Introduction to Secure Software Coding Practices for RDK & CPE

Introduction

- Why conduct training in using secure techniques to develop code for RDK and other CPE products?
  - "Practicing strong information and cybersecurity is a nonnegotiable requirement for organizations doing business today." – *Carnegie Mellon Software Engineering Institute CERT*
- It's hard to disagree. Therefore, the RDK development community's policy is to strongly encourage all developers to be familiar with best practices. Our goal is to eventually establish best practices as a formal policy, where code not following security guidelines will be flagged during check-in.
- This training is an *introduction* to some of the most commonly encountered issues and techniques to ensure a minimum level of compliance with secure coding practices.
- References and additional resources listed at the end of this training should be studied to build a more complete skill set.

The Goals of Using Secure Coding Methods

- Again, from CMU/SEI CERT:
  - "Reduce the number of vulnerabilities to a level that can be fully mitigated in operational environments"
  - "Eliminating vulnerabilities during development can result in a two to three orders-of-magnitude reduction in the total cost of repairing the code versus making the repairs afterwards."
- These goals area high bar. This training is the a step towards realizing them
- This is only an introduction . Developers *must* refer to the *CERT C/C++ Coding Standard* for additional detailed and authoritative standards.

Categories

- String Functions
- Formatted String Functions
- Tainted Strings
- Arrays and Buffers
- Dynamic Memory
- Pointers
- Integers
- Casts
- Concurrency
- Paths and Files
- Using assert()
- Tools

String Functions

- strcpy(dest,src)
- *Must* be replaced by strncpy(dest,src,len) where the value of len is guaranteed to the LESS THAN the available memory at dest.
  - Leave room for terminator byte
  - Don’t use source string length
  - Be vigilant for edge cases, off-by-one, ends of strings
- Examples:

```
char dest[4];
const char *src=“1234”;
const char src2[] = “12345”;
strcpy(dest,src); /* ERROR: BUFFER OVERFLOW, terminator byte */
strncpy(dest,src,sizeof(dest));  /* ERROR: UNTERMINATED STRING */
strncpy(dest,src2,sizeof(src2)); /* ERROR: Don’t use source string length! */
strncpy(dest,src,sizeof(dest)-1);  /* dest = “123” and UNterminated */
dest[3] = ‘\0’;                    /* string is terminated, “123\0” */
```


RDK Management, LLC

RDK REFERENCE DESIGN KIT

String Functions (continued)

- strcat(dest,src)
- *Must* be replaced with strncat(dest,src,len) where len is guaranteed to be *less than* the available memory available at dest
  - Leave room for terminator byte, it is always included in the concatenation
  - Don’t use source string length
  - Be vigilant for edge cases, off-by-one, ends of strings
- Examples:
- char dest[8];
- const char *src=“1234”;
- strncpy(dest,src,sizeof(dest)); /* Set up initial string */
- strncat(dest,src,sizeof(dest)); /* ERROR – Overflow: there are not 8 bytes remaining! */
- strncpy(dest,src,sizeof(dest)); /* Set up initial string */
- strncat(dest,src,sizeof(dest)-strlen(dest)); /* ERROR – No room for terminator */
- strncpy(dest,src,sizeof(dest)); /* Set up initial string */
- strncat(dest,src,sizeof(dest)-strlen(dest)-1); /* OK */

String Functions (continued)

- strlen(a)
- *Should* be replaced by strnlen(str,len) when processing untrusted data, where the value of len is less or equal to the length of the longest valid string (with terminator) that could legitimately be stored at “str” or reflects another limiting boundary.
- Why?
  - Often a string length is obtained for use in subsequent operations (e.g. memory allocation)
  - Exceptions can be forced with unreasonable string lengths
  - Unreasonable lengths used in combination with other string operations can lead to swamping the processor
  - May help mitigate other errors, such as unterminated strings
- If parameter str is a const string literal (e.g. in a table of string consts), a counted function is unnecessary.

String Functions (continued)

- strcmp(str1,str2)
- *Should* be replaced by strncmp(str1,str2,len) for untrusted input, where the value of len is less or equal to the length of the longest valid string (with terminator) that could legitimately be stored at “str1” and “str2” or reflects another limiting boundary.
- Why?
  - Exceptions can be forced with unreasonable string lengths
  - May help mitigate other errors, such as unterminated strings
- The length must be long enough to account for the termination byte of one of the strings, or else a prefix match will result.

