-compliant code executed.
```

This noncompliant code example also violates ERR33-C. Detect and handle standard library errors because the value returned by fopen() is not checked for errors.

Compliant Solution (getc())

In this compliant solution, fopen() is called before getc() and its return value is checked for errors:

```
#include <stdio.h>

void func_compliant(const char *file_name) {
    int c;
    FILE *fptr = fopen(file_name, "r"); // Open the file before using it
    if (fptr == NULL) {
        printf("Error: Unable to open the file.\n");
        return;
    }
    c = getc(fptr);
    if (c == EOF) {
        printf("Compliant Error: End of file reached or a read error occurred.\n");
    } else {
        printf("Compliant code read character: %c\n", c);
    }
    if (fclose(fptr) == EOF) {
        printf("Error: Unable to close the file.\n");
    }
    printf("Compliant code executed successfully.\n");
}

int main() {
    func_compliant("example.txt");
    return 0;
}
```

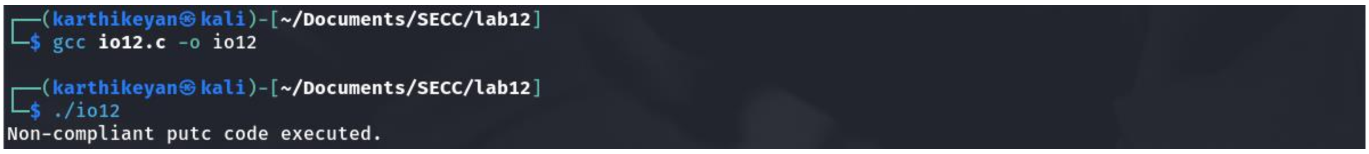


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io11.c -o io11

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io11
Compliant code read character: A
Compliant code executed successfully.
```

Noncompliant Code Example (putc())

In this noncompliant example, putc() is called with an expression as the stream argument. If putc() is implemented as a macro, this expression might be evaluated multiple times.

```
#include <stdio.h>

void func_putc_non_compliant(const char *file_name) {
    FILE *fptr = NULL;
    int c = 'a';
    while (c <= 'z') {
        if (putc(c++, fptr ? fptr : (fptr = fopen(file_name, "w"))) == EOF) {
            printf("Non-compliant Error: Unable to write to the file.\n");
        }
    }
    if (fptr != NULL && fclose(fptr) == EOF) {
        printf("Error: Unable to close the file.\n");
    }
    printf("Non-compliant putc code executed.\n");
}

int main() {
    func_putc_non_compliant("output.txt");
    return 0;
}
```



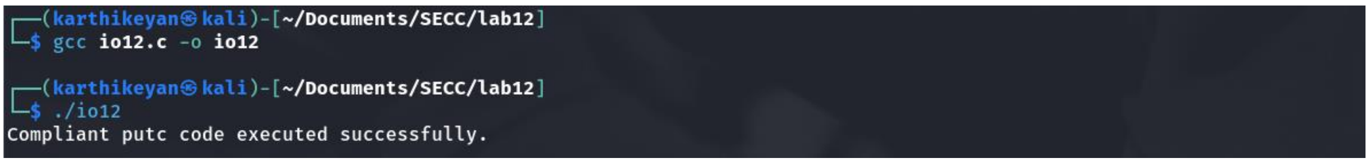
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io12.c -o io12

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io12
Non-compliant putc code executed.
```

This noncompliant code example might appear safe even if the putc() macro evaluates its stream argument multiple times, as the ternary conditional expression ostensibly prevents multiple calls to fopen(). However, the assignment to fptr and the evaluation of fptr as the controlling expression of the ternary conditional expression can take place between the same sequence points, resulting in undefined behavior. This code also violates ERR33-C. Detect and handle standard library errors because it fails to check the return value from fopen ().

Compliant Solution (putc())

In this compliant solution, the stream argument to putc() no longer has side effects:

