SAPIENZA
UNIVERSITÀ DI ROMA

SAPIENZA UNIVERSITY OF ROME

MASTER'S DEGREE IN CYBERSECURITY

DEPARTMENT OF COMPUTER SCIENCE

Project 1

Static analysis of a C code fragment using *flawfinder*

Author:
Marco Ruvolo

Matricola number:
1883257

Professor:
Daniele Friolo

Course:
Security in Software Applications

November 9, 2022

Contents

List of Figures

1 Introduction to static analysis

1.1 Static analysis

Static analysis is a set of techniques for verifying software without executing it (e.g., by examining the source code or the object code of a program). Actually, there are different static analysis approaches and tools such as human analysis, type checkers, text scanners and many others.

Therefore, one of these static analysis tools will be described in the following subsection.

1.2 A static analysis tool: *flawfinder*

Flawfinder[1] is a tool used to examine C/C++ source code and to report possible security vulnerabilities in the program.

On the one hand, this tool has various strenghts:

- it works by using a built-in database of C/C++ functions with well-known problems (e.g., buffer overflows, . . .) so you do not need to create this database by yourself;
- it carries out a list of potential security flaws (i.e., hits) sorted by risk and it may be able, in some cases, to determine that some construct are not dangerous at all, thus reducing false positives;
- it can analyse software that you cannot build and it can even analyse files you cannot compile, in some cases;
- it gives finer information than simply running 'grep'[2] Linux command on the source code (e.g., it knows to ignore comments and the insides of strings, it knows to examine parameters to estimate risk levels, . . .).

On the other hand, *flawfinder* has some weaknesses too:

- it does not check on the data types of function arguments and it does not do control flow or data flow analysis at all (however, there exist other tools to analyse software more deeply);
- it does not understand the semantics of the code at all: it mainly does (simple) text pattern matching;
- it may report false positives (i.e., hits that are not actual security vulnerabilities);
- it may not necessarily find every security vulnerability in the program.

Summing up, *flawfinder* is a simple tool that can be helpful in finding (and then removing) security vulnerabilities in C/C++ programs. However, source code should nevertheless be inspected and evaluated to find possible vulnerabilities not reported by this tool.

1.3 Project repository

The project track carried out, the static analysis tool used, the different code fragment versions analysed, the images within this report and this latter itself, are collected in an online GitHub[3] repository.

For further information please visit the following link: https://github.com/mrcruv/flawfinder_static_analysis.

2 A code fragment analysis

2.1 Syntax error and warning fixing

First of all (although *flawfinder* can analyse files that cannot even be compiled), inspecting the original code fragment (see subsection 5.1), trying to compile it and fixing possible typos/syntax errors, is good practice before (actually, after as well) analysing software:

warnings at lines 3, 4, 5, 6: extra tokens at end of #include directive $\xrightarrow{fix}$ the extra ';' tokens are removed at the end of each line;

warning at line 26 implicit declaration of function 'read'; did you mean 'fread'? $\xrightarrow{fix}$ the missing #include directive for 'unistd.h'[4] is added;

error at line 27 'buf' undeclared (first use in this function); did you mean 'buf2'? $\xrightarrow{fix}$ the suggested correction is applied;

warning at line 39: implicit declaration of function 'error'; did you mean 'perror'? $\xrightarrow{fix}$ 'error' is reasonably replaced by 'fprintf'[5] rather than by 'perror'[6], since 'errno'[7] is not set;

error at line 54: expected '{' at end of input $\xrightarrow{fix}$ the missing '{' at the end of line 47 is added;