String Functions (continued)

- Issues to consider:
  - There are other approaches to string security - consider use of a secure C string library.
  - The functions strstr and strchr are not well supported with secure versions but an implementation is on Confluence.
  - Length/capacity values are often not in scope where they are needed. The call chain may need to be modified to pass them.
  - Use the basic_string class (and subclasses) in C++ when possible
    - Even basic_string is not foolproof. Study the references

Formatted String Functions

- Note: Details of some functions may vary depending on the character set or encoding. Developers may need to adapting these guidelines for variations.
- sprintf (and related functions)
- *Must* be replaced by snprintf(dest,len,format,…)
  - len must be <= the available memory available at dest
  - Return value of < 0 means the function failed
  - Return value of **>=** len means the string was truncated and *must not* be used to account for the length of the actual output string, but is the number of characters that would have been output.
  - Watch out for off-by one errors!
  - Resulting string *will* have a 0-termination

Formatted String Functions (continued)

- Untrusted data *must not* be used as a format string. Strongly prefer const format strings
- The %n format specifier can be used to write arbitrary values and *must not* be present. If a use case is encountered where its use seems mandator, a security team engineer *must* review the use case before implementation
- Validate runtime library: some versions ignore the length in snprintf()!
- Enable compiler checking to validate arguments to a formatted string function
  - -Wformat
  - -Wformat-nonliteral
  - -Wformat-security

Tainted Strings

- Tainted strings come from untrusted sources:
  - Files
  - User Input
  - Network and other Device I/O
  - Environment variables
  - Command parameters
- Threats:
  - Unsafe usage
    - Format strings. See above: untrusted input *must not* be used for string formatting
  - Data-as-code
    - Command injection, major problem:

```
char cmdbuf[256] = “chmod 666 “;
fgets(cmdbuf+strlen(cmdbuf), sizeof(cmdbuf)-strlen(cmdbuf)-1, file);
system(cmdbuf);  /* VULNERABILITY – untrusted input */
```

    - The string in the file can be crafted to cause arbitrary shell commands to execute
  - Resource violations
    - Buffer overruns, memory exhaustion

Tainted Strings (continued)

- **Untrusted input *must not* be used as function invocations, parameters to functions, as query parameters, JSON elements, DOM elements, etc. without sanitization**
- String sanitization *should* use a common function with clear definitions of sanitization options.
  - Stripping metacharacters
  - Quoting may be appropriate
  - Different functions can be used to define sanitizing schemes for shell scripts, JSON, user input, web forms, etc.
  - Research options in the references and on the web
- All untrusted string inputs *must* be length-limited

Arrays and Buffers

- Except in trivial cases:
- Array dimensions *must* be defined by symbolic constants and not by “magic numbers”
  - Array definitions like int16_t myarray[100]; can lead to the introduction of overrun errors. Use #define, static consts, e.g.
    - #define ARRAYMAX 100
    - int16_t myarray[ARRAYMAX];
- Accessor variables *must* use symbolic dimension constants and not “magic numbers.”
  - That way if the array dimension is ever changed, all accessors are still correct
    - for(i=0;i<ARRAYMAX;i++) {} /* Correct */
    - for(i=0;i<100;i++){} /* error prone, avoid this */

Arrays and Buffers (continued)

- Data buffer access not within a limit-checked loop must be bounds-checked to ensure the access is legitimate. If the buffer access is in a performance-sensitive function the bounds check may be in a debug-only build.
- Buffer bounds checks must check for underrun as well as overrun
- If a bounds check violation is detected, appropriate action must be taken, consistent with system design and policy

Arrays and Buffers (continued)

- Bad Example:

```
static int buffer[123];          /* VULNERABLE - “Magic number” */
void somefunc( int somevalue )
{
  int someIndex = someCalculation();
  buffer[someIndex] = somevalue;    /* VULNERABLE – no bounds check */
}
```

- Good Example:

```
#define SOME_BUFFER_LENGTH 123      /* Use symbolic dimension */
static int buffer[SOME_BUFFER_LENGTH];
bool somefunc( int somevalue )      /* Add error return code */
{
 int someIndex = someCalculation();
 if (someIndex < 0 || someIndex >= SOME_BUFFER_LENGTH) {
        return false;             /* Do no harm if range error */
 }
 buffer[someIndex] = somevalue;
 return true;
}
```