```
#include <stdio.h>

void func_putc_compliant(const char *file_name) {
   int c = 'a';
   FILE *fptr = fopen(file_name, "w"); // Open the file before using it
   if (fptr == NULL) {
      printf("Error: Unable to open the file.\n");
      return;
   }
   while (c <= 'z') {
      if (putc(c++, fptr) == EOF) {
         printf("Compliant Error: Unable to write to the file.\n");
      }
   }
   if (fclose(fptr) == EOF) {
      printf("Error: Unable to close the file.\n");
   }
   printf("Compliant putc code executed successfully.\n");
}

int main() {
   func_putc_compliant("output.txt");
   return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io12.c -o io12

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io12
Compliant putc code executed successfully.
```

The expression c++ is perfectly safe because putc() guarantees to evaluate its character argument exactly once.

FIO42-C. Close files when they are no longer needed

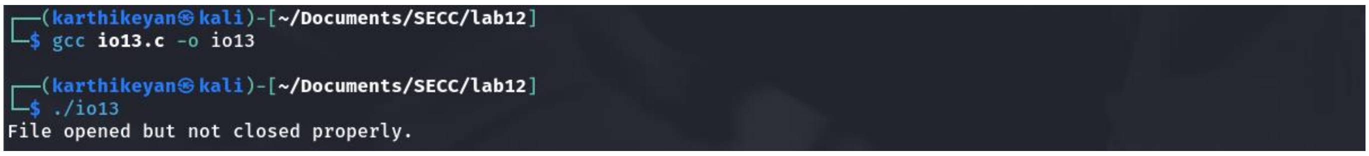
A call to the fopen() or freopen() function must be matched with a call to fclose() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first.

In general, this rule should also be applied to other functions with open and close resources, such as the POSIX open() and close() functions, or the Microsoft Windows CreateFile() and CloseHandle() functions.

Noncompliant Code Example (fopen())

This code example is noncompliant because the file opened by the call to fopen() is not closed before function func() returns:

```
#include <stdio.h>
#include <stdlib.h>

int func(const char *filename) {
    FILE *f = fopen(filename, "r");
    if (NULL == f) {
        return -1; // Error opening file
    }
    // Simulate some operations (e.g., reading)
    // Not closing the file
    return 0;
}

int main(void) {
    const char *filename = "example.txt";
    if (func(filename) == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
    printf("File opened but not closed properly.\n");
    exit(EXIT_SUCCESS);
}
```



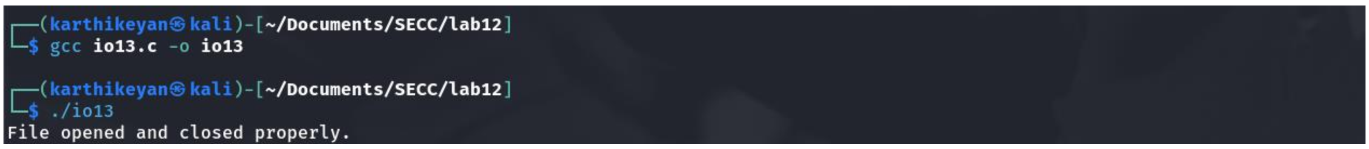
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io13.c -o io13

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io13
File opened but not closed properly.
```

Compliant Solution (fopen())

In this compliant solution, the file pointed to by f is closed before returning to the caller:

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

```
#include <stdlib.h>

int func(const char *filename) {
    FILE *f = fopen(filename, "r");
    if (NULL == f) {
        return -1; // Error opening file
    }
    // Simulate some operations (e.g., reading)
    if (fclose(f) == EOF) {
        return -1; // Error closing file
    }
    return 0;
}

int main(void) {
    const char *filename = "example.txt";
    if (func(filename) == -1) {
        perror("Error opening or closing file");
        exit(EXIT_FAILURE);
    }
    printf("File opened and closed properly.\n");
    exit(EXIT_SUCCESS);
}
```

