# List of Common Bugs and Programming Practices to avoid them

Vipindeep V, Pankaj Jalote
Indian Institute of Technology, Kanpur
{vvdeep, jalote}@cse.iitk.ac.in

March 30, 2005

**Abstract**

Software bugs are common in practice. Although there are large variety of bugs possible, some bugs occur more commonly and are frequently the cause of software failures. In this report, we describe some of the common programming errors with suitable examples for them. The aim of this compilation is to make programmers aware of these errors, so they can avoid them. This is an initial compilation which, with feedback [1] , we hope to enhance in future and make it more comprehensive. The report also lists some poor programming practices that are often the cause of many types of bugs.

## 1 Introduction

A software bug can be defined as that part of the code which would result in an error, fault or malfunctioning of the program. Some bugs can be detected easily during development. But some bugs will be found late in the development process. These are low probability errors which are hard to detect and occur on very sparse set of inputs. According to IEEE standards, a 'bug' is an incorrect step, instruction or data in a program. A failure is caused because of a bug and may alter the external behavior of the program.

There have been many attempts to classify the bugs. Most of these represent the general classification which is valid in the cycle of software development and evolution. The classification scheme given in Orthogonal Defect Classification [9] defines eight categories of defects as - assignment, checking, algorithm, interface, timing/serial, function documentation, and build/pack/merge. Boris Beizer [5] has given an extensive bug taxonomy which classifies bugs based on the possible places in various phases of development cycle. Software bugs cause an increase in the cost of development both in terms of time and revenue. The basic challenge which a programmer faces is to minimize the introduction of defects , detect and remove existing bugs early in the development process. The defects detected early will cause much lesser damage than those which are detected later in the usage. In this report we focus on commonly occurring defects and their impact with simple classification.

The seriousness of a bug can be ascertained based on the damage caused to the functioning system. A general classification of bugs can be done based on the *frequency of the occurrence* of that bug and *severity* of that bug.

The severity of bug is a subjective measure depending on the software and the system. However we can broadly have following classes for severity:

- *Catastrophic*: Defects that can cause very serious effects (system may lose functioning, security concerns etc.)
- *Major*: Defects that could cause serious consequences for the system like losing some important data.

[1]Suggestions to enhance the list of common bugs and programming practices can be sent to *vipindeep.v@gmail.com*. Suggestions included will be suitably credited.

- *Minor*: Defects that can cause small or negligible consequences for the system. Ex. Displaying result in some different format.
- *No Effect*: These may not necessarily hamper the system performance, but they may give slightly different interpretation and generally avoidable. Ex. simple typographic errors in documentation.

**Goal of this TR:**
In this TR we compile a list of bugs commonly found in programs. The compilation is based on various published articles on the topic and users experience in programming. The aim is to educate the programmers about frequent mistakes, so that they can avoid them. In addition, we also describe some programming practices - both do's and dont's, that can in general help to reduce the incidence of bugs.

## 2 Common Bugs

This section lists common bugs which occur during coding which directly or indirectly manifest themselves to a larger damage to the running program.

### 2.1 Bugs with Pointers and Memory

The following are various possible bugs in dealing with pointers and memory.

#### 2.1.1 Memory leaks

*(frequent, catastrophic)* [6]
Memory leak is a common programming bug which occurs frequently in the languages which don't have automatic garbage collection (like C, C++). A memory leak is a situation, where the memory is allocated to the program which is not freed subsequently. This kind of situation can cause ever increasing usage of memory and hence at some point of time, the program may have to come to an exceptional halt because of the lack of free memory [22] [4] . An example of this error is:

```
char* foo(int s)
{
        char *output;
        if (s>0)
          output=(char*) malloc (size);
        if (s==1)
          return NULL;
        return(output);
        /* if s>0 and s==1 then,
         * allocated memory for output
         * is leaked */
}
```

#### 2.1.2 Temporary values returned

*(rare, catastrophic)*
The variables declared in a function represents some data which is private to that function. If the data is not dynamically allocated, it will be stored in the stack otherwise it will be stored in the heap. If we return the statically allocated variables address, then we are giving access to programs stack. If some data needs to be shared, then declare the variable as public and use that to access the common data.

```
char * foo(){
        char ch;
        //some operations
        return(&ch);
        /* local variable from the stack is returned*/
}
```

### 2.1.3 Free the already freed resource

*(frequent, major) [26]*
This a common form of error where the programmer tries to free the already freed resource. In general the resources are first allocated and then freed. For example, memory is first allocated and then deallocated. So we should not try to free the already freed resource.

```
main(){
        char *str;
        str=(char *)malloc(10);
        if (global==0)
                free(str);
        //Some statements
        free(str); /* Here str is already freed
                    * on true branch */
}
```

This may be more erroneous, if we have some malloc statement between the two free statements. There is a chance that the first freed locations are now allocated to the new variable. And the subsequent free will deallocate the new variable. Hence dereferencing it may cause runtime error.