Arrays and Buffers (continued)

- The ARRAY_LENGTH construct *may* be used, where it is defined as
  - #define ARRAY_LENGTH(x) (sizeof(x)/(x)[0])

  and “x” is a dimensioned array
- ARRAY_LENGTH *must not* be used on a function parameter or pointer variable
- Arrays *should* be strongly typed when their use includes passing as function parameters:
  - static const my_buffer_length=10;
  - typedef char my_buffer_type[my_buffer_length];
  - void myfunc(my_buffer_type buffer);

Arrays and Buffers (continued)

- Buffer Overflow Detection
  - "Canaries": can be inserted by compiler into stack to detect overruns of local arrays. Can be useful, but are limited.
  - GCC "ProPolice" invoked by
    - -fstack-protector: protection for vulnerable functions, *should* be invoked for production builds
    - -fstack-protector-all: protection for all functions, *should* be invoked for debug builds
- Inhibiting overflow attacks
  - ASLR: Address Space Layout Randomization *should* be enabled for production builds

Dynamic Memory

- Beware common dynamic memory utilization errors:
  - Referencing uninitialized memory
  - Referencing freed memory
  - Multiple frees
  - Missing frees (memory leaks)
  - NULL pointer accesses
  - *ALL of these errors can be exploited with security impacts*
- Return values from memory allocation functions *must* be checked
- NULL pointers can be exploited. NULL pointers *must not* be dereferenced
- Checks for invalid pointer values *should* be made in critical functions, especially where related to security operations

Dynamic Memory (continued)

- 0-length allocations *must* be avoided, especially in calls to realloc()
  - Edge cases that are implementation dependent and error prone
- free() *must not* be used with pointers to memory allocated with new()
- delete() *must not* be used with pointers to memory allocated with malloc()
- delete() *must not* be called with pointers to memory allocated using new[] ()
- RAII *should* be used at all times: allocate resources in a constructor, free resources in a destructor
- Pointer variables *must* be set to NULL after they are freed, unless they are trivially going out of scope
- Valgrind memcheck *should* be incorporated into development processes

Pointers

- Many function pointers are vulnerable to attack: function pointers in the ELF Global Offset Table, C++ vtables, the stack pointer, pointers to global destructors, atexit, exception handlers, jmpbuf
- Compromise of a function pointer = attacker has complete control
- Prevent compromise! Follow guidelines and requirements:
  - **Prevent buffer overflow, arbitrary memory writes via format strings, and inappropriate use of tainted input**
- Securing pointers: encryption/obfuscation. See references

Integers

- Integer arithmetic can lead to security breaches
- Sources of integer errors:
  - Overflow/underflow/wraparound
  - Datatype conversion, truncation
  - Signed/unsigned mismatch
  - The references have details on the types of errors and how they occur
- Integer errors introduce vulnerabilities when the incorrect results are used to:
  - Allocate memory
  - Access array elements
  - Perform timing functions

Integers (continued)

- Array accessors *must* be range-checked or clamped to valid values when not trivially known to be correct
- Tainted integer variables from external untrusted sources *must* be sanitized/range checked in all cases, before any use
- Memory allocation sizes *should* be sanity-checked and kept “reasonable”
- Integer operations *must* be coded to prevent overflow, underflow, truncation, loss of sign, incorrect conversions, implementation-dependent side effects

Integers (continued)

- Typedefs *may* be used to help enforce type safety with integers. Consider the use of wrapper functions for critical applications. Use abstract types in C++
- The size_t type *must* be used for memory allocation, lengths
- Consider secure integer libraries for critical operations
- Range-checking integer variables will prevent all integer security vulnerabilities.