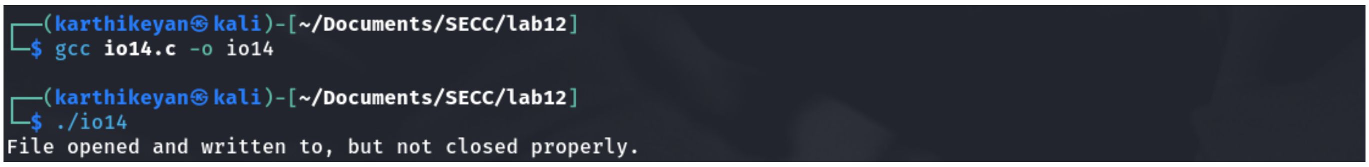


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io13.c -o io13

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io13
File opened and closed properly.
```

Noncompliant Code Example (exit())

This code example is noncompliant because the resource allocated by the call to fopen() is not closed before the program terminates. Although exit() closes the file, the program has no way of determining if an error occurs while flushing or closing the file.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const char *filename = "example.txt";
    FILE *f = fopen(filename, "w");
    if (NULL == f) {
        exit(EXIT_FAILURE); // Error opening file
    }
    // Simulate writing to the file
    fprintf(f, "Writing to the file without closing properly.\n");
    // Exiting without closing the file
    printf("File opened and written to, but not closed properly.\n");
    exit(EXIT_SUCCESS);
}
```

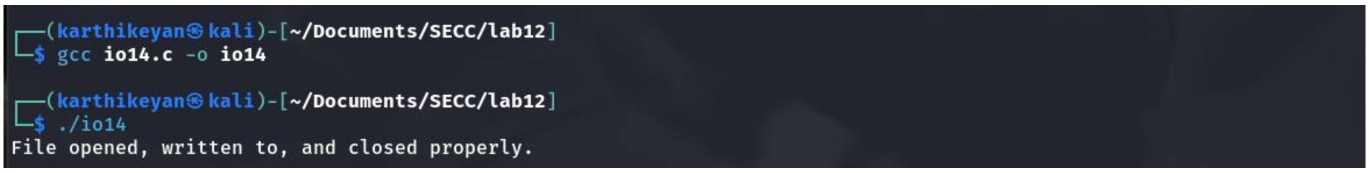


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io14.c -o io14

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io14
File opened and written to, but not closed properly.
```

Compliant Solution (exit())

In this compliant solution, the program closes f explicitly before calling exit(), allowing any error that occurs when flushing or closing the file to be handled appropriately:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const char *filename = "example.txt";
    FILE *f = fopen(filename, "w");
    if (NULL == f) {
        perror("Failed to open file");
        exit(EXIT_FAILURE); // Handle error
    }
    // Simulate writing to the file
    fprintf(f, "Writing to the file and closing properly.\n");
    // Properly closing the file
    if (fclose(f) == EOF) {
        perror("Error closing file");
        exit(EXIT_FAILURE); // Handle error
    }
    printf("File opened, written to, and closed properly.\n");
    exit(EXIT_SUCCESS);
}
```



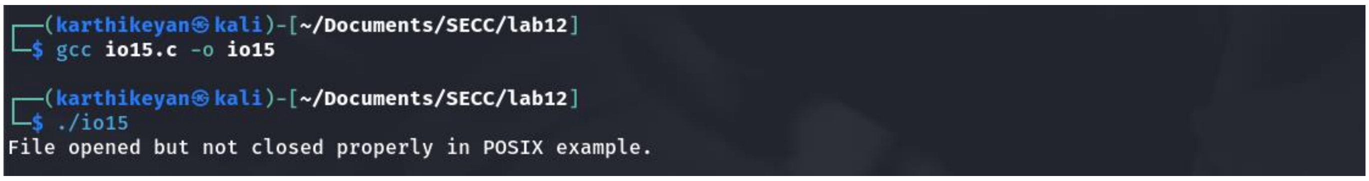
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io14.c -o io14

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io14
File opened, written to, and closed properly.
```

Noncompliant Code Example (POSIX)

This code example is noncompliant because the resource allocated by the call to open() is not closed before function func() returns:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <stdlib.h>

int func(const char *filename) {
    int fd = open(filename, O_RDONLY);
    if (-1 == fd) {
        return -1; // Error opening file
    }
    // File descriptor not closed
    return 0;
}

int main(void) {
    const char *filename = "example.txt";
    if (func(filename) == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
    printf("File opened but not closed properly in POSIX example.\n");
    exit(EXIT_SUCCESS);
}
```



