Robust Programming by Example

Matt Bishop, Chip Elliott

Proceedings of the 7th World Conference on Information Security Education
9-10 June 2011, Lucerne, Switzerland

Robust Programming by Example

Matt Bishop[1] and Chip Elliott[2]

[1] Dept. of Computer Science, University of California at Davis
Davis, CA 95616-8562 USA; bishop@cs.ucdavis.edu
[2] GENI Project Office, BBN Technologies, 10 Moulton Street
Cambridge MA 02138 USA; celliott@bbn.com

Abstract. Robust programming lies at the heart of the type of coding called "secure programming". Yet it is rarely taught in academia. More commonly, the focus is on how to avoid creating well-known vulnerabilities. While important, that misses the point: a well-structured, robust program should anticipate where problems might arise and compensate for them. This paper discusses one view of robust programming and gives an example of how it may be taught.

1 Introduction

The results of poorly written software range from the merely inconvenient to the catastrophic. On September 23, 2010, for example, a software error involving the mishandling of an error condition made Facebook inaccessible for over 2 hours [7]. Medical software, electronic voting systems, and other software [1, 2, 13] also have software problems.

Problems in software often appear as security problems, leading to a demand that students learn "secure programming". While laudable, this focuses the discussion of programming on *security* rather than good programming style. The reason this difference is important lies in the definition of "security".

Security is defined in terms of a security policy, which describes those states that the system is allowed to enter [3]. Should the system enter any other state, a breach of security occurs. "Secure programming" therefore ties programming to a particular policy (or set of policies). As an example, consider a program in which a buffer overflow on the stack will not be caught. The attacker overflows the buffer, uploading a changed return address onto the stack. This causes the process to execute code that the attacker desires. If the program adds privileges to the attacker, for example as a setuid-to-*root* program on a Linux system, the ability of the attacker to perform arbitrary tasks by exploiting this buffer overflow is a violation of any reasonable security policy. So, "secure programming"—in which one focuses on those programming problems that cause security violations–would cover this case.

Consider the same program, but it does not add privileges to the attacker. The attacker can exploit the same buffer overflow flaw as before, but that code will execute with the attacker's *original* privileges. As there is no increase in

privileges, exploiting the buffer overflow flaw usually does not violate a security policy. Hence this is not a "secure programming" problem.

But it is a robustness problem. Such a buffer overflow can cause the program to act in unexpected ways. Robust code would handle the input causing the overflow in a reasonable way. In this case, a robust program would gracefully terminate, telling the user about the invalid input that caused the problem.

In what follows, we refer to "code" when we mean a program or a library. The reader should make the obvious generalization to terms. Thus, "input to code" means any input that the user or environment provide to a program or a library, including the parameters passed and the return value in the case of the latter. "Calling a function" may refer to invoking a program.

The goal of this paper is to discuss the principles of robust programming, and provide an example of how to explain the issues to students. The next section focuses on the principles. We then present an example of non-robust coding, and then show how to write the same library function in a robust way. We conclude with a brief discussion of our experiences using this example.

2 Background

There is amazingly little written about robust coding. Certainly any survey of the literature must begin with Kernighan and Plaugher [9] and Ledyard [10], who provide general rules. Other books focus on specific programming languages [8, 11, 12]. These books provide detailed rules and examples of the application of the rules. Specific exercises and mentoring [5, 6, 4] have also been discussed. This paper aims at a broader scope, enunciating some fundamental principles and then applying them to library functions.

3 Principles

Robust code differs from non-robust, or fragile, code by its adherence to the following four principles:

1. *Be paranoid.* The code must check any data that it does not generate to ensure it is not malformed or incorrect. The code assumes that all inputs are attacks. When it calls a function, the code checks that it succeeds. Most importantly, the programmer assumes that the code will have problems, and programs defensively, so those problems can be detected as quickly as possible.
2. *Assume stupidity.* The programmer must assume that the caller or user cannot read any manual pages or documentation. Thus, the code must handle incorrect, bogus, and malformed inputs and parameters appropriately. An error message should not require the user to look up error codes. If the code returns an error indicator to the caller (for example, from a library routine), the error indicator should be unambiguous and detailed. As soon as the problem is detected, the code should take corrective action (or stop). This keeps the error from propagating.