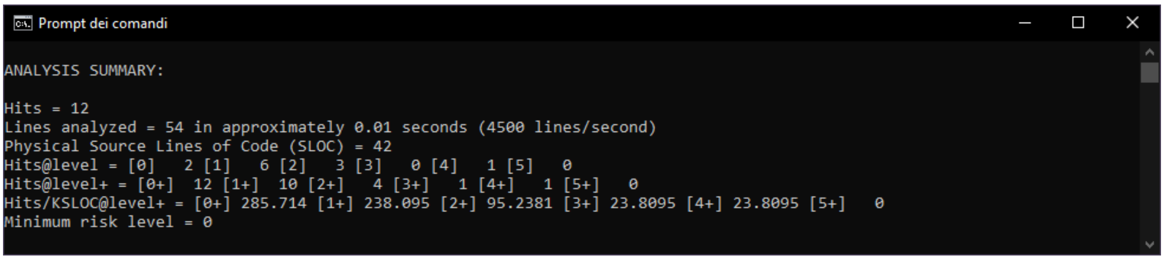
warnings at line 55: implicit declaration of function 'len' [-Wimplicit-function-declaration] $\xrightarrow{fix}$ 'len' is reasonably replaced by 'strlen'[8]; passing argument 1 of 'strlen' makes pointer from integer without a cast $\xrightarrow{fix}$ 'foo' is passed as the first argument of 'strlen' instead of '*foo'.

Please note that, with respect to the request to run *flawfinder* before fixing typos, these fixes do not have a significant impact on the analysis performed.

2.2 Security warning analysis

Now, running *flawfinder* on the "preprocessed" code fragment (see subsection 5.2) using the flag '-m 0' or '–minlevel=0', to set the minimum risk level to 0 (i.e., the minimum risk level can go from 0 - "no risk" - to 5 - "maximum risk" - and the default is 1) for inclusion in the hitlist (i.e., final results), carries out the analysis results.

The analysis summary shows that there are some possible security vulnerabilities in the code fragment, as can be seen from figure 1.



```
ANALYSIS SUMMARY:

Hits = 12
Lines analyzed = 54 in approximately 0.01 seconds (4500 lines/second)
Physical Source Lines of Code (SLOC) = 42
Hits@level = [0]   2 [1]   6 [2]   3 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]  12 [1+]  10 [2+]   4 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 285.714 [1+] 238.095 [2+] 95.2381 [3+] 23.8095 [4+] 23.8095 [5+]   0
Minimum risk level = 0
```

Figure 1: summary of the analysis of the code fragment (subsection 5.2)

In particular, the hits shown in the final results are twelve (mainly related to strings and buffer overflows), as can be seen from figure 2:

1) hit at line 52 (CWE-120[9]): since the destination string 'buffer' is not large enough to receive the copy of the source string 'foo' the 'strcpy'[10] invocation at line 52 will cause a buffer overflow, hence this hit is a vulnerability;

2) hit at line 10 (CWE-119![11]/CWE-120): since at line 13 'fgets'[12] is used to initialize 'buffer' and the second argument is set to the size in characters of 'buffer', 'buffer' will contain at most 1023 characters from 'stdin'[13] input channel (moreover, its 1024-th character will be the null byte), hence this hit is not a vulnerability (namely, 'buffer' is not improperly restricted);

3) hit at line 16 (CWE-119!/CWE-120): since the destination string 'errormsg' is large enough to either receive the copy of the source string 'buffer' ('strncpy'[14] at line 18) and to have the source string 'buffer' concatenated to itself ('strcat'[15] at line 19), and 'buffer' is null-terminated (by 'fgets' at line 13), this hit is not a vulnerability (namely, 'errormsg' is not improperly restricted) - moreover, 'errormsg' will be even null-terminated by 'strcat';

(4) hit at line 19 (CWE-120): since 'errormsg' is large enough to store the result carried out by the invocation of 'strcat' at line 19, this hit is not a vulnerability;

(5) hit at line 18 (CWE-120): since 'buffer' is null-terminated, after the 'strncpy' invocation at line 18, 'errormsg' will be null-terminated too, hence this hit is not a vulnerability;

6) hit at line 27 (CWE-120, CWE-20[16]): since the third argument of 'read'[17] at line 27 is set to the size in bytes of 'len', this hit is not a vulnerability;