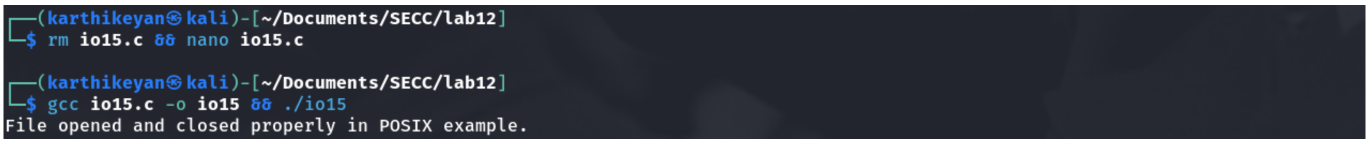
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io15.c -o io15

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ ./io15
File opened but not closed properly in POSIX example.
```

Compliant Code - Using POSIX open()

This compliant solution ensures that the file descriptor is closed before returning from the function, preventing resource leaks and maintaining system integrity.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int func(const char *filename) {
    int fd = open(filename, O_RDONLY);
    if (-1 == fd) {
        return -1; // Error opening file
    }
    // Properly closing the file descriptor
    if (-1 == close(fd)) {
        return -1; // Handle error
    }
    return 0;
}
```

```
int main(void) {
    const char *filename = "example.txt";
    if (func(filename) == -1) {
        perror("Error opening or closing file");
        exit(EXIT_FAILURE);
    }
    printf("File opened and closed properly in POSIX example.\n");
    exit(EXIT_SUCCESS);
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ rm io15.c && nano io15.c

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io15.c -o io15 && ./io15
File opened and closed properly in POSIX example.
```

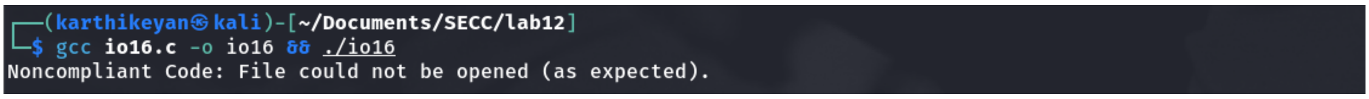
FIO44-C. Only use values for fsetpos() that are returned from fgetpos()

The fsetpos function sets the mbstate_t object (if any) and file position indicator for the stream pointed to by stream according to the value of the object pointed to by pos, which shall be a value obtained from an earlier successful call to the fgetpos function on a stream associated with the same file.

Noncompliant Code Example

This noncompliant code example attempts to read three values from a file and then set the file position pointer back to the beginning of the file:

```
#include <stdio.h>
#include <string.h>

int opener(FILE *file) {
    int rc;
    fpos_t offset;
    memset(&offset, 0, sizeof(offset)); // Incorrectly initializing offset

    if (file == NULL) {
        printf("Noncompliant Code: File is NULL.\n");
        return -1; // Error handling
    }

    // Simulated file read operation
    // Normally you would read data here, but for this example, it's omitted

    // Attempt to set position using an invalid offset
    rc = fsetpos(file, &offset);
    if (rc != 0) {
        printf("Noncompliant Code: Error occurred during file position setting (expected failure).\n");
        return rc; // Return error code if fsetpos fails
    }
    return 0; // Success
}

int main() {
    FILE *file = fopen("nonexistent.txt", "r"); // Trying to open a nonexistent file
    if (file == NULL) {
        printf("Noncompliant Code: File could not be opened (as expected).\n");
        return -1;
    }

    int result = opener(file); // Call to noncompliant function
    fclose(file);

    if (result == 0) {
        printf("Noncompliant Code: File position reset (unexpected success).\n");
    }

    return 0;
}
```