Casts

- In C++, C-style casts *must not* be used
- See CERT standards for recommended use of dynamic_cast, static_cast, and reinterpret_cast
- Casting between pointer types and integer types *should* be avoided
- Casting to avoid strong typing *should* be avoided
  - typedef enum { 1,2,3 } mystrongtype;
  - void mystrongfunction(mystrongtype myvar) { … }
  - for (int i=0;i<100;i++) { mystrongfuncton((mystrongtype)i); } // WRONG

Concurrency

- How to program concurrent code is out of scope for this training but note:
  - Be aware that compilers may reorder instructions making code that appears thread-safe no longer safe.
  - Security vulnerabilities that may be introduced by incorrect concurrency include:
    - Deadlock (DoS)
    - Breaking security-sensitive algorithms
    - Exposing concurrency errors in system services
- Study the references

Paths and Files

- Path and directory strings *must* be “canonicalized” before use, including resolution of symlinks
  - “.”, “..”, trailing “/”, and other path element tricks can cause path-compare operations to be incorrect
  - Canonicalization is tricky. See the references.
- Symlinks expose vulnerabilities. They *should* be avoided if possible and *must* be used carefully, resolved during canonicalization.
- Hard links are less vulnerable but still *must* be used carefully
- Secure partitions *should* be separate from non-secure partitions

Paths and Files (continued)

- Functions using path/filenames *must* ensure a device is not specified where a file is expected.
  - Device files are vulnerable when accessed by operations that are only appropriate for normal files
- Files *must* be locked or synchronized between access checks and usage
  - Since files can be global objects, they are subject to race conditions
  - "TOCTOU" – time-of-check/time-of-use vulnerability very common.

Using assert()

- Use assert() liberally to force runtime exceptions in debug builds for invalid, unexpected, or undefined results
- Use static compile-time asserts to detect the failure of expected compile-time
- Example (from https://github.comcast.com/CPT/xfinity-classic/blob/master/general/src/general.h)
  - #define CASSERT(predicate, dbg_name) \
  - typedef char __ATPASTE4(cassert__,dbg_name,__Line_,__LINE__) [2*!!(predicate)-1]
  - CASSERT(sizeof(uint32_t) == sizeof(int32_t), my_check_11)

Tools

- Compiler warnings and errors must be set to the most stringent level available, e.g. –Wall
- Compilation *must* complete without warnings or errors
- Coverity static source code scans *must* be run on all sources, and no “high impact” security-related defects found
  - As of June ‘15, the exact scan criteria for acceptance is still being defined
- Valgrind memory analysis *should* be incorporated into debug builds to check for memory errors.
- QA automated testing *should* include a pass with a Valgrind-enabled production build
- Modules *should* be subjected to extensive automated black-box fuzz testing
  - Specific tools vary, for example, for C/C++ libraries http://lcamtuf.coredump.cx/afl/
- More tools will be brought online over time

Other General Security Issues

- Object Reuse: after using a security-sensitive variable, its value *must* be overwritten
- Principle of least privilege *must* be applied: do not open a file for writing if you only need to read
- All cryptographic operations *must* use an approved library with a strong source of randomness
- Function parameters *must* be checked for validity
- Outdated security algorithms and protocols *must not* be used
- Threat analysis *must* be performed on non-trivial features and subsystems
- A security-conscious process *must* be followed when developing software

References

- Authoritative References
  - Seacord, Robert – Secure Coding in C and C++, *Addison-Wesley 2013*
  - Carnegie Mellon University Software Engineering Institute CERT – https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard
  - op.cit. http://www.cert.org/secure-coding/index.cfm
- Supplemental References
  - Wheeler, David A., Secure Programming HOWTO, http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html
  - OWASP – Open Software Security Community, https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project

Note about Knowledge Checks

- These are code fragments meant to illustrate concepts, and may include constructs that do not compile as-is.

Knowledge Check: Strings

- What is wrong with the following code:

1. static const int len1=12;
2. char buffer[len1];
3. char otherstring[] = “otherstring”;
4. strncpy(buffer, “mystring”, sizeof(buffer)-1);
5. strncat(buffer, otherstring, sizeof(otherstring));

① “len1” uses a magic number
② buffer is not 0-terminated after line 4
③ otherstring is too long
④ The length parameter in line 5. is wrong

Correct answer: 4. The correct code *for this example* is
strncat(buffer, otherstring, sizeof(buffer)-strlen(buffer)-1);

Knowledge Check: Strings

- Is the following code correct?