### 2.1.4 NULL dereferencing

*(frequent, catastrophic)* [13]
Improper initialization and missing the initialization in different paths leads to the NULL reference error. This can also be caused because of aliases (two variables refer to the same object, and one is freed and an attempt is made to dereference the second variable).
To dereference a memory location, it should be initialized first and then dereferenced. The following code pattern illustrates NULL dereference bug.

```
char *ch=NULL;
if (x>0){
        ch='c';
}
printf("\%c", *ch);
// ch may be NULL
```

Whenever an object is being dereferenced, take care to see that it has been initialized in all possible paths to the point of dereferencing. Missing the initialization in any path may cause this error on some input value, which follows this control path.

### 2.1.5 Exposure of private data to untrusted components

*(rare, catastrophic) [26]*
Sometimes it is most important to preserve the integrity and security of data. The data which is supposed to be private should not be given access to external sources. The following example illustrates this fact.

```
struct node{
        char *ch;
};
char * foo(struct node nn){
        return (nn.ch);
}

main(){
        struct node n;char *ff;
        n.ch=(char *)malloc(1);
        *(n.ch)=100;
        ff=f(n);
        /*Here ff has an access to structure variable*/
}
```

Do not use simple structures when fair amount of security is needed. Or take care that internal data is not exposed to external resources.

## 2.2 Aliases

[16]

When there is unexpected aliasing between parameters, return values, and global variables, errors may be inevitable. Aliasing problems sometimes lead to deallocation errors. Static analysis of all feasible paths in the program can detect possible aliases.

### 2.2.1 Need of Unique addresses

*(frequent, major) [16]*

Aliasing creates many problems among them is violation of unique addresses when we expect different addresses. For example in the string concatenation function, we expect source and destination addresses to be different.

```
strcat(src,destn);
/* In above function, if src is aliased to destn,
 * then we may get a runtime error */
```

To avoid this, keep a check on the parameters before using them. This is more needed when the function can cause a dangerous side-effect. Be cautious when dealing with functions which expect parameters to be in certain format.

## 2.3 Synchronization Errors

[10] [25]

In a parallel program, where there are multiple threads which are accessing some common resources, there is a great chance of causing synchronization problems. There should be some means of controlling the execution of such concurrent threads and even more when there is a shared data. These errors are very difficult to find as they don't manifest easily, but are low probability events causing serious damages to system. This type of errors are generally discovered late in the development process. In general, there are three categories of synchronization errors and each of them may occur under different circumstances.

1. Deadlocks
2. Race conditions
3. Live lock

### 2.3.1 Deadlock

*(rare, catastrophic)*

It is a situation in which one or more threads mutually lock each other. The most frequent reason for the cause of deadlocks is inconsistent locking sequence. The threads in deadlock wait for resources which are in turn locked by some other thread. The general way to detect deadlocks is to construct a lock graph and analyze if it has a loop. Loop in the lock graph represents the presence of a deadlock. Deadlock detection and avoidance are other possible strategies followed in general by operating systems [14].

The following is a small JAVA code fragment which can give rise to a deadlock.

```
Thread 1:
        synchronized(A){
          synchronized(B){
                        }
        }
Thread 2:
        synchronized(B){
          synchronized(C){
                        }
        }
Thread 3:

        synchronized(C){
          synchronized(A){
                        }
        }
```

### 2.3.2 Race Condition

*(frequent, major)*

This is an error which results when two threads try to access the same resource and the result depends on the order of the execution of the threads. Consider the following example which illustrates this error.

```
class reservation
{
  int seats_remaining;
  public int reserve(int x)
  {
        if (x <= seats_remaining) {
          seats_remaining-=x;
          return 1;
         }
          return 0;
  }
}
```

If two threads simultaneously invoke reserve(x), then even though numbers of seats remaining are not enough for both of them , they may both be returned ‘1’. Here, the function reserve should be executed by one thread at a time and forms a critical section where the object of reservation is shared resource.

Whenever some shared data has to be accessed by threads, the conditions of critical section have to be satisfied (mutual exclusion, bounded waiting, progress)[25]. To ensure this, use the synchronization constructs like monitor, semaphores etc.