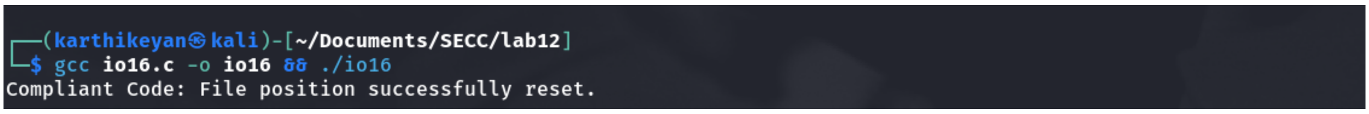
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io16.c -o io16 && ./io16
Noncompliant Code: File could not be opened (as expected).
```

Only the return value of an fgetpos() call is a valid argument to fsetpos(); passing a value of type fpos_t that was created in any other way is undefined behavior.

Compliant Solution

In this compliant solution, the initial file position indicator is stored by first calling fgetpos(), which is used to restore the state to the beginning of the file in the later call to fsetpos():

```
#include <stdio.h>
#include <string.h>

int opener(FILE *file) {
    int rc;
    fpos_t offset;

    if (file == NULL) {
        return -1; // Error handling
    }

    rc = fgetpos(file, &offset); // Correctly obtaining the current position
    if (rc != 0) {
        return rc; // Return error code if fgetpos fails
    }

    // Simulated file read operation
    // Normally you would read data here, but for this example, it's omitted

    rc = fsetpos(file, &offset); // Using valid offset
    if (rc != 0) {
        return rc; // Return error code if fsetpos fails
    }

    return 0; // Success
}

int main() {
    FILE *file = fopen("example.txt", "r"); // Opening a sample file
    if (file == NULL) {
        return -1;
    }

    int result = opener(file); // Call to compliant function
    fclose(file);

    if (result == 0) {
        printf("Compliant Code: File position successfully reset.\n");
    } else {
        printf("Compliant Code: Error occurred during file position setting.\n");
    }

    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io16.c -o io16 && ./io16
Compliant Code: File position successfully reset.
```

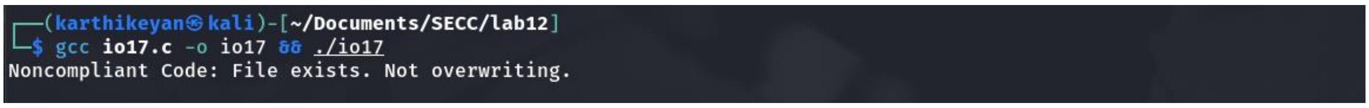
FIO45-C. Avoid TOCTOU race conditions while accessing files

- *TOCTOU (Time-of-Check to Time-of-Use) race conditions occur when two processes concurrently access and operate on the same file.*
- *Typically, an initial check verifies file attributes, followed by file use; an attacker can exploit the time between these actions.*
- *During this gap, an attacker could alter, remove, or replace the file (e.g., with a symbolic or hard link to another file), creating a race window.*
- *This vulnerability arises from assuming the file remains unchanged between operations.*
- *Such race conditions are particularly risky for programs that repeatedly access the same file name or path name, as the file's identity or state might be manipulated.*

Noncompliant Code Example

If an existing file is opened for writing with the w mode argument, the file's previous contents (if any) are destroyed. This noncompliant code example tries to prevent an existing file from being overwritten by first opening it for reading before opening it for writing. An attacker can exploit the race window between the two calls to fopen() to overwrite an existing file.

```
#include <stdio.h>
void open_some_file(const char *file) {
   FILE *f = fopen(file, "r"); // Check if the file exists
   if (f != NULL) {
      printf("Noncompliant Code: File exists. Not overwriting.\n");
      fclose(f); // Close the file if it exists
   } else {
      f = fopen(file, "w"); // Attempt to open for writing
      if (f == NULL) {
         printf("Noncompliant Code: Failed to open file for writing.\n");
      } else {
         printf("Noncompliant Code: Writing to file.\n");
         // Write to file (simulated)
         if (fclose(f) == EOF) {
            printf("Noncompliant Code: Error closing file.\n");
         }
      }
   }
}

int main() {
   open_some_file("example.txt"); // Attempt to open a file
   return 0;
}
```

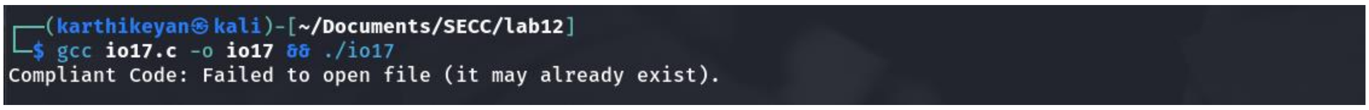


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io17.c -o io17 && ./io17
Noncompliant Code: File exists. Not overwriting.
```

