8 Program Design and Security

The objective of this chapter is to introduce security as another program design criteria.

8.1 Preconditions

The following should be true prior to starting this chapter.

- You understand four program design criteria: separation of concerns, design for reuse, design only what is needed, and performance.
- You have evaluated program code using these four criteria.

8.2 Concepts and Context

As is evident in data breaches that are frequently disclosed, developing more secure software is an issue that requires serious attention. This chapter defines security and information security, and explores information security from a programmer's perspective.

8.2.1 Security

Security can be defined as "the condition of not being threatened, especially physically, psychologically, emotionally, or financially" [1]. While security has tradi-

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_8

tionally been thought of in terms of protecting physical assets, the protection of information assets has become an important consideration when developing software.

The need to secure both information and information systems is not a new dilemma. The Morris worm was written by a computer science student in November 1988 [2]. While the author of this worm claims that its purpose was to determine the size of the Internet, the Morris worm became an early example of a *denial-of-service attack*. This worm resulted in a denial of service since it was designed to run multiple times on the same server. Eventually, a server had so many processes running the Morris worm that it was unable to run services related to it its intended purpose. This worm was an early indicator of just how important software security would become as the Internet and the World Wide Web became the communications backbone for personal and business uses.

The latest computer science curriculum guidelines include a knowledge unit called *information assurance and security* (IAS). This knowledge unit is defined as a "set of controls and processes, both technical and policy, intended to protect and defend information and information systems" [3].

The National Institute of Standards and Technology in the U.S.A. provides the following definitions [4].

Information assurance "Measures that protect and defend information and information systems by ensuring their availability, integrity, authentication, confidentiality, and non-repudiation. These measures include providing for restoration of information systems by incorporating protection, detection, and reaction capabilities."

Information security "The protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability."

Information assurance is an umbrella term that includes information security. Information assurance includes understanding the risks associated with information and information systems, privacy, regulatory compliance, standards compliance, and auditing. Professionals that practice information assurance tend to have multidisciplinary knowledge that may include accounting, fraud examination, forensics, management, systems engineering, security engineering, criminology, and computer science. The NIST Cybersecurity Framework [5] illustrates the breadth of information assurance by describing standards, guidelines, and best practices an organization should adhere to in an effort to minimize cybersecurity-related risks.

Information security is the part of information assurance that focuses on the creation and application of security controls. Information security topics tend to be technical in nature and fall largely within the domain of computer science.

8.2.2 Information Security

There are many security concepts/topics that fall within the information security domain. The following descriptions [3,4,6] represent a subset of these concepts/-topics that are directly related to developing a program design and writing code to reduce the number of security vulnerabilities. Note that, at this point in the book, we are focusing on coding practices that will reduce the number of security vulnerabilities. We will discuss software design principles related to developing more secure software in Chap. 24.

Data input validation Data input into a software application, regardless of its source, shall be validated. Examples of input data sources include a user entering data, another software system sending data to our application, and reading data from a persistent data store.

For instance, server software should never assume that data coming from a client has already been validated. This is because a malicious attacker could inject specially formatted data into a client that lacks input validation or could build/use their own client that sends requests to the server [7]. An example of this type of attack was discovered by TripWire that affects three of the top-selling smart home hubs sold on Amazon [8]. As stated in this article, the attacks can be done through malicious web pages or a smartphone application. This implies that the software on the hub is not validating requests it receives from a client.

Data output validation Data output by a software application, regardless of its destination, shall be validated. Examples of output data destinations include displaying data to a user, sending data to another application, and storing data in a persistent data store.

Writing to a log file, data file, or a database can place a “logic bomb” or “data bomb” into a persistent data store that gets activated at some later point based on a specific state being reached [7].

Exception handling Design code to correctly handle software-based exceptions that occur during execution.

Arithmetic errors, buffer overflows, and security exceptions are three types of exceptions that, when not caught and dealt with properly, may result in a situation that a malicious attacker can take advantage of. From a programming perspective, it is easy to detect these types of exceptions. Once the exception is detected (i.e., trapped or caught) it is fairly easy to write code that reacts in a secure way to the error. And yet these types of vulnerabilities continue to exist in software. One of the most popular sets of exploits take advantage of buffer overflows, where the data being stored in memory is larger than the amount of memory allocated for storage of the data. In fact, the book written by Hoglund and McGraw [7] on exploiting software has an entire chapter on buffer overflow.

Fail-safe defaults When a software failure occurs, design the default behavior of the software to fail in a way that denies access to the data.

This coding practice is tied to exception handling and other forms of detecting errors. Any error that is discovered should result in a program state that denies access to data or denies further processing that may lead to accessing data.

Type-safe languages A programming language is type-safe when it raises an exception when an operation is attempted on a value whose data type is not appropriate. The exception may be raised in a static context (i.e., while compiling the code) or in a dynamic context (i.e., while executing the code).

The first four program design security criteria listed above will be further explored in Chaps. 9 and 10 on object-oriented programming (OOP) and structured programming (SP), respectively. Type-safe languages are discussed next.

8.2.2.1 Type-Safe Languages

Of the three programming languages used in this book, Java and Python are type-safe languages. Any attempt at applying an operation on a value not of the appropriate data type will raise an exception. In contrast, C++ is not a type-safe language.

8.2.2.2 Java: A Type-Safe Language

In the case of Java, the compiler will often catch the misuse of a type by reporting a syntax error. (This is called static type checking.) The following examples illustrate the misuse of a type in Java caught by the compiler.

When the Java class in Listing 8.1 is compiled, the compiler generates the syntax error shown in Listing 8.2. Since an *int* in Java is a primitive data type, a variable of type *int* cannot be used to invoke a method.

Listing 8.1 BadMethodCall.java

```java
public class BadMethodCall
{
    public BadMethodCall()
    {
        int a = 5;
        a.callMethod();
    }
}
```

Listing 8.2 BadMethodCall.java Syntax Error

```
BadMethodCall.java:6: error: int cannot be dereferenced
          a.callMethod();
```

When the Java class in Listing 8.3 is compiled, the compiler displays the syntax error shown in Listing 8.4. Since the variable *a* has not been defined prior to the *a.callMethod()* statement, the compiler does not know what variable *a* represents.

Listing 8.3 BadVariableName.java

```
public class BadVariableName
{
    public BadVariableName()
    {
        a.callMethod();
    }
}
```

Listing 8.4 BadVariableName.java Syntax Error

```
BadVariableName.java:5: error: cannot find symbol
         a.callMethod();
         ^
    symbol: variable a
```

When the Java class in Listing 8.5 is compiled, the compiler displays the syntax error shown in Listing 8.6. The plus symbol in Java does not have a definition allowing the first operand to be an *int* and the second operand to be a *String*.

Listing 8.5 BadUseOfOperator.java

```
public class BadUseOfOperator
{
    public BadUseOfOperator()
    {
        int a = 5;
        int b = a + "five";
    }
}
```

Listing 8.6 BadUseOfOperator.java Syntax Error

```
BadUseOfOperator.java:6: error: incompatible types: String cannot
be converted to int
     int b = a + "five";
               ^
  required: int
  found: String
```

There are also cases where Java will catch the misuse of a type while the program is being executed. This is called dynamic type checking. The code in Listing 8.7 illustrates the misuse of a type in Java detected at runtime. When this program is executed, it causes the runtime exception shown in Listing 8.8. This runtime exception occurs on the *sum = sum + (double)numbers.get(idx)* statement during the second iteration of the *for* loop. The Float object added to the *numbers* ArrayList (see *//11* in listing) cannot be cast into a Double object.

Listing 8.7 BadCast.java

```
import java.math.BigDecimal;
import java.util.ArrayList;

class BadCast
{
    public static void main(String[] args)
    {
        ArrayList<Object> numbers;
        numbers = new ArrayList<Object>();
        numbers.add(new Double(3.14159265358979323846264338327 95));
        numbers.add(new Float(3.14159265358979323846264338327 95)); //11
        numbers.add(new BigDecimal(3.14159265358979323846264338327 95));
        double sum = 0;
        for (int idx=0; idx < numbers.size(); idx++)
            sum = sum + (Double)numbers.get(idx); //15
    }
}
```

Listing 8.8 BadCast.java Runtime Error

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Float cannot be cast to java.lang.Double
      at BadCast.main(BadCast.java:15)
```

8.2.2.3 Python: A Type-Safe Language

In the case of Python, the interpreter will catch the misuse of a type while the program is being executed (i.e., Python performs dynamic type checking). The following examples illustrate misuse of a type in Python.

When the Python interpreter runs the code in Listing 8.9, it displays the exception shown in Listing 8.10. This runtime exception occurs since the Python type *int*, which is implemented as a class, does not have a method whose name is *callMethod*.

Listing 8.9 AttributeError.py

```
def func():
    a = 5
    a.callMethod()

func()
```

Listing 8.10 AttributeError.py Runtime Error

```
Traceback (most recent call last):
  File "AttributeError.py", line 5, in <module>
    func()
  File "AttributeError.py", line 3, in func
    a.callMethod()
AttributeError: 'int' object has no attribute 'callMethod'
```

When the Python interpreter runs the code in Listing 8.11, it displays the exception shown in Listing 8.12. This runtime exception occurs since the variable *a* has not been defined prior to the *a.callMethod()* statement. That is, the interpreter does not know what variable *a* represents.

Listing 8.11 Ch5NameError.py

```
def func():
    a.callMethod()

func()
```

Listing 8.12 NameError.py Runtime Exception

```
Traceback (most recent call last):
  File "NameError.py", line 4, in <module>
    func()
  File "NameError.py", line 2, in func
    a.callMethod()
NameError: global name 'a' is not defined
```

When the Python interpreter runs the code in Listing 8.13, it displays the exception shown in Listing 8.14. This runtime exception occurs since the plus symbol in Python does not include a definition where the first operand is an *int* and the second operand is a *string*.

Listing 8.13 TypeError.py

```
def func():
    a = 5
    b = a + "five"

func()
```

Listing 8.14 TypeError.py Runtime Exception

```
Traceback (most recent call last):
  File "TypeError.py", line 5, in <module>
    func()
  File "TypeError.py", line 3, in func
    b = a + "five"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

8.2.2.4 C++: Not a Type-Safe Language

As mentioned, C++ is not a type-safe language. Listing 8.15 shows one example of why C++ is not a type-safe language. This code declares, initializes, and displays the contents of three variables. First, the *value* variable is declared as an *int* that is initialized to -1. Second, the variable *p_int* is declared as a pointer to an *int* that is initialized to the memory location of the *value* variable. Finally, the variable *p_float* is declared as a pointer to a *float* that is also initialized to the memory location of the *value* variable. This C++ NotSafe program produces the output shown in Listing 8.16.[1]

Listing 8.15 NotSafe.cpp

```
#include <iostream>

using namespace std;

void displayValues(int* ptr_int, float* ptr_float);

int main()
{
    int value = -1;
    int* p_int = &value;
    float* p_float = (float*)p_int;

    displayValues(p_int, p_float);
    displayValues(p_int+1, p_float+1);
    displayValues(p_int+10, p_float+10);

    return 0;
}

void displayValues(int* ptr_int, float* ptr_float)
{
    cout << ptr_int << "\t" << *ptr_int << "\t"
            << ptr_float << "\t" << *ptr_float << endl;
}
```

Listing 8.16 C++ NotSafe Output

```
0x61ff2c  -1  0x61ff2c  -1.#QNAN
0x61ff30  6422300  0x61ff30  8.99956e-039
0x61ff54  6422368  0x61ff54  8.99965e-039
```

The following explains the output produced from running the code in Listing 8.15.

- The first and third values displayed on each output line show the same values. This confirms the pointer arithmetic being performed on memory address values pointing to an *int* and a *float* are consistently resulting in the same memory location. That is, *int* and *float* values are both stored in 32 bits (i.e., 4 bytes).
- The first call to *displayValues* displays the first line of the output.
  The *0x61ff2c* value is the memory address where the *value* variable is stored on the

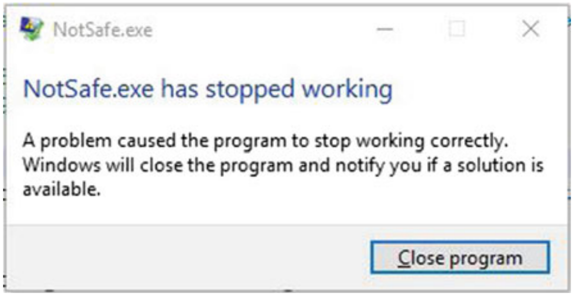
[1]The output shown for the C++ NotSafe program results from compiling and linking this C++ example using GNU C++ (GCC) version 4.8.1 and then executing on Windows 10 Home Edition version 1607 OS Build 14939.1198. The behavior of this C++ program may differ when using a different C++ compiler or operating system.

runtime stack.[2] The *−1.#QNAN* value comes from the **ptr_float* expression in the *cout* statement. This dereferences the pointer to display a floating-point number. Since this memory location contains an integer −1, all 32 bits of the 4-byte integer value are set to a 1. Furthermore, the IEEE floating-point standard defines all 1 bits for a float as representing not-a-number (NaN) value. Specifically, a *quiet NaN* is displayed to indicate the bit-pattern represents an invalid floating-point value.
- The second call to *displayValues* displays the second line of the output.
  The *0x61ff30* value is the result of adding 1 to the *p_int* and *p_float* pointer values. Since *p_int* is a pointer to an int and *p_float* is a pointer to a float, adding 1 results in adding the size of an int or float (both typically 4 bytes) to the *p_int* and *p_float* memory address value. The *6422300* value (which is *0x61FF1C*) is the integer value of the 4-bytes stored at the *0x61ff30* memory address. This value is stored on the runtime stack and represents the memory location for the *p_int* variable, which contains the address where the *value* variable is found on the stack.
- The last call to *displayValues* displays the third line of the output.
  The *0x61ff54* value is the result of adding 10 to the *p_int* and *p_float* pointer values. (As mentioned in the previous item, this results in adding 40 to the *p_int* and *p_float* memory address values). The *6422368* value (which is *0x61FF60*) is the integer value of the 4-bytes stored at the *0x61ff54* memory address. The *8.99965e-039* is the floating-point value of the 4-bytes stored at this same memory address.

This C++ program illustrates two language features that make C++ not type-safe. First, the number of bytes reserved for an *int* value is dependent on the compiler implementation. While the GNU C++ compiler treats *int* as a 4-byte value, other C++ compilers may treat this as a 2-byte value. Second, doing pointer arithmetic is a dangerous way to navigate around memory. For example, a programmer can use pointer arithmetic to alter the contents of the runtime stack or the machine code of the executable image currently in the main memory. Depending on the C++ compiler and operating system, it's possible that pointer arithmetic could result in gaining access to memory locations that are outside the memory range being used by your program (i.e., process).

Making a minor change to the code in Listing 8.15 may result in different behavior. For example, changing *displayValues(p_int+10, p_float+10);* to *displayValues(p_int+100, p_float+100);* will produce the error message displayed in Fig. 8.1. In this example, Windows prevented access to a memory location that was outside the process space.

[2]The runtime stack is used to store information related to each function/method call. Each parameter value passed via the call and any local variables defined within the function/method are stored on the runtime stack. In addition, the return address of where the function/method call should return to is also stored on the runtime stack.


NotSafe.exe

NotSafe.exe has stopped working

A problem caused the program to stop working correctly. Windows will close the program and notify you if a solution is available.

Fig. 8.1 C++ NotSafe windows message

8.3 Post-conditions

The following should have been learned when completing this chapter.

- Developing a safe (more secure) software application involves consideration of a number of factors, including
  - Validate data input into an application.
  - Validate data output by an application.
  - Include an exception handler in code to react in a safe (more secure) way when an exception may occur. In Python, use try-except blocks to react to runtime exceptions. In Java, use try-catch blocks to react to runtime exceptions.
  - Use fail-safe defaults when an error condition or exception is detected, to ensure data is not used in an inappropriate (less secure) manner.
- Making your code more secure often involves adding some complexity to your solution. Testing your code needs to include test cases to determine whether your code responds correctly to both proper and improper use of your application.
- When you have a choice, a type-safe programming language should be used.

8.4 Next Chapter?

If you are interested in seeing how security may be applied to a small object-oriented programming solution, continue with Chap. 9. This chapter will continue the Address Book Application case study using Python and Java.

If you are interested in seeing how security may be applied to a small structured programming solution, continue with Chap. 10. This chapter will continue the Address Book Application case study using Python. Since C++ is not a type-safe programming language, C++ is no longer used to demonstrate structured programming and structured design concepts.

Exercises

Discussion Questions

1. Why is it important to use a type-safe language to improve the security of code?
2. Validating input and output data is a critical part of making more secure software. Adding validation of data to a design would appear to be relatively straight forward.

    a. List the steps that should be performed to validate a date in the format YYYY-MM-DD.
    b. Which of these steps must be changed if you needed to validate a date in the format DD MMM YYYY, where *MMM* is the first three letters of the month name?
    c. Search the World Wide Web to identify two types of attacks that would not be possible if data validation is designed and implemented.

References

1. Wiktionary.org: Security (2019) Wiktionary the free dictionary. Wikimedia Foundation. https://en.wiktionary.org/wiki/security. Accessed 10 Feb 2019
2. Wikipedia.org: Morris worm (2015) Wikepedia the free encyclopedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/Morris_worm. Accessed 29 July 2015
3. The joint task force on computing curricula: computer science curricula (2013) Curriculum guidelines for undergraduate degree programs in computer science. ACM and IEEE
4. National institute of standards and technology: computer security resource center glossary (2019) NIST. https://csrc.nist.gov/glossary. Accessed 10 Feb 2019
5. National institute of standards and technology: cybersecurity framework (2019) NIST. https://www.nist.gov/cyberframework. Accessed 09 March 2019
6. Alicherry M, Keromytis AD, Stavrou A (2009) Deny-by-default distributed security policy enforcement in mobile ad hoc networks. http://www.cs.columbia.edu/~angelos/Papers/2009/manet-securecomm.pdf. Accessed 24 June 2014
7. Hoglund G, McGraw G (2004) Exploiting software: how to break code. Addison Wesley, Boston
8. Tripwire: Tripwire uncovers smart home hub zero-day vulnerabilities. Tripwire Press Release (2015). https://www.tripwire.com/company/press-releases/2015/08/tripwire-uncovers-significant-security-flaws-in-popular-smart-home-automation-hub/. Accessed 10 August 2015