## 2.4 Some common bug patterns dealing with synchronization

### 2.4.1 Inconsistent synchronization

*(frequent, major)* [13] [19]

This bug indicates that the class contains a mix of locked and unlocked accesses. So if there are some locked accesses on shared variables and some other accesses are unlocked, then this may be an alarm for improper synchronization. Here we may give more importance to writes than reads of a shared variable. This is more often when we synchronize one of the methods in a class that is intended to be thread safe and forget some other.

### 2.4.2 Incorrect lazy initialization of static field

*(rare, major)* [13] [19]

A volatile static field has a set of 'release semantics' and 'acquire semantics' for reading and writing the data item. So if there is a lazy initialization of a non-volatile field shared by different threads, then it can possibly cause a synchronization problem. This is because, the compiler may reorder the execution steps and the threads are not guaranteed to see the expected initialization of the shared object which was non-volatile.

Hence such data items have to be declared as static volatile to rectify this problem. If we declare as static volatile, then the order of execution can be preserved by the semantics of the volatile field [23].

### 2.4.3 Naked notify in method

*(rare, minor)* [13] [10]

This bug is not necessarily an indication of error, but it can act as a possible alarm. A call to notify() or notifyAll() was made without any accompanying modification to mutable object state. A notify method is called on a monitor when some condition occurs, and there is another thread which is waiting for it has to be called. For the condition to be meaningful, it must involve a shared object between the threads which has been modified [13].

### 2.4.4 Method spins on field

*(rare, catastrophic)* [10]

This method spins in a loop which reads a field. The compiler may legally take out the read out of the loop, turning the code into an infinite loop. The class should be changed so that it uses proper synchronization (including wait and notify calls). [10] [13]

```
lock=0;
while(true)
{
        if (lock==0)
        /* if compiler moves this outside,
         * it causes an infinite loop*/
        break;
}
```

So when such waiting is needed, it is better to use proper synchronization constructs like wait and notify, instead of using primitive spin locks.

## 2.5 Data and Arithmetic errors

### 2.5.1 Uninitialised memory

*(rare, major)* [26]

Sometimes, it may be possible that a number of cases are included which initializes the data. But some cases might have been skipped because they were not expected. But if such cases arise, they impose problems like uninitialised memory.

```
switch( i )
{
       case 0: s=OBJECT_1; break;
       case 1: s=OBJECT_2;break;
}
return (s);
/* Object s is un-initialized for values other
 * than 0 or 1 */
```

A simplest solution to such errors is to have a safety initialization. This may be after the declaration. A more complex and a good strategy is to consider all the possible scenarios and paths and make suitable initialization whenever required in each path.

### 2.5.2 Value outside the domain

*(frequent, minor)* [26]

If we have initializations to variable with a value which is not inside its range, then it may give rise to unexpected results. This will be generally seen in type conversions.

```
int x,y
//some statements
if((p=x+y) < z)
       return (1);
else
       return (0);
```

Here if the value (x+y) is outside the range of an integer, it causes an overflow and sign bit is set to 1. Hence the 'if' condition evaluates to true and 1 is returned instead of 0. Some special values for floating point variables may give rise to exceptions (Ex. Not a Number) *Understand the range of variables used in the expressions.*

### 2.5.3 Buffer overflow/underflow

*(frequent, catastrophic)* [11] [7]

Buffer Overflow is an anamolous situation where the data is written beyond the acceptable limit of the given storage (buffer). In general, when an array is declared, we need to specify the size and any access on that array should be limited to its valid indices. So we are not supposed to write beyond this limit. Any code which puts data in some buffer without having any kind of checking for the size can cause a buffer overflow.

```
main(){
        char ch[10];
        gets(ch);
        /* can cause a buffer overflow */
        }
```

A simple buffer overflow is capable of putting the security of the system at risk. It can cause following vulnerabilities:

- Attacking Stack- Modify the return address to execute malicious code.
- Pointer overriding to get data on a specific location.

- It can cause an array index out of bounds.
- Heap overflow attack.

### 2.5.4 Arithmetic exceptions

*(frequent, major-minor)* [22]

Arithmetic exceptions are a class of errors. Some examples are: divide by zero, floating point exceptions etc. The result of these may vary from getting unexpected result to termination of the program.

### 2.5.5 Off by one

*(frequent, major)* [22]

This is a one of the most common error which can be caused in many ways. Starting a loop variable at 1 instead of starting at 0 or vice versa, writing <= N instead of < N or vice versa and so on are some examples of this error.