Compliant Solution

This compliant solution invokes fopen() at a single location and uses the x mode of fopen(), which was added in C11. This mode causes fopen() to fail if the file exists. This check and subsequent open is performed without creating a race window. The x mode provides exclusive access to the file only if the host environment provides this support.

```
#include <stdio.h>
void open_some_file(const char *file) {
   FILE *f = fopen(file, "wx"); // Open file for writing exclusively
   if (f == NULL) {
        printf("Compliant Code: Failed to open file (it may already exist).\n");
    } else {
        printf("Compliant Code: Writing to file.\n");
        // Write to file (simulated)
        if (fclose(f) == EOF) {
            printf("Compliant Code: Error closing file.\n");
        }
    }
}

int main() {
    open_some_file("example.txt"); // Attempt to open a file
    return 0;
}
```

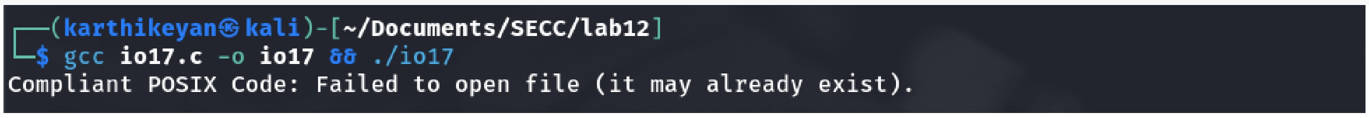


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io17.c -o io17 && ./io17
Compliant Code: Failed to open file (it may already exist).
```

Compliant Solution (POSIX)

This compliant solution uses the O_CREAT and O_EXCL flags of POSIX's open() function. These flags cause open() to fail if the file exists.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
void open_some_file(const char *file) {
    int fd = open(file, O_CREAT | O_EXCL | O_WRONLY, 0644); // Exclusive write mode
    if (fd == -1) {
        printf("Compliant POSIX Code: Failed to open file (it may already exist).\n");
    } else {
        FILE *f = fdopen(fd, "w");
        if (f != NULL) {
            printf("Compliant POSIX Code: Writing to file.\n");
            // Write to file (simulated)
            if (fclose(f) == EOF) {
                printf("Compliant POSIX Code: Error closing file.\n");
            }
        } else {
            if (close(fd) == -1) {
                printf("Compliant POSIX Code: Error closing file descriptor.\n");
            }
        }
    }
}

int main() {
    open_some_file("example.txt"); // Attempt to open a file
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io17.c -o io17 && ./io17
Compliant POSIX Code: Failed to open file (it may already exist).
```

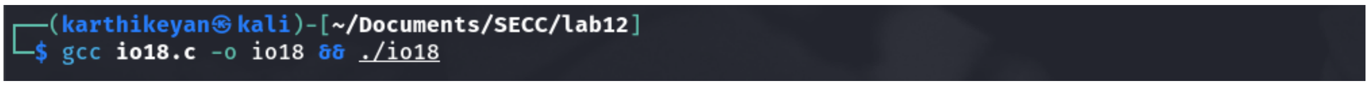
FIO46-C. Do not access a closed file

Accessing a closed file in C leads to undefined behavior. This assignment demonstrates the risks associated with using pointers to FILE objects after they have been closed and presents compliant solutions that prevent such scenarios.