7) hit at line 29 (CWE-120, CWE-20): since the third argument of 'read' at line 29 is set to the size (minus one) in characters (here, equal to the size in bytes minus one) of 'buf2', this hit is not a vulnerability;

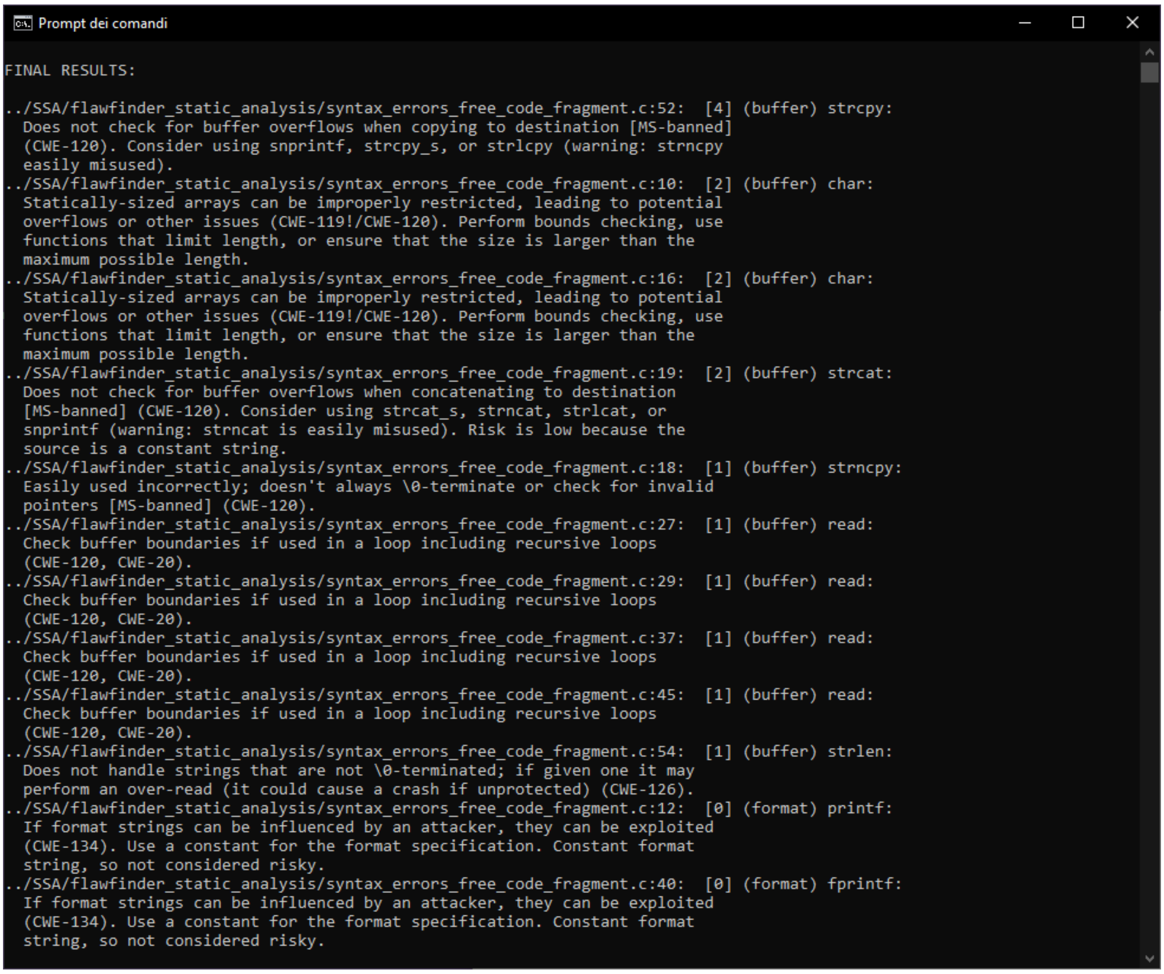
8) hit at line 37 (CWE-120, CWE-20): since the third argument of 'read' at line 37 is set to the size in bytes of 'buf3', this hit is not a vulnerability;