### 2.5.6 Enumerated data types

*(frequent, major)* [26]

Enough care should be taken while assuming the values of enumerated data types. This is more needed when the value of enumerated data type is used for indexing an array. These can be prevented by checking the values before performing the operation.

```
typedef enum {A, B,C, D} grade;
void foo(grade x){
      int l,m;
      l=GLOBAL_ARRAY[x-1];
      /* Underflow when A */
      m=GLOBAL_ARRAY[x+1];
      /* Overflow when D and size of array is 4 */
      }
```

### 2.5.7 Wrong assumptions on operator precedence

*(frequent, minor)* [26]

This is an error which is not directly visible and the results are not as expected. This is more dependent on the programmer's logic in coding. Learn the precedence rules of the programming language. Do not write big and complicated expression. Use parenthesis to avoid the confusions.

### 2.5.8 Undefined order of side effects

In a single expression itself, we may not surely guess the order of the side effects [22]. As in the following part of code, depending on the compiler used, I/++I might be either 0 or 1.

```
int foo(int n) {printf("Foo got %d\n", n); return(0);}
int bar(int n) {printf("Bar got %d\n", n); return(0);}
int main(int argc, char *argv[])
{
       int m = 0;
       int (*(fun_array[3]))();
       int i = 1;
       int ii = i/++i;
```

```
        printf("\ni/++i = %d, ",ii);
        fun_array[1] = foo; fun_array[2] = bar;
        (fun_array[++m])(++m);
}
```

The code prints either i/++i = 1 or i/++i=0; and either "Foo got 2", or "Bar got 2"

### 2.5.9 String handling errors

[20]

There are a number of ways in which string handling functions like strcpy, sprintf, gets etc can fail. Firstly, one of the operands may be NULL. Secondly they may not be NULL terminated resulting in unexpected results. The source operand may have greater size than the destination. This can create a buffer overflow which has many severe consequences .

## 2.6 Security vulnerabilities

### 2.6.1 Buffer overflow - Execute malicious code

Generally program code resides beside the data part of the memory. And hence buffer overflows are a commonly exploited computer security risk. By means of a buffer overflow condition the computer can be made to execute arbitrary and a potentially malicious code that is fed to the buggy program as an input data [24]. The following code fragment exploits buffer overflow to execute a malicious code:

```
char s1[1024];
void mygets(char *str){
      int ch;
      while(ch=getchar() !='\n' && ch!='\0')
              *(str++)=ch;
      *str='\0';
}
main(){
      char s2[4];
      mygets(s2);
}
```

Here there is a possible buffer overflow attack. If we have malicious code in s1 and if return address of mygets() is replaced by this address by overflowing the buffer s2, then we can have the code in s1 executed. A good practice to prevent this error is by checking the ranges of the indices before copying the data into buffers. There are techniques which can be employed to detect if the return addresses are modified like stackgurad [15], stackshield [1] etc. By using them, we can avoid such an attack.

### 2.6.2 Gaining root privileges

*(rare, catastrophic)* [8]

This is also not a functionality error but a security flaw which can be exploited by a threat. In order to perform some operations, an application might be using some privileged commands which are generally executed only in the root user mode. Hence the application is allowed to execute some commands being in privileged root mode. An ordinary user will not have root privileges. Even then, by exploiting the vulnerabilities present in the application and operating system, he can gain privileged control by getting access to the root.

For example, consider the case of Linux operating system. Every user and hence the processes created by him have a user id which will be used for the access control. The root has a user id of '0'. Each process possesses three user id's - real uid (which is the uid of the owner process), effective uid (which is the current uid of the process which will be used to get accesses over resources), saved uid( the uid which has to be restored back). The effective uid is the most important identifier which decides the access control for the current process. A process can drop its privileges by setting the effective uid. The root has all these values set to ‘0’.

Now, an application program like sendmail which is running on behalf of the user may need some privileged access like writing onto the user queue. For this, it has to run with root privileges even though it is executing on behalf of a user. In order to do this, a special bit (USERID bit) is set. When a user executes this process, it runs with privileges of the owner which set USERID bit. Hence the required process (sendmail) which needs to write onto user queue has this bit set by kernel and hence runs with root privileges. Now, the effective and saved uid of the send mail program is ‘0’ whereas the real uid will be that of user. So the sendmail program can do the privileged operations, and while exiting it sets back the uid’s back to user by making a call to setuid(getuid()). Hence after the necessary operations the user runs with his normal control.

But in Linux 2.2.16, the call setuid(getuid()) sets only effective userid. Hence saved uid will be that of root. Now a malicious user can exploit this fact and by using some vulnerability (like buffer overflow) of sendmail, he can execute setuid (-1,0) and gain the root permissions. Hence from being an ordinary user, he could gain root permissions. The details of setuid and this vulnerability is in [8].

This kind of attack is possible in any system using privileges. To perform some specific operations, we need to elevate some of the privileges for the user/application. This may be for short duration of time. Even then, it is prone to attack. One more important aspect to be taken care is regarding the set of privileges given to an application should follow ‘principle of least privilege’ which ensures that the application is given only necessary privileges and not more than that.