3. *Don't hand out dangerous implements.* A "dangerous implement" is any data that the code expects to remain consistent across invocations. These implements should be inaccessible to everything external to the code. Otherwise, the data in that data structure may change unexpectedly, causing the code to fail—badly. A side benefit is to make the code more modular.
4. *Can happen.* It's common for programmers to believe conditions can't happen, and so not check for those conditions. Such conditions are most often merely highly unlikely. Indeed, even if the conditions cannot occur in the current version, repeated modifications and later additions to code may cause inconsistent effects, leading to these "impossible" cases happening. So the code needs to check for these conditions, and handle them appropriately (even if only by returning an error indicator).

The defensive nature of robust programming protects the program not only from those who use it but also from programming errors. Good programming assumes such errors occur, and takes steps to detect and report those errors, internal as well as external.

4 The Non-Robust Example

This example is part of a queue management library. It consists of a data structure and routines to create and delete queues as well as to enqueue and dequeue elements. We begin with the queue structure and the interfaces.

```
/* the queue structure */
typedef struct queue {
  int *que; /* the actual array of queue elements */
  int head; /* head index in que of the queue */
  int count;  /* number of elements in queue */
  int size; /* max number of elements in queue */
} QUEUE;

void qmanage(QUEUE **, int, int);/* create, delete queue */
void qputon(QUEUE *, int);       /* add to queue */
void qtakeoff(QUEUE *, int *)    /* remove from queue */
```

This organization is fragile. The pointer to the QUEUE structure means the location of the data, and hence the data itself, is accessible to the caller, so the caller can bypass the library to obtain queue values—or, worse, alter data in the structure. So this encapsulation is not hidden from the caller.

Next, consider the queue manager routine *qmanage*. The first argument is the address of the pointer to the QUEUE structure; the second, a flag set to 1 to create a new queue and 0 to delete the queue; and the third, the size of the queue to be created. If the second argument is 0, the third argument is ignored.

```
void qmanage(QUEUE **qptr, int flag, int size)
{
  if (flag){
```

```
    /* allocate a new queue */
    *qptr = malloc(sizeof(QUEUE));
    (*qptr)->head = (*qptr)->count = 0;
    (*qptr)->que = malloc(size * sizeof(int));
    (*qptr)->size = size;
  }else{
    /* delete the current queue */
    (void) free((*qptr)->que);
    (void) free(*qptr);
  }
}
```

This routine is composed of two distinct, logically separate operations (create and delete) that could be written as separate functions. Thus, its cohesion is low. Poor cohesion generally indicates a lack of robustness.

Indeed, this code is not robust. The arguments are not checked. Given that the last two are integers, a caller could easily get the order wrong. The semantics of the language means that if the second argument is non-zero, the function creates a queue. Thus the call

```
qmanage(&qptr, 85, 1);
```

allocates a queue that can hold at most 1 element. This is almost certainly not what the programmer intended. Further, this type of error cannot be easily detected. Decoupling the two separate functions solves this problem.

Lesson 1. *Design functions so that the order of elements in the parameter list can be checked.*

Next, consider the *flag* argument. The intention is for 1 to mean "create" and 0 to mean "delete", but the code makes any non-zero value mean "create". There is little connection between 1 and creation, and 0 and deletion. So psychologically, the programmer may not remember which number to use. This can cause a queue to be destroyed when it should have been created, and *vice versa*.

Lesson 2. *Choose meaningful values for the parameters*

The third set of problems arises from a failure to check parameters. Suppose *qptr* is a nil pointer (**NULL**) or an invalid pointer when a queue is being created. Then the first *malloc* will cause a crash. Similarly, if *size* is non-positive, when the queue is allocated (the second *malloc*), the result is unpredictable.

Now consider queue deletion. Suppose either *qptr* or **qptr* is **NULL**. Then the result of the function *free* is undefined and may cause a crash.

Lesson 3. *Check the sanity of the parameters.*

More generally, the pointer parameter poses problems because of the semantics of C. C allows its value to be checked for **NULL**, but not for a meaningful non-**NULL** value. Thus, sanity checking pointers in C is in general not possible.

Lesson 4. *Using pointers in parameter lists leads to errors.*

The function does not check sequences of invocations. Consider:

```
qmanage(&qptr, 1, 100);
  /* . . . */
qmanage(&qptr, 0, 1);
  /* . . . */
qmanage(&qptr, 0, 1);
```

This deallocates the queue twice. The second deallocation calls *free* on previously deallocated memory, and the result is undefined (usually a crash).

Lesson 5. *Check that the function's operations are semantically meaningful.*

In the body of the function, failure of either memory allocation *malloc* call will cause references through nil pointers.