Noncompliant Code Example

In this noncompliant code example, the stdout stream is used after it is closed:

```
#include <stdio.h>
int close_stdout(void) {
    if (fclose(stdout) == EOF) {
        return -1; // Handle error if closing stdout fails
    }
    printf("stdout successfully closed.\n"); // Undefined behavior
    return 0;
}

int main() {
    close_stdout(); // Attempt to close stdout
    return 0;
}
```

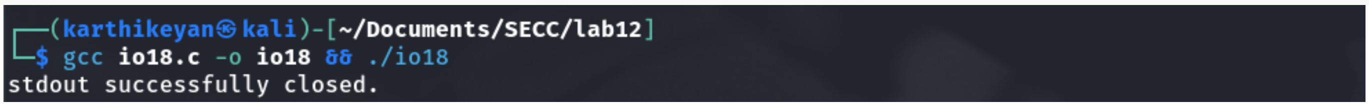


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io18.c -o io18 && ./io18
```

Compliant Solution

In this compliant solution, stdout is not used again after it is closed. This must remain true for the remainder of the program, or stdout must be assigned the address of an open file object.

```
#include <stdio.h>
int close_stdout(void) {
    if (fclose(stdout) == EOF) {
        return -1; // Handle error if closing stdout fails
    }
    fputs("stdout successfully closed.\n", stderr); // Safe output to stderr
    return 0;
}

int main() {
    close_stdout(); // Attempt to close stdout
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io18.c -o io18 && ./io18
stdout successfully closed.
```

FIO47-C. Use valid format strings

Formatted output functions (like fprintf()) print arguments based on format strings containing:

- *Ordinary characters copied directly to output*
- *Conversion specifications for formatting arguments*

Format String Structure:

Each conversion specification starts with % and includes:

- ***Flags**: Modify conversion (e.g., -, +, space, #, 0)*
- ***Minimum Field Width**: Optional, controls output width*
- ***Precision**: Sets digit count, byte limit, etc., based on specifier*
- ***Length Modifier**: Adjusts argument size (e.g., h, l, L)*
- ***Conversion Specifier**: Determines argument type conversion (e.g., d, f, s)*

Common Format String Mistakes:

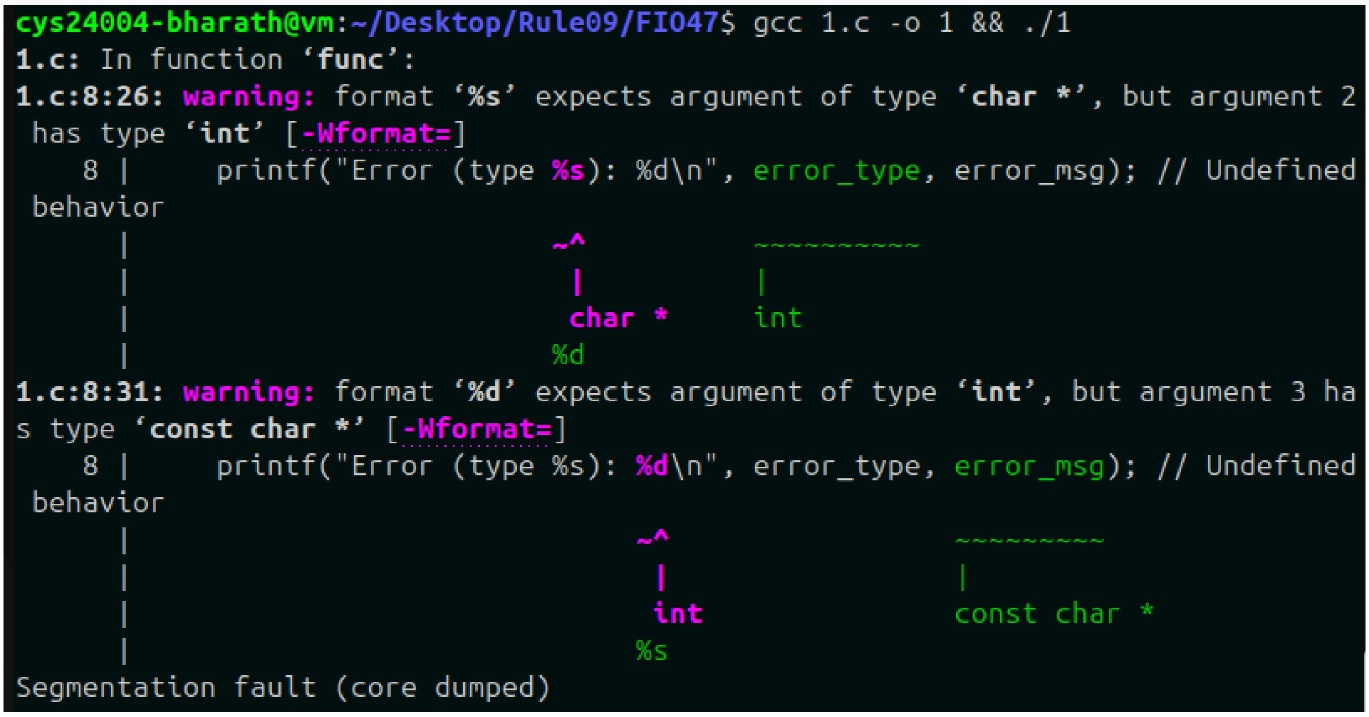
- *Incorrect number of arguments*
- *Invalid conversion specifiers*
- *Incompatible flag or length modifier*
- *Mismatched argument type and conversion specifier*
- *Non-int type used for width or precision*

Noncompliant Code Example

Mismatches between arguments and conversion specifications may result in undefined behavior. Compilers may diagnose type mismatches in formatted output function invocations. In this noncompliant code example, the error_type argument to printf() is incorrectly matched with the s specifier rather than with the d specifier. Likewise, the error_msg argument is incorrectly matched with the d specifier instead of the s specifier. These usages result in undefined behavior. One possible result of this invocation is that printf() will interpret the error_type argument as a pointer and try to read a string from the address that error_type contains, possibly resulting in an access violation.