### 2.6.3 Exploiting Heap/BSS Overflows

*(rare, catastrophic)* [2]

A heap overflow is a genuine buffer overflow which writes beyond allocated storage in the heap. This can be used to execute some malicious code and do some illegal operations. The same attacks can also generated using stack based overflow techniques but now a days, stacks are being protected by several means, for example using Stack shield like tools or by making stack segments as non executable. So, heap based overflow attacks are more probable to work under this scenario.

The following code demonstrates dynamic overflow in heap. This code fragment is taken from [2].

```
#define BUFSIZE 16
#define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */

int main(){
      u_long diff;
      char *buf1 = (char *)malloc(BUFSIZE); *buf2 = (char
                      *)malloc(BUFSIZE);
      diff = (u_long)buf2 - (u_long)buf1;
      printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1,
              buf2, diff);
      memset(buf2, ‘A’, BUFSIZE-1), buf2[BUFSIZE-1] = ‘\0’;
      printf("before overflow: buf2 = %s\n", buf2);
      memset(buf1, ‘B’, (u_int)(diff + OVERSIZE));
      printf("after overflow: buf2 = %s\n", buf2);
      return 0;
 }
```

```
If we run this, we’ll get the following:
      [root /examples]# ./heap1 8
      buf1 = 0x804e000, buf2 = 0x804eff0, diff = 0xff0 bytes
      before overflow: buf2 = AAAAAAAAAAAAAAA
```

```
        after overflow: buf2 = BBBBBBBBAAAAAAA
```

This works because buf1 overruns its boundaries into buf2's heap space. But, because buf2's heap space is still valid (heap) memory, the program does't crash. This is the basis of almost all Heap-based overflow attacks.

Using this filenames, passwords, and a saved uids can be overwritten by exploiting this vulnerability in appropriate privileged programs. This can also be used to write a malicious code at specific locations of the heap. A function pointer allows a programmer to dynamically modify a function to be called. A function pointer can be overwritten by using either stack based overflow or heap based overflow vulnerability with the address of the malicious code in the heap. Now, when function pointer gets executed, it calls the function we point it to instead.

### 2.6.4 Dealing with volatile objects

*(rare, catastrophic)* [26]

When we are dealing with global volatile objects which are going to be shared between several threads, enough care should be taken to check that the intended operation is performed. If necessary we might have to add extra checking to ensure this. Consider the following example: A file is shared between the threads. Let us say that we performed enough access checks before gaining access to file and performed some reading. If at some later point, the child got executed first, and because of execution of some commands, files access structure have now been changed. Now the parent thread should not assume the prior access check. It has to again check the access conditions before reading/writing the file again. Otherwise there may be intermittent termination of the program if it accesses without an additional check.

```
FILE *fin;
int pid;
fin=fopen("inputfile", "rw");
//read some data from file
pid=fork(); /*create a new child process */
if (pid==0) {
/* child process may execute some commands
 * causing changes to file permissions
 */
}
else {
       wait();/*wait for child to complete */
       /* do some read operations on fin
        * Now permissions on file might have changed
        * So accessing again without checking permission
        * may cause an error */
       fin.close();
}
```

## 2.7 Redundant Operations

Redundancies prevail in a program because of too conservative programmer or carelessness which often results in some kind of hard error. Flagged redundancies can be used to predict non redundant errors [28].

### 2.7.1 Redundant Assignment

*(rare, minor)*

Redundant assignments which are never used are a result of defensive programming. These can be found through a simple intra procedural analysis. The following code fragment illustrates the error possible through an unexpected redundant assignment.

```
  X[i]= X[i]++;
Here the value of X[i] depends upon the way compiler evaluates.
This can be modeled as:
  X[i]=X[i];
  /* Redundant assignment resulting in a harder bug */
  X[i]++;
```

### 2.7.2 Dead Code

*(frequent, major)*
Dead code refers to that part of code which will be never executed. These reflect the false beliefs of the programmer that some code which is unreachable, gets executed. Mistaken statement terminators, single iteration loop (where next increment is never executed), unintentional fall through may result in common dead code problems. An example is shown below.

```
  for (i=0;i<10;i++)
  {
        if (i%3+i%2==0)
                break;
        else
                continue;
        x+=100; y--;
        /* This code is never executed as either of break
         * or continue is executed, skipping this step */
  }
```

### 2.7.3 Redundant Conditionals

*(rare, major-minor)*
In some cases, the presence of if, while, switch can't effect the flow of control in the program because of redundant conditional statements. So this kind of programming is likely to be a serious error. The type of bug can range from non sensical error (of type x=0 ; if (x==0) ) to programmers lack of understanding or missed conditions. The following fragment illustrates a case where the programmer has missed an error condition and placed a wrong one.

```
  if(x!=0)              { -------}
  else
        if (x!=1)               { ----- }
        else
                if (x!=0)               { ----- }
                        /* This may be x!=2 */
```

The other type of redundant operation involve Redundant NULL's (values returned from functions to be checked against NULLs etc), idempotent operations.