9) hit at line 45 (CWE-120, CWE-20): since the third argument of 'read' at line 29 is set to the size in characters (here, equal to the size in bytes) of 'buf3', (yet 'buf3' will not be null-terminated in general: contrariwise, 'buf2' in 'func2' is explictly null-terminated at line 30), this hit is not a vulnerability (although all char buffers should be null-terminated);

10) hit at line 54 (CWE-126[18]): since 'foo' is null-terminated by definition (even though the last character assigned to it is not explicitly the null byte, 'foo' is automatically null-terminated after being declared at line 49), this hit is not a vulnerability (namely, 'strlen' will not perform an over read);

11) hit at line 12 (CWE-134[19]): since the format string passed as argument to 'printf'[20] invocation at line 12 is constant, this hit is not a vulnerability (namely, no attacker can influence this format string);

12) hit at line 40 (CWE-134): since the format string passed as argument to 'fprintf' invocation at line 40 is constant, this hit is not a vulnerability (namely, no attacker can influence this format string).



```
FINAL RESULTS:

../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:52:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination [MS-banned]
  (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy
  easily misused).
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:10:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:16:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:19:  [2] (buffer) strcat:
  Does not check for buffer overflows when concatenating to destination
  [MS-banned] (CWE-120). Consider using strcat_s, strncat, strlcat, or
  snprintf (warning: strncat is easily misused). Risk is low because the
  source is a constant string.
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:18:  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check for invalid
  pointers [MS-banned] (CWE-120).
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:27:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:29:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:37:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:45:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:54:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:12:  [0] (format) printf:
  If format strings can be influenced by an attacker, they can be exploited
  (CWE-134). Use a constant for the format specification. Constant format
  string, so not considered risky.
../SSA/flawfinder_static_analysis/syntax_errors_free_code_fragment.c:40:  [0] (format) fprintf:
  If format strings can be influenced by an attacker, they can be exploited
  (CWE-134). Use a constant for the format specification. Constant format
  string, so not considered risky.
```

Figure 2: final results of the analysis of the code fragment (subsection 5.2)

2.3 Other vulnerabilities and flaws

Inspecting code again might help find other security vulnerabilities or simply weaknesses.

For instance, the following should be considered:

return value checking: with respect to each function invocation in the code, return values (if present) should be checked to determine whether a function (e.g., 'printf', 'fgets', 'malloc'[21], 'read', ...) has been successfully executed or not;

exit in case of error: when an error is detected it should be handled (e.g., terminating the code execution or trying again);

unsafe string library usage: since some standard string library (i.e., 'string.h'[22]) functions (e.g., 'strcpy', ...) have been banned and have more robust or secure replacements (i.e., these functions are deprecated), a safe string library should be used instead (e.g., 'strsafe.h'[23]);

memory management: with respect to memory allocation, 'calloc'[24] might be used rather than 'malloc' (e.g., to zero out memory in advance), furthermore allocated memory should be eventually released using 'free'[25];

explicit type casting: when useful (e.g., 'calloc' return value, for code readability or disambiguation) or required (e.g., 'isalpha'[26] checks argument to have the value of an unsigned char or EOF) explicit type conversion should be performed.

3 Vulnerability fixing

At this point, a corrected version of the code fragment (see subsection 5.3) is produced by removing the security vulnerabilities found.

Here follows a summary of the vulnerability and flaw analysis performed in section 2:

1) vulnerability at line 52: 'buffer' size is not large enough to receive the copy of 'foo';

2) flaws at multiple lines: return values should be checked;

3) flaws at multiple lines: errors should be handled;

4) flaws at multiple lines: deprecated/unsafe string functions should be replaced by more robust or secure ones;

5) flaws at multiple lines: memory management should be done properly;

6) flaws at multiple lines: explicit type casting should be performed.