1. static const int len1=11;
2. char buffer buffer[len1];
3. strncpy(buffer, ”first string”, len1);
4. strncat(buffer, “second string”, sizeof(buffer)-strlen(buffer)-1);

① Yes
② No

Correct answer: NO. The strncpy on line 3 will leave the string unterminated. The strlen call on line 4 will return a random number >= sizeof(buffer), making the expression sizeof(buffer)-strlen(buffer)-1 a negative number, which will overflow, resulting the strncat call overrunning the memory allocated to buffer. One way to make the code work without this occurring, is: add “buffer[len1-1]=‘\0’”; after line 3. to terminate the string.

Knowledge Check: Formatted String Functions

- What is wrong with this code:

```
1.  #define LEN=12;
2.  char buffer[LEN];
3.  void myfunction( char *string ) {
4.      sprintf( buffer, string );
5.  }
6.  int main( int argc, char *argv[] ) {
7.      myfunction( argc > 1 ? argv[1] : “Invalid\n” );
8.  }
```

① sprintf on line 4 should use counted function snprintf
② main() doesn’t handle missing command parameters
③ The definition at line 3. should be “myfunction(char const *string)”
④ Untrusted input is used as a format string
⑤ 1 and 4
⑥ 2 and 3
⑦ All of the above

Correct answer: (5)

Knowledge Check: Formatted String Functions

- Where are the errors in the following code? Check all that apply

```
1.  static const int len1=80;
2.  void myfunction( int foo ) {
3.    char buffer[len1];
4.    int printed;
5.    snprintf(buffer, sizeof(buffer), "%*cIs this %s code?%n\n",foo,' ',
          foo > len1 ? "good" : "bad", &printed, printed);
6.  }
```

① The sizeof() expression in line 5 is too small
② The "%n" format specifier is used
③ The variable "printed" on line 4. should be unsigned
④ The width specifier will be evaluated before the %n value is evaluated
⑤ Buffer is too big to be allocated on the stack
⑥ The number of arguments to snprintf is wrong

Correct answer: 2,6

Knowledge Check: Tainted Strings

- Select the best code from these choices (assume char buffer[100]):

1. fgets(buffer, sizeof(buffer), stdin); system(buffer);
2. gets(buffer); system(buffer);
3. fgets(buffer, sizeof(buffer), stdin);
   for(int i=0;i<strlen(buffer);i++) {
   if (buffer[i] == ';' || buffer[i] == '|') { buffer[i] = ' '; }
   }
4. fgets(buffer, sizeof(buffer), stdin);
   if (!fork()) { execlp("sh", "-c", buffer); }
5. fgets(buffer, sizeof(buffer), stdin); stripShellMetachars(buffer); system(buffer);

Correct answer: 5. Use centralized function to sanitize string input before passing to an external string sink. It's better to pass a sanitized string to system(), than an unsanitized string to execlp().

Knowledge Check: Tainted Strings

- Which of the following does *not* result in untrusted input:

1. char *mypath = getenv(“PATH”);
2. const char *mycommand = “ls –l”;
3. fread(buffer,sizeof(buffer),1,”/opt/script”);
4. char *mycommand = argv[1];

Correct answer: 2. A hardcoded const string would not be considered untrusted input

Knowledge Check: Arrays and Buffers

- How can you protect against stack-smashing attacks?

1. When using gcc, build with the –fstack-protector option
2. Insert “stack-gerbils” after array definitions
3. Always bounds-check array accesses, especially writes
4. Never use local buffer arrays
5. 1. and 3.
6. 2. and 4.

Correct answer: 5

Knowledge Check: Arrays and Buffers

- Is the following code secure with respect to buffer overruns?

1. int index, value;
2. static const int arraylen=100;
3. int arrayvals[arraylen];
4. fscanf(stdin,"%d,%d",&index,&value);
5. if (index < arraylen) { arrayvals[index] = value; }

① Yes, the bounds checking makes the code secure

② No, there's something missing

Correct Answer: 2. The bounds check does not check for an index < 0, potentially allowing buffer underrun

Knowledge Check: Dynamic Memory

- In C++, what is the best way to avoid memory leaks?

1. Only use static memory buffers
2. Use only new and free
3. Use RAII
4. Have your code reviewed
5. All of the above

Correct Answer: 3. Resource Acquisition Is Initialization, ties resources to object lifetimes. It doesn't prevent all leakage, but it supports good cohesion between allocation of resources in a constructor, and freeing of resources in a destructor.