## 2.8 Inheritance related bugs

This section lists some common bugs caused due to false belief on the inheritance hierarchy.

### 2.8.1 Virtual function is not overridden

*(rare, major)* [10] [22]

If there is a virtual function in the base class with some prototype, and there is another function in the derived class with same name but with a mismatch in the prototype. Hence the virtual function of base class is not overridden. The programmer may be in a false belief that the derived class method is called when there is dynamic method dispatch.

```
class A{
        public virtual void foo() {
                            }
        }
class B extends A{
        public void foo(int x) {
        /* This might have been foo() */
                            }
        }
```

The same bug can be viewed in reverse direction as a derived class specific method is not defined. Hence the super class method is wrongly bound to the subclass. For example sub class level methods like findArea(), which will be present in a set of derived classes, may not be overridden with necessary derived class implementation resulting in the wrong logic in the program This type of bug can't be directly termed as a serious error. But it may manifest the error from design perspective.

### 2.8.2 Component of derived class shadows base class

*(rare, major)* [10]

If the derived class component has the same name as that of base class component, then there will be two objects with different references sharing the same name. The methods in base class and derived classes refer to the different components and hence subsequent derivations may tend to give unexpected results. Here derived class component shadows the base class component [22].

```
class A{
        int x;
        public void foo() {
        /* Here we have access to A.x */
        }
}
class B extends A {
        int x;
        public void foo()
        {
        /* Here we have access to B.x not A.x */
        }
}
class C extends B {
        /* This has access to B.x and A.x is lost */
        }
```

```
class A{
        int var=100;
        public static void foo(){
                int var=10;
        /*The class variable "var" is obscured here by
         *the local variable and hence the value of "var"
         *accessible here is 10 */
        }
}
```

### 2.8.3 Override both equals and Hashcode

*(frequent, minor)* [13]

When we implement a hash table, it is suggested to override both `public boolean Object.equals(Object x)`, and `public int Object.hashCode()`. If this is not done we might have left both not defined.

### 2.8.4 Some other flaws

There may be some design flaws in the code while dealing with inheritance. Some of these are:

- A subclass may be incorrectly located in the hierarchy.
- Deeply built inheritances may lead to confusion and may be error prone [18].
- Same operation may be inherited in multiple paths of inheritance which leads to confusion and even more when they are overridden in the paths.
- The invariants of base class may be violated by the computations of the derived class.

## 2.9 Some other common bug patterns include

### 2.9.1 Data can be lost as a result of truncation

*(frequent, minor)* [12]

This error is caused when significant bits can be lost as a result of conversion from large type to smaller. Such conversions have to be always explicitly specified by programmer.[1]

```
int x = (int)(y >>> 32); // no error
short s = (int)(x & 0xffbf00); // truncation
```

### 2.9.2 Compare strings as object references

*(rare, minor)* [13]

In JAVA, when string operands are compared by == or != operator. '==' returns true only if operands point to the same object, so it can return false for two strings with same contents. The following function will return false. So use of == instead of equals() can cause a serious error like this.

```
public boolean bug() {
        return
         (Integer.toString(100)==Integer.toString(100));
}
```

### 2.9.3 Array index values

Take care to see that the array index values are never negative. Keep possible checks before using index variable for arrays.

```
int len = -1;
char[] a = new char[len]; // negative array length
```

# 3 Some Programming Practices

## 3.1 Switch case with default

*(frequent, major)* [13] [26] [12]

If there is no default case in a 'switch' statement, the behavior can be unpredictable if that case arises at some point of time, which was not predictable at development stage. It is a good practice to include a default case.

```
switch (x){
      case 0 : {----}
      case 1 : {----}
      }
/* What happens if case 2 arises and there is a pointer
 * initialization to be made in the cases. In such a case,
 * we can end up with a NULL dereference */
```

Such a practice can result in a bug like NULL dereference, memory leak as well as other types of serious bugs. For example we assume that each condition initializes a pointer. But if default case is supposed to arise and if we don't initialize in this case, then there is every possibility of landing up with a null pointer exception. Hence it is suggested to use a default case statement, even though it may be trivial.