In practice, these latter are fixed in such a way:

1), 2), 4): 'strcpy' at line 52 is replaced by 'StringCbCopyA'[27] and a return value check is performed;

2): a return value check is performed on 'printf' at line 12;

2): a return value check is performed on 'fgets' at line 13;

2), 4): 'strncpy' at line 18 is replaced by 'StringCbCopyA' and a return value check is performed;

2), 4): 'strcat' at line 19 is replaced by 'StringCbCatA'[28] and a return value check is performed;

2): return value checks are performed on 'read' at lines 27, 29, 37, 45;

2), 5): 'malloc' at lines 28, 44, 50 are replaced by 'calloc', return value checks are performed and memory is eventually deallocated using 'free';

2): 'strlen' at line 54 (as argument of 'func3') is replaced by 'strnlen'[29];

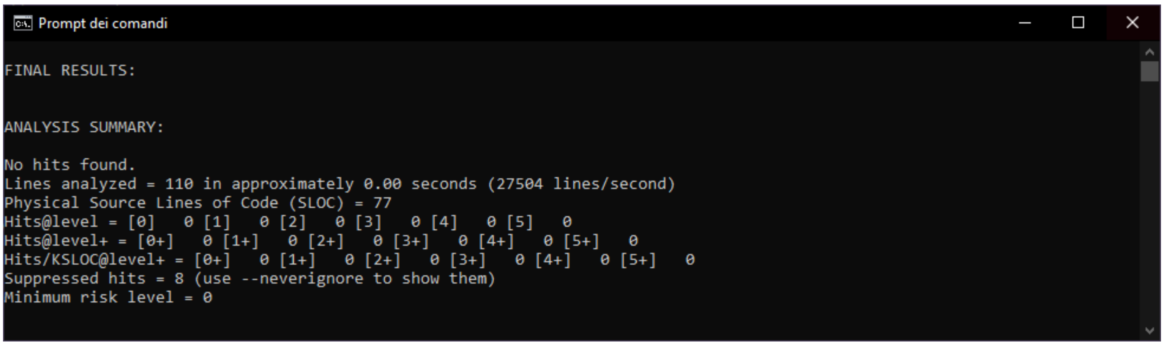
3): in case of error (at lines 12, 13, 18, 19, 27, 28, 29, 37, 40, 44, 45, 50, 52), 'exit'[30] is called ('EXIT_FAILURE' is passed as argument);

6): explicit type casting is performed ('isalpha' argument to 'unsigned char' at line 15 and 'calloc' return value to 'char *' at lines 28, 44).

Eventually, some minor corrections are made, for instance 'perror' is used to show messages on some standard errors (e.g., on 'read').

Please note that other message errors are not explicitly showed on 'stderr'[13] using 'fprintf', for the sake of brevity. Moreover, neither a return value check is performed nor possible errors are handled on 'fprintf' at line 40 (this invocation works, in itself, as an error handler).

Now, running *flawfinder* on the "fixed" code fragment using the flag '-m 0' or '–minlevel=0' (and ignoring the warnings that do not represent actual vulnerabilities by adding '// flawfinder: ignore' at the end of the involved lines in the code fragment) shows that all the vulnerabilities found have been removed, as can be seen from figure 3.



```
FINAL RESULTS:


ANALYSIS SUMMARY:

No hits found.
Lines analyzed = 110 in approximately 0.00 seconds (27504 lines/second)
Physical Source Lines of Code (SLOC) = 77
Hits@level = [0]   0 [1]   0 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   0 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+]   0 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Suppressed hits = 8 (use --neverignore to show them)
Minimum risk level = 0
```

Figure 3: final results and summary of the analysis of the fixed code fragment (subsection 5.3)

4 References

[1] *Flawfinder*. URL: https://dwheeler.com/flawfinder/.

[2] *grep(1) — Linux manual page*. URL: https://man7.org/linux/man-pages/man1/grep.1.html.

[3] *GitHub*. URL: https://github.com/.