Lesson 6. *Check all return values, unless the value returned does not matter.*

Finally, consider the multiplication. If the system has 4 bytes per integer, and *size* is 2^{31} or more, on a 32-bit machine overflow will occur. Thus the amount of space may be much less than what the caller intended. This will probably cause a crash later on, in a seemingly random location.

Lesson 7. *Check for overflow and underflow when performing arithmetic operations*

We now apply these lessons to construct a robust data structure and queue management routine.

5 The Robust Example

Begin with the queue structure, which is at the heart of the library. That structure is to be unavailable to the caller, so we need to define two items: the structure itself, and its interface. We deal with the interface first. The object that the caller uses to represent a queue will be called a *token*.

If the token is a pointer to a structure, the user will be able to manipulate the data in the structure directly. So we need some other mechanism. Indexing into an array is the obvious alternative. However, simple indices enable the caller to refer to queue 0, and have a high degree of certainty of getting a valid queue. So instead we use a function of the index such that 0 is not in the range of the function. Thus, we will represent the queue as an entry in an array of queues. The token will be the output of an invertible mathematical function of this index.

Also, the code must never reference a queue after that queue is deleted. Suppose a programmer uses the library to create queue A. He subsequently deletes queue A and creates queue B, which has the same index in the array of queues as queue A did. If the token is a function of the index only, a subsequent reference to queue A will refer to queue B. To avoid this problem, each queue is assigned a nonce that is merged into the token. For example, suppose queue A has nonce 124 and queue B has nonce 3086, and both have index 7. The token for queue A is $f(7, 124)$ and the token for queue B is $f(7, 3085)$. As these values differ, the token will refer to queue A and be rejected.

This simplifies the interface considerably. The type **QTOKEN** consists of the value of the function that combines index and nonce. We must be able to derive the index and the nonce from this token; a moment's thought will suggest several ways to create such a function. Then, rather than use pointers, the value of the token be the represents the queue. The caller cannot access the internal queue representation directly; it can only supply the token, which the library then maps into the corresponding index.

This applies lesson 4. Because we designed the **QTOKEN**s so their values could be easily sanity checked, this also follows lesson 3.

The queue structure and storage then becomes:

```
typedef struct queue {
  QTICKET ticket; /* contains unique queue ID */
  int que[MAXELT];  /* the actual queue */
  int head;      /* head index in que of the queue */
  int count;       /* number of elements in queue */
} QUEUE;

static QUEUE *queues[MAXQ];   /* the array of queues */
static unsigned int noncectr = NOFFSET;/* current nonce */
```

For simplicity, all queues are of fixed size. All global variables are declared static so they are not accessible to any functions outside the library file. An empty queue has its count eld set to 0 (so the queue exists but contains no elements); a nonexistent queue has the relevant element in the array *queues* set to *NULL* (so the queue does not exist).

We modularize the functions. We define two new functions. *qgentok* generates a token from an index. *qreadtok* validates a given token and, if valid, returns the corresponding index. This enables us to make changes to the mapping between the token and the index in *queues*.

The queue creation and deletion operations are decoupled into two separate functions. Due to space limitations, we examine only the queue creation function:

```
QTOKEN qcreate(void)
{
  register int cur;   /* index of current queue */
  register QTOKEN tkt;/* new ticket for current queue */

  /* check for array full */
  for(cur = 0; cur < MAXQ; cur++)
    if (queues[cur] == NULL)
      break;
  if (cur == MAXQ){
    ERRBUF2("create_queue: too many queues (max \%d)", MAXQ);
    return(QE_TOOMANYQS);
  }
  /* allocate a new queue */
  if ((queues[cur] = malloc(sizeof(QUEUE))) == NULL){
    ERRBUF("create_queue: malloc: no more memory");
```

```
    return(QE_NOROOM);
  }
  /* generate ticket */
  if (QE_ISERROR(tkt = qgentok(cur))){
    /* error in ticket generation -- clean up and return */
    (void) free(queues[cur]);
    queues[cur] = NULL;
    return(tkt);
  }
  /* now initialize queue entry */
  queues[cur]->head = queues[cur]->count = 0;
  queues[cur]->ticket = tkt;
  return(tkt);
}
```

The differences between this routine and the non-robust version are instructive. Creating a queue requires no information beyond the invocation. So if one wanted to allow the user to specify the size of the queue, that size could be passed as the single parameter. Therefore, the parameters cannot be confused with one another—there are no parameters, or (if modified as suggested above) exactly 1 parameter. This applies lessons 1 and 2.