## 3.2 Empty Catch Block

*(frequent, minor)* [13]

An exception is caught, but if there is no action, it may represent a scenario where some of the operations to be done are swallowed. Whenever exceptions are caught, it is a good practice to do some default action. It may be as trivial as printing of error messages.

```
try {
              FileInputStream fis = new
      FileInputStream("InputFile");
}
catch (IOException ioe) {
      // not a good practice
}
```

## 3.3 Empty conditional statements

*(rare, major)* [26]

A statement is checked and nothing is done. This will happened due to some mistake and should be caught. Other similar errors include empty finally, try, synchronized, empty static method etc. It is better to avoid such useless checks unless it is implicitly doing some useful operation like spin lock.

```
if (x == 0) {
      /* nothing is done after checking x.
       * So what is need for this check?*/
}
```

## 3.4 For loop should be while loop

*(rare, minor)* [10]

Some for loops can be simplified to while loops which makes them concise and expressive. It is a good practice to use more expressive constructs which will be more intuitive and easy to understand. Consider the following example code:

```
for (;i<MAX;) {
       foo();
}
/* No initializations, increments.
 * So better to have a while loop */
```

In the above case, while loop is more expressive than the for loop.

## 3.5 Read return to be checked

*(rare, minor)* [13]

This bug indicates that the result from read's can be less than what is expected and hence that value should be read before accessing the read variable [1]. It is not mandatory practice, but it is suggested to check the consistency of the values returned by the functions before using them.

There may be some cases where neglecting this condition may result in some serious error. For example, read from scanf is more than expected, then it may cause a buffer overflow. Whenever a function returns a value, try to use it so as to handle alternate scenarios possibly giving exceptions.

## 3.6 Unnecessary temporary variable

*(frequent, minor)* [26]

Avoid unnecessary temporaries in a function. This is more devious if we consider string manipulations.

On the other hand, we may use more variables to capture specific operations done in the functions [17]. Also introducing new meaningful temporary variables may increase the readability of code.

## 3.7 Return From Finally Block

*(rare, minor)* [13] [26]

One should not return from finally block. In some cases it can create false beliefs.

```
public String foo() {
       try {
               throw new Exception( "An Exception" );
       }
       catch (Exception e) {
               throw e;
       }
       finally {
               return "Some value";
               /* May return on exceptions
                * and non exceptions also */
               }
       }
```

The value is returned both in exception and non exception scenarios. Hence at the caller site, the user might have been lead to a false belief. Another interesting case arises when we have a return from try block. In this case, if we assume that there is a return in finally also, then the value from finally is returned instead of the value from try.

## 3.8 Empty statement not in loop

*(rare, minor)* [28]

An empty statement in a program performs no useful operation. So this may be treated as an error. Further if it happens after control structures, it may even cause bad behavior of the program.

```
public void f(){

        // this is probably not what you meant to do
        for (i=0;i<10;i++); x++;
        // the extra semicolon here is not necessary
}
```

Such cases may not directly point us to an error, but check whether the intended logic is being done.

## 3.9 Closing the Streams

*(frequent, major)* [12]

In JAVA, there is automatic garbage collection and streams are supposed to be closed by the JVM. But the time when it may happen is not known. Input or output streams have to be closed whenever they are opened. Otherwise the resource may end up being blocked and other processes in need of it can't use that.

```
        FileOutputStream fout=new FileOutputStream("File.txt");
        //some operations
        fout.close();
        /* File stream should be closed so that other processes can
        now use it */
```

## 3.10 Loop condition confusion

*(rare, major)* [26]

It may happen that there are multiple conditions checked in the loop statement. The next statement after the loop may assume that one condition has become true while the other condition of the loop might have become true.

```
for (int i=0; i<MAX_SIZE && ARRAY[i] ; i++);

/* Here the loop terminates after counting MAX_SIZE or upon
 * reaching end of string */

strcpy (Destn, ARRAY);
/* Here if first condition is satisfied and  the above
 * statement assumes second condition to be satisfied,
 * then it may cause a buffer overflow */
```

A possible way to avoid such scenario is to be optimistic in assuming the result of the conditions. Check the possible cases when you are supposed to get out of loop, instead of assuming that once case has become true.

## 3.11 Correlated Parameters

*(rare, catastrophic)* [12]

The implicit correlation assumed between the parameters have to be validated before using the same. In the example below, even though we assume that 'length' represents size of BUFFER, we should check this. Otherwise we may run into a serious problem like buffer overflow which is illustrated by the following code fragment.

```
void (char * BUFFER, int length, char destn[]) {
      if (length < MAX_SIZE)
              strcpy (destn, BUFFER);
      /* This may cause buffer overflow if
       * length of BUFFER >MAX_SIZE > length.
       * Hence we should check the correlation also */
}
```

It is therefore suggested to do some counter checks on implicit assumptions. Like in the above case extra statements checking the consistency of the length of array 'BUFFER' with 'length' can be added.