[4] *unistd.h(0p) — Linux manual page*. URL: https://man7.org/linux/man-pages/man0/unistd.h.0p.html.

[5] *fprintf(3p) - Linux manual page*. URL: https://man7.org/linux/man-pages/man3/fprintf.3p.html.

[6] *perror(3) - Linux manual page*. URL: https://man7.org/linux/man-pages/man3/perror.3.html.

[7] *errno(3) - Linux manual page*. URL: https://man7.org/linux/man-pages/man3/errno.3.html.

[8] *strlen(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/strlen.3.html.

[9] *CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')*. URL: https://cwe.mitre.org/data/definitions/120.html.

[10] *strcpy(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/strcpy.3.html.

[11] *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer*. URL: https://cwe.mitre.org/data/definitions/119.html.

[12] *fgets(3p) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/fgets.3p.html.

[13] *stdin(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/stdin.3.html.

[14] *strncpy(3p) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/strncpy.3p.html.

[15] *strcat(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/strcat.3.html.

[16] *CWE-20: Improper Input Validation*. URL: https://cwe.mitre.org/data/definitions/20.html.

[17] *read(3p) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/read.3p.html.

[18] *CWE-126: Buffer Over-read*. URL: https://cwe.mitre.org/data/definitions/126.html.

[19] *CWE-134: Use of Externally-Controlled Format String*. URL: https://cwe.mitre.org/data/definitions/134.html.

[20] *printf(3) - Linux manual page*. URL: https://man7.org/linux/man-pages/man3/printf.3.html.

[21] *malloc(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/malloc.3.html.

[22] *string.h(0p) — Linux manual page*. URL: https://man7.org/linux/man-pages/man0/string.h.0p.html.

[23] *strsafe.h header*. URL: https://learn.microsoft.com/en-us/windows/win32/api/strsafe/.

[24] *calloc(3p) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/calloc.3p.html.

[25] *free(3p) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/free.3p.html.

[26] *isalpha(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/isspace.3.html.

[27] *StringCbCopyA function (strsafe.h)*. URL: https://learn.microsoft.com/en-us/windows/win32/api/strsafe/nf-strsafe-stringcbcopya.

[28] *StringCbCatA function (strsafe.h)*. URL: https://learn.microsoft.com/en-us/windows/win32/api/strsafe/nf-strsafe-stringcbcata.

[29] *strnlen(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/strnlen.3.html.

[30] *exit(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/exit.3.html.

5 Appendixes

5.1 Appendix A

```
// original code fragment:

#include <stdio.h>;
#include <ctype.h>;
#include <string.h>;
#include <stdlib.h>;

void func1() {
   char buffer[1024];

   printf("Please enter your user id :");
   fgets(buffer, 1024, stdin);

   if (!isalpha(buffer[0])) {
      char errormsg[1044];

      strncpy(errormsg, buffer, 1024);
      strcat(errormsg, " is not a valid ID");
   }
}

void func2(int f2d) {
   char *buf2;
   size_t len;

   read(f2d, &len, sizeof(len));
   buf = malloc(len + 1);
   read(f2d, buf2, len);
   buf2[len] = '\0';
}

void func3(int f3d){
   char *buf3;
   int i, len;

   read(f3d, &len, sizeof(len));

   if (len > 8000) {
      error("too large length");
      return;
   }

   buf3 = malloc(len);
   read(f3d, buf3, len);
}

void main()
   char *foo = "foooooooooooooooooooooooooooooooooooooooooooooooooooo";
   char *buffer = (char *)malloc(10 * sizeof(char));

   strcpy(buffer, foo);
   func1();
   func3(len(*foo));
}
```