Next, the return values of all functions are checked. If *malloc* fails, an error code and an expository message are returned. This should never happen; the amount of space requested is a small constant, and with virtual memory, it would be very rare for that allocation to fail. Nevertheless, we check for failure (thereby checking for "impossible" cases). Similarly, if a token cannot be generated, an error is returned. This follows the lesson of checking *all* function calls—our own as well as library and system calls. This applies lesson 6.

The routine provides two types of error indicators. The return value is an integer outside the range of the function used to generate the token (so it cannot be confused with a valid token). A header file supplies a macro, **Q_ISERROR**, that takes an integer and returns 1 if the value represents an error, and 0 if it represents a token. In addition, a special error buffer contains a string describing the problem and any limits that were exceeded. A good example of this is in the body of the first *if* statement in the above function. If there were no available space in the array, the queue cannot be allocated. So the routine provides the caller an error indicator, and **ERRBUF2** loads into the error buffer a message giving the maximum number of queues that can be created. This way, the programmer knows the maximum number of queues the library can create.

Lesson 8*. Provide meaningful and useful error indicators and messages.*

6 Conclusion

Teaching robust programming is implicit in every beginning programming course. Unfortunately, as students advance through other computer science courses, they

often do not use the techniques for writing robust programs. Changing this situation requires having the students apply the techniques they have learned, and are learning, rather than treating the subject as an abstract exercise in analysis.

A similar statement holds for "secure programming". A focus on security, though, is misplaced—while it is a critical element of software, its exact definition varies from system to system and site to site. Programming robustly provides the basis for adding security elements to the program; but without robust programming, secure programming will never achieve the desired effect.

The above example was constructed many years ago to illustrate the problems of non-robust code, and how library routines written robustly overcome the problems. When teaching this lesson, having the students find the problems in the non-robust library challenges them to think of what can go wrong—in computer security terms, to think like an attacker (except that the "attacker" may not be malicious). This is the key to robust programming, and indeed all code reviews—a mode of thought in which problems are anticipated by examining the structure of the code and asking, "What if . . . ?"

References

1. Infusion pump improvement initiative. Tech. rep., Center for Devices and Radiological Health, U. S. Food and Drug Administration (Apr 2010), http://www.fda.gov/downloads/MedicalDevices/ProductsandMedicalProcedures/-/parGeneralHospitalDevicesandSupplies/InfusionPumps/UCM206189.pdf
2. Bilton, N.: Bug causes iphone alarm to greet new year with silence (Jan 2, 2011), http://www.nytimes.com/2011/01/03/technology/03iphone.html
3. Bishop, M.: Computer Security: Art and Science. Addison-Wesley, Boston, MA (Dec 2002), http://www.amazon.com/gp/product/0201440997
4. Bishop, M.: Some 'secure programming' exercises for an introductory programming class. In: Proceedings of the Seventh World Conference on Information Security Education (July 2009)
5. Bishop, M., Frincke, D.: Teaching secure programming. IEEE Security & Privacy 3(5), 54–56 (Sep 2005)
6. Bishop, M., Orvis, B.J.: A clinic to teach good programming practices. In: Proceedings of the Tenth Colloquium on Information Systems Security Education. pp. 168–174 (June 2006)
7. Johnson, R.: More details on today's outage (Sep 2010), http://www.facebook.com/note.php?note_id=431441338919&id=9445547199&ref=mf
8. Kernighan, B.W., Pike, R.: The Practice of Programming. Addison-Wesley Professional, Boston, MA, USA (Feb 1999)
9. Kernighan, B.W., Plauger, P.J.: The Elements of Programming Style. Computing McGraw-Hill, second edn. (1978)
10. Ledgard, H.F.: Programming Proverbs. Hayden Book Co. (1975)
11. Maguire, S.: Writing Solid Code. Microsoft Programming Series, Microsoft Press, Redmond, WA (Jan 1993), http://www.amazon.com/dp/1556155514
12. Seacord, R.C.: Secure Coding in C and C++. Addison-Wesley Professional, Upper Saddle River, NJ, USA (Sep 2005), http://www.amazon.com/dp/0321335724
13. Zetter, K.: Serious error in Diebold voting software caused lost ballots in California county—Update (Dec 8, 2008), http://www.wired.com/threatlevel/2008/12/unique-election/