Knowledge Check: Dynamic Memory

- What is wrong with the following code, check all that apply

```
1.  void myfunction(void) {
2.      int *buffer;    int i;
3.      static const int mylength=123;
4.      if (buffer == NULL) { buffer=malloc(mylength); }
5.      for( i=0;i < 100; i++ ) { buffer[i] = i; }
6.      printf("Done!\n");
7.  }
```

① mylength is too big and is defined locally
② buffer is uninitialized, but checked for NULL
③ There is no bound check on the array access on line 5.
④ buffer should be an array of unsigned int.
⑤ The index count should be from 1 to 100
⑥ The loop limit is not defined in terms of the actual buffer length
⑦ There is no NULL check after the malloc on line 4.
⑧ The allocated memory is not freed at the function exit

Correct Answer: 2., 3., 6., 7., 8. are all errors

Knowledge Check: Pointers

- Which of the following creates a potential arbitrary code execution vulnerability?

1. A jump table generated by the compiler for handling a switch statement
2. An array passed as a parameter to a function
3. A C++ global object definition
4. Missing NULL check after new()
5. Allocating too large a buffer

Correct answer: 3. Allocating a global object makes the global pointer to the object destructor vulnerable.

Knowledge Check: Integers

- What is *not* a source of integer errors?

1. Using casts, resulting in unexpected sign extension
2. Overflow of intermediate results in multi operator expressions
3. Undefined compiler behavior occurring in expression evaluation
4. Range checking all variables before use in an expression
5. Truncation resulting from assignment

Correct answer: 4. is the one item in the list guaranteed to ensure integer errors don’t occur

Knowledge Check: Integers

- Is the following code correct?

1. long var = (16 << -2);
2. unsigned int foo = var * -100000;
3. if (foo == -400000) { printf(“Correct!\n”); }

①Yes
②No
③Maybe

Correct answer: 2. No. This is undefined, and there are a lot of confused interpretations – the compiler is free to do what it wants. Keep it simple, avoid areas of tangled interpretation. Don’t left shift negative values. Be familiar with unclear portions of the standards and avoid them.

Knowledge Check: Casts

- What's wrong with this code:
- for(int i = 0; i<1000; i++) { cout << "the value of somefunc is " << somefunc((unsigned int)i) << "\n"; }

①It is in C++ when it doesn't need to be

②It's missing a cast

③The C-style cast in C++ is to be avoided

④The cast will thrown an exception if the code is changed to "for (int i = -1000; i < 1; i++)

Correct answer: 3.

Knowledge Check: Concurrency

- Which of the following are potential security issues with concurrent coding – check all that apply:

1. Makes the code harder to understand
2. Deadlock can result in Denial of Service attacks
3. Subtle process timing issues can increase the opportunities for side-channel attacks
4. Variables can be left with sensitive values
5. The compiler can manipulate security-sensitive code in unexpected ways
6. Thread privileges can cause security violations

Correct answer: 2., 3., 5.

Knowledge Check: File Systems

- What is wrong with the following code? Check all that apply:

```
1. static const int len1=1000;
2. char buffer[len1];
3. fgets(buffer,len1,stdin);
4. if ( checkfileaccess( buffer ) == ACCESS_OK ) {
5.   int newfile = open(buffer,O_RDWR);
6.   read( newfile, buffer, len1 );
7.   close( newfile );
8. }
```

① Data is read into the buffer used for the filename
② The file is opened for writing even though data is only read
③ Open and read are used instead of fopen and fread
④ There is a TOCTOU error
⑤ Untrusted input is used as a file path without canonicalization and checking

Correct answer: 2., 4., 5.

Knowledge Check: Asserts

- Where should/could an assert be added in the following code:

```
1.  static const int len1=1000;
2.  char buffer[len1];
3.  fgets(buffer,len1,STDIN);
4.  if ( checkfileaccess( buffer ) == ACCESS_OK ) {
5.    int newfile = open(buffer, O_RDWR);
6.    read( newfile, buffer, len1 );
7.    close( newfile );
8.  }
```

① Line 3.: It should read “assert(fgets(buffer,len1,STDIN)!=NULL);”
② Between lines 5. and 6.: “assert(newfile != -1);”
③ Both 1. and 2.
④ None of the above

Correct answer: only 2. If the assert is added to line 3., as in answer (1), the fgets call could be omitted. Never use an expression with a side-effects as an argument to assert()

Thank You