5.2 Appendix B

```
// syntax errors (and warnings) free code fragment:

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

void func1() {
   char buffer[1024];

   printf("Please enter your user id :");
   fgets(buffer, 1024, stdin);

   if (!isalpha(buffer[0])) {
      char errormsg[1044];

      strncpy(errormsg, buffer, 1024);
      strcat(errormsg, " is not a valid ID");
  }
}

void func2(int f2d) {
   char *buf2;
   size_t len;

   read(f2d, &len, sizeof(len));
   buf2 = malloc(len + 1);
   read(f2d, buf2, len);
   buf2[len] = '\0';
}

void func3(int f3d){
   char *buf3;
   int i, len;

   read(f3d, &len, sizeof(len));

   if (len > 8000) {
      fprintf(stderr, "too large length");
      return;
   }

   buf3 = malloc(len);
   read(f3d, buf3, len);
}

void main() {
   char *foo = "foooooooooooooooooooooooooooooooooooooooooooooooooooo";
   char *buffer = (char *)malloc(10 * sizeof(char));

   strcpy(buffer, foo);
   func1();
   func3(strlen(foo));
}
```

5.3 Appendix C

```c
// fixed code fragment:

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <strsafe.h>

#define FUNC1_BUFFER_SIZE 1024
#define MAIN_BUFFER_SIZE 10

void func1() {
   char buffer[FUNC1_BUFFER_SIZE]; // flawfinder: ignore
   int ret;
   HRESULT hr;

   ret = printf("Please enter your user id :"); // flawfinder: ignore

   if (ret < 0) exit(EXIT_FAILURE); // actually the condition to be verified would be 'ret < strnlen("Please enter your user id :", 29)'

   if (fgets(buffer, FUNC1_BUFFER_SIZE, stdin) == NULL) exit(EXIT_FAILURE);

   if (!isalpha((unsigned char)buffer[0])) {
      char errormsg[1044]; // flawfinder: ignore

      hr = StringCbCopyA(errormsg, FUNC1_BUFFER_SIZE, buffer);

      if (FAILED(hr)) exit(EXIT_FAILURE);

      hr = StringCbCatA(errormsg, 1044, " is not a valid ID");

      if (FAILED(hr)) exit(EXIT_FAILURE);
  }
}

void func2(int f2d) {
   char *buf2;
   size_t len;
   int ret;

   ret = read(f2d, &len, sizeof(len)); // flawfinder: ignore

   if (ret == -1) {
     perror("error on first read (func2)");
     exit(EXIT_FAILURE); // errno is set
  }

   buf2 = (char *)calloc(len + 1, sizeof(char));

   if (buf2 == NULL) exit(EXIT_FAILURE);

   ret = read(f2d, buf2, len); // flawfinder: ignore

   if (ret == -1) {
     perror("error on second read (func2)");
     exit(EXIT_FAILURE);
  }

   buf2[len] = '\0';
}

```

```
void func3(int f3d){
    char *buf3;
    int len, ret;

    ret = read(f3d, &len, sizeof(len)); // flawfinder: ignore

    if (ret == -1) {
      perror("error on first read (func3)");
      exit(EXIT_FAILURE);
   }

    if (len > 8000) {
        fprintf(stderr, "error on too large length"); // flawfinder: ignore
        exit(EXIT_FAILURE);
    }

    buf3 = (char *)calloc(len, sizeof(char));

    if (buf3 == NULL) exit(EXIT_FAILURE);

    ret = read(f3d, buf3, len); // flawfinder: ignore

    free(buf3);
    buf3 = NULL;

    if (ret == -1) {
      perror("error on second read (func3)");
      exit(EXIT_FAILURE);
   }
}

void main() {
    char *foo = "foooooooooooooooooooooooooooooooooooooooooooooooooo";
    char *buffer = (char *)calloc(MAIN_BUFFER_SIZE, sizeof(char));
    HRESULT hr;

    if (buffer == NULL) exit(EXIT_FAILURE);

    hr = StringCbCopyA(buffer, MAIN_BUFFER_SIZE, foo);

    free(buffer);
    buffer = NULL;

    if (FAILED(hr) && hr != STRSAFE_E_INSUFFICIENT_BUFFER) exit(EXIT_FAILURE);

    func1();
    func3(strnlen(foo, 53));
    exit(EXIT_SUCCESS);
}
```