```
#include <stdio.h>
void func(void) {
    const char *error_msg = "Resource not available to user.";
    int error_type = 3;

    /* Incorrect format specifiers */
    printf("Error (type %s): %d\n", error_type, error_msg); // Undefined behavior
}
```

```
int main() {
   func(); // Call the function
   return 0;
}
```

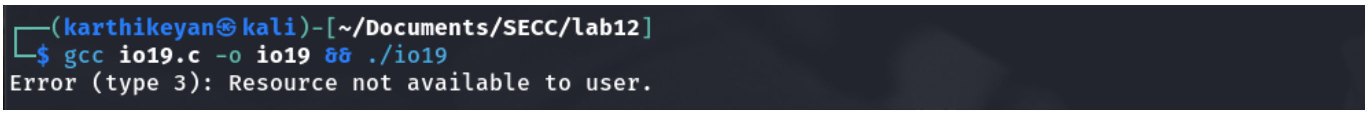


```
cys24004-bharath@vm:~/Desktop/Rule09/FI047$ gcc 1.c -o 1 && ./1
1.c: In function ‘func’:
1.c:8:26: warning: format ‘%s’ expects argument of type ‘char *’, but argument 2
 has type ‘int’ [-Wformat=]
    8 |     printf("Error (type %s): %d\n", error_type, error_msg); // Undefined
 behavior
      |                         ~^          ~~~~~~~~~~
      |                          |          |
      |                          char *     int
      |                         %d
1.c:8:31: warning: format ‘%d’ expects argument of type ‘int’, but argument 3 ha
s type ‘const char *’ [-Wformat=]
    8 |     printf("Error (type %s): %d\n", error_type, error_msg); // Undefined
 behavior
      |                              ~^                 ~~~~~~~~~
      |                               |                 |
      |                               int               const char *
      |                              %s
Segmentation fault (core dumped)
```

Compliant Solution

This compliant solution ensures that the arguments to the printf() function match their respective conversion specifications:

```
#include <stdio.h>

void func(void) {
   const char *error_msg = "Resource not available to user.";
   int error_type = 3;

   /* Correct format specifiers */
   printf("Error (type %d): %s\n", error_type, error_msg); // Safe and defined behavior
}

int main() {
   func(); // Call the function
   return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab12]
└─$ gcc io19.c -o io19 && ./io19
Error (type 3): Resource not available to user.
```