## 3.12 Trusted data sources

*(frequent, catastrophic)* [26]

Counter checks have to be made before accessing or assuming the input data. For example, while doing the string copy operation, we should be assured that the source string is null terminated. Similar is the case with some network data which may be sniffed and prone to some modifications.

Consider following example in which the function getContext() is a remote call:

```
SECURITY_OBJECT getContext(TIME t){
      //Do some processing and return a security object
      return (object);
}

ACCESS_CODE read(){
      SECURITY_OBJECT s;
      s=getContext(time);
      //Now access secure data using structure
      code=s->security_code;
}
```

If the data is modified in the network, the last dereference ($s - > securitycode$) may cause Segmentation fault or give incorrect result. This is due to the possible change in the object s returned from network remote call.

To avoid such a devious error, we have to put some checks on the correctness of the incoming data. The simplest way we can assure this is some kind of parity check, computing hashes etc. Enough care should be taken when we assume the data coming from unreliable sources like external network.

For example consider the following code:

```
int remoteCall(char *str, int length){
        if (length!=0){

                strcpy(GLOBALDATA, str);
                /* If we assume that length is corresponding to
                * str which was supposed to be ensured by
                * interface, and if length!=0 str==NULL, it
                * causes an error */
```

```
            }
        }
```

## 3.13 Check for correct objects

When accessing the objects, we have to make sure that we are working with correct and expected objects. For example, while dealing with files, we have to ensure that the correct FILE object is being used. Naming conventions will be of great help here [12]. This is mostly a coding concern, but following common naming conventions really help in associating various parts of the code.

## 3.14 Concentrate on the Exceptions

Most programmers tend to give lesser attention to the possible exceptional cases as these could cause much damage to the application than some other possible bugs in the written code. There may be innumerable ways in which one's code may fail. So, it is the programmer's job to intuitively see all possible data paths in the program and write the suitable and necessary exception handlers [27]. When writing error handlers in the code, the following are some subtle guidelines which make the code efficient and secure [21].

1. Pay attention to genuine exceptional cases and weird ones with equal care.
2. Analyze the state of the system during exceptional handling and see that the system will be in allowed consistent state after the exception and handling.
3. Prepare test cases which fail the modules written by you. Hence it will give you possible exceptions to be dealt with.

Here is a bad code example. More on such errors can be found at [3].

```
bool accessGranted = true; // optimistic!
 try {
     // see if we have access to c:\test.txt
       new FileStream("c:\test.txt",
       FileMode.Open,
       FileAccess.Read).Close();
}
catch (SecurityException x) {
      // access denied
      accessGranted = false;
}
catch (...) {
      // something else happened
}
```

If access is granted based policies of CLR (common language runtime) and operating system, then Security Exception is not thrown. If for some case, the discretionary access control list on the file does't grant access then different type of exception will be thrown. But due to our optimistic assumption in the first line of code, this case is never known. A better way to write this code is to be pessimistic:

```
bool accessGranted = false; // pessimistic!
try {
      // see if we have access to c:\test.txt
      new FileStream("c:\test.txt",
      FileMode.Open,FileAccess.Read).Close();
```

```
        // if we're still here, we're good!
        accessGranted = true;
}
catch (...) {}
```

This is more secure than the previous case because of pessimistic assumption.

### 3.15 Use of finalize

Finazlizers incur an extra cost and do not perform predictably at all times. In some cases the use of these constructs is unavoidable. Here are some ways in which finalizer can be misused [13]. Empty finalizer

```
        protected void finalize() { }
```

The finalizers which call only call super.finalize(). They can prevent some optimizations that can be done at runtime

```
        protected void finalize() { super.finalize(); }
```

Finalizers that do not call super.finalize() will negate effect of superclass finalizer. In general the following format is suggested:

```
protected void finalize() { }
protected void finalize() {
        try {
                doSomeOperations();
        }
        finally {
                super.finalize();
        }
}
```

### 3.16 Use of simpler methods

Larger methods can be split into small, cohesive ones giving more intuitive idea of what is being done. This improves readability of the code increasing reusability at the same time.

Building complex logic in a single method should be prevented. Otherwise, they include large amount of logic, making them too specialized to perform some operations. Hence these will not be good candidates for reuse.

### 3.17 Use of complicated expressions

It is a good practice to represent the expressions in a simple and proper way. Mixing up various operations in single expressions complicates the scenario and may lead to unnecessary confusions. Further, it may lead to harