Coping with Security in Programming

Frank Schindler

Department of Applied Computer Science and Engineering
Faculty of Electrical Engineering and Information Technology
Slovak Technical University, Iľkovičová 3, SK-812 19 Bratislava, Slovakia
e-mail: frank.schindler@stuba.sk

Abstract: This article deals with importance of security issues in computer programming. Secure software can only be designed with security as a primary goal. To achieve that we would have to redesign our computer systems with security in our mind including entire computer environment, e.g. hardware, programming languages and, of course, operating systems. In software development process the quality of resulting computer code should be the most important aspect during the whole program development process. Simplicity of the code in computer programs always pays off. Extra options and features can result in unmanageable complexity. For computer security purposes, program modularisation is of a paramount importance and seems to be the only way how to properly cope with complexity. Internal consistency of the whole program should be frequently checked via assertions. They are the best way to check parameter validity coming from other program units e.g. modules. Especially each module must distrust everything else coming from other modules and/or from the user. Frequently used code optimisations are classically leading to some sort of redundant code options and features and thus indirectly causing a useless code complexity. Extensive testing of programs is necessary for finding possible bugs in programs. However it does not locate security holes in the system. Standard software implementation techniques are completely inadequate in the production of a secure code. Consequently an introductory programming course as a college course should be taught in parallel with introductory security of computer systems, since it is too late to teach it as an elective at the end of computer science curriculum. In general, security of computer systems and programming should not be separated as two different and separate disciplines instead of it they should be integrated together.

Keywords: security of computer systems, principle of least privileges, lack of functionality, module, module's specification, module's implementation

1 Introduction

Standard software development techniques are completely inadequate to create secure software, since they only deal with correctness of software e.g. with its specified functionality. If you press the key A, then action B will happen.

Consequently a correct program behaves exactly according to its specification. On the other hand, secure software relates to a lack of functionality (see [3]). No matter what the user (attacker) does, he cannot do action X. It is possible to test the functionality of a program, but there is no known way to test the lack of it.

In real life situations, there are many different ways a computer program can be made to fail or crash. Often this may be easily achieved when the user (attacker) provides invalid input either on purpose or by an accident. Deliberate actions on the side of the user could also include feeding the program with viscous data. Programs react on such inputs in various ways. Some of them simply fail without any error messages, others act incorrectly, and yet others crash the whole operating system. A program that crashes the whole system is unacceptable above all in the computer security area, because it may be possible leading to some sort of security breach. Therefore development of secure software is sufficiently different from programming other software applications. Common program's bugs (e.g. buffer overflows) are the most serious security problems in today's computer systems. To be more specific the biggest problem in computer security is related to the weakest link property. In old days of computing a programmer received a task to be performed, went away and developed the whole program alone. Nowadays programs for complex tasks are programmed by a team of many different programmers that can produce millions of lines of code. The level of output of such a typical programmer involved in the team is on average only 5 or 10 lines of code per day. Large programs require precise design documents showing what each piece of code does and how it interacts with the rest of the program. Bug-free code is duty for all developers on the team, therefore they have to be involved in peer design reviews and peer code reviews. When a programmer finishes a particular part of code other team members must make a complete walk-through of the programmer's design and/or code. Basic principles of software engineering advocate to write small, self-contained program units called modules. Each module should be isolated from harmful effects of other modules. This can be achieved via information hiding (encapsulation). Secure programming implies that each program's (or subprogram's) actions must be contained in the program's specification. To write safe and sound programs we should stick to three basic principles: information hiding, defensive (robust) programming, and assuming the impossible.

2 Information Hiding (Encapsulation)

A module often specifies and implements an abstraction (see [4]). The module's specification describes the behavior and properties of the abstraction, and the module's implementation contains the concrete realization in the program code. Effective programming also takes advantage of reusing existing code libraries and/or off-the-shelf (reusable) components. Each well-designed module should

encapsulate (group and hide) private/public data and the code bodies. Moreover the module should provide well-defined interfaces through which the program can access or modify module's data. Any undocumented, unspecified code options, side effects or function definitions may result in a covert channel or trap door (back door) through which data could either leak, or be changed or damaged and thus they could pose a severe security problem. Most of these ideas are a direct consequence of the "Principle of Least Privilege" and they form the basis for program code integrity and security. During the whole process of the program specification, design, implementation, testing and maintenance keep in mind that simplicity always pays off.

3 Defensive Programming

It is based on the idea that the given program, being executed, should not depend on anything that is not self-created in the program. Every time when a user (attacker) is running the program the programmer must suppose that the user may break it either intentionally or unintentionally by providing flawed input to it. Therefore, insert into your code as many of assertions as possible to catch erroneous data flowing from function (module) to the other function (module). By no means: "garbage-in garbage out". That would be fatal. On the other hand the programmer should never abuse the features of the given programming language like pointers. Especially de-allocation of pointers is a very dangerous operation often leading to dangling pointers. On the other hand, if a pointer is returned by a function it allows illegal access to program's data (data leakage vulnerability). Also the same holds for array indices. Another potential weakness may relate to error codes returned from the functions. When they are left unchecked the values returned from the function should not be used, because they could act as a destructive trash when they are used as input for rest of the code. Avoid use of cryptic code or extra features and options of the programming language that only very few programmers know and use. Technique of defensive programming is often referred to as "robust programming" (see [1]).

4 Assuming the Impossible

Module's specification should be used as a document against which the program should be tested after it has been finished. Without it there is no rigorous way to describe what has been accomplished in the program. Consequently it has to be complete and up-to-date. Let me rephrase it. Anything that is not in there does not have to be implemented in the code. Programmers first write a program and then they test it to see if it functions correctly. Then, if any bugs are found they fix

them and try it again. This process cannot lead to a completely correct program, since this way we are unable to show absence of bugs. Such a program usually works fine in the most typical situations. To verify that a program is correct is way over our capabilities today. Nevertheless we can try to do our best when we are testing programs. Generally, two types of tests are needed. The first one should be generic one made from the module's specification. The second set of tests should examine module's implementation limits, e.g. buffer management errors. Perhaps, designing, writing and testing of a secure computer program could be best compared with driving the car on the busy highway. Programming language, we are using, should never be misused the same way as the car. Defensive driving is a counterpart of defensive programming in this case. Some features like pointers should be used with caution the same way as when we are riding the car we must anticipate that anytime there could be a cat or some other animal running from behind the bush into our way. Therefore numerous assertions should be embedded in the source code in order to improve the quality of code. Anytime in the program there is a possibilities to check the internal consistency of the system you should include an assertion. If it fails it can abort the program and report what was going wrong. This way it is possible to catch up a lot of errors from which some of them could lead to a serious security breach. Producing wrong answers in the program can do a lot more harm. Do not allow garbage data to propagate freely through your program! In order to illustrate some basic concepts, here I provide a few short and simple examples concerning secure programming. Remember that most computing errors happen exactly in cases like these.

5 Programming Examples

5.1 Buffer Overflows

```
int i = 0;
int a[10];              //  Here, buffer is allocated as an array made of 10 integers
....
i = 11;                 //  index i is set over the upper bound of the array
a[i] = 1;
....
// Buffer overflows cause about 50% of the security problems on the
// Internet (Ferguson, 2003).  Algol 60 solved this problem!
// However C, C++ allow buffer overflows!
// Solution: Avoid any such language for secure applications!
```

5.2 Missing Initialization of Variables

```
#include <stddef.h>       //                for NULL
...
int    *ptr;              //                int      *ptr = NULL;
...
free(ptr);
// here a pointer is declared, that has not been allocated, but it is de-allocated.    //
This leads to a typical situation in which operating system can crash.
```

5.3 Cryptic Code

```
char *p1, *p2;
...
while (*p1++ = *p2++)
          ;
// *p1++ is equivalent to:
// *(p1++)  it is a unary operator...right to left
```

5.4 Program Ignoring Error Messages

```
const int NO_ERROR = 0;
....
int  One_Function(char ch, char *ptr);
int err_Message;
char u, v;
....
err_Message = One_Function('?',&v);
u = v;
// it should be:  if  (err_Message == NO_ERROR)  u = v;
```

5.5 Missing Null-Condition Restrictions

```
int  a[5];                    // stack is represented as an array made of 5 integers
int  top = -1;                // this condition means the stack is empty
...
```

```
i = pop();                    // trying to pop the top element from an empty stack
// it always must  be:
if (top != -1)                // see if the stack is non-empty and then pop
   i = pop();
```

5.6 Dangling Pointers

```
#include <stddef.h>       // for NULL
...
int    *ptr1 = new int;
int    *ptr2 = new int;
*ptr2 = 42;
*ptr1  = *ptr2;
delete ptr1;              // avoid an inaccessible object
ptr1 = ptr2;
delete ptr2;
// Notice a missing assignment:
ptr1 = NULL;              // avoid dangling pointer
```

5.7 Assertions

```
#include <assert.h>
// C++ standard library providing executable assertions
.....
double FindAverage ( int    sumOfScores, int    studentCount )
// Precondition:
//                sumOfScores is set
//                studentCount > 0
// Postcondition:
//                average = sumOfScores / studentCount
{
      double average = 0.0;

      assert (studentCount > 0);
```

```
    // this function halts the program if the expression is false
    average = sumOfScores / studentCount;
    return average;
}
```

5.8 Work with Library Functions

It seems to make a perfect sense to ask students to write a program calling some of the standard input/output library functions in C. Their task is to feed them with such an input, that will crash the whole operating system. When that happens they should use debugger to see why the system collapsed.

Example:

```
char a, b, c, s[10];
int  n;
double  x;
scanf ("%c%c%c%d%s%lf", &a, &b, &c, &n, s, &x);
```

As a direct continuation of this assignment students should be asked to write a robust scanf function in C of the same type.

Examples of this sort could be quite useful during the lab sessions associated with the corresponding face-to-face lectures!

5.9 Paradoxes – Testing the Impossible (Variant of "Y2K Bug")

An insurance company offers a life insurance to its clients based on their age. It uses a program reading from a file a list of people consisting of the name and year of birth per line. The year of birth is a two-digit item. Assume we are in 20th century and today we have the first day of January 2000.

Age of the person born in 1925 can be obtained as:

"00" - "25" = -25 years.

Conclusion

Software manufacturers are used to sell their products with many known bugs and therefore they disclaim any legal liability for using their merchandise in the corresponding software license. For some applications like computer games this is not a real problem. However computer programs are more and more used everywhere in our lives. Standard implementation techniques are completely inadequate to create robust software. Software security can be guaranteed only if

all program parts do their job. Low-quality code is the most common cause of real world attacks, and should be avoided. A secure program may just be written by the sound technique of defensive (robust) programming coupled along with information hiding provided it has been tested thoroughly. The mission to test large programs with millions of lines of code is almost impossible. Only thousands (or millions) of users might test it exhaustively. In college programming courses the special attention should be paid to the secure design, implementation and testing of programs systematically. It is not enough when a program runs satisfactorily for correct data only. Moreover secure programs should not cause any unnecessary vulnerability such as information damage, leakage or data diddling by strictly adhering to "Principle of Least Privilege" and information hiding. Although this approach is not going to solve all security problems lurking on us in our programs it could lead to programs that are better structured, better tested, better thought of and at last more secure. The only imaginable way to make secure software would be to redesign our entire computer environment, including hardware, programming languages and operating systems with security as a primary goal in our mind. And, that is time consuming and costly.

References

[1] M. Bishop, D. Frincke: Teaching Robust Programming, IEEE Security and Privacy, published by IEEE Computer Society, Vol. 2, No. 2, (2004) pp. 54-57

[2] M. Blaha: A Copper Bullet for Software Quality Management, Computer published by IEEE Computer Society, Vol. 37, No. 2, (2004) pp. 21-25

[3] N. Ferguson, B. Schneier: Practical Cryptography, Wiley Publishing Inc. Indianapolis, Indiana, (2003) ISBN 0-471-22357-3

[4] M. R. Headington, D. D. Riley: Data Abstraction and Structures with C++, D. C. Heath and Company, Lexington, MA, (1994) ISBN 0-669-29220-6

[5] N. R. Mead: Who is Liable for Insecure Systems? COMPUTER published by IEEE Computer Society, Vol. 37, No. 7, (2004) pp. 27-34

[6] S. Meyers: Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Addison-Wesley, Reading, Massachusetts (1997)

[7] C. P. Pfleger: Security in Computing, Prentice-Hall International, Inc., Upper Saddle River, NJ, (1997) ISBN 0-13-185794-0

[8] F. Schindler: Coping with Safe Programming, Proceedings of Conference "I & IT 2004", Banská Bystrica, Slovakia, (2004) pp. 142-145, ISBN 80-8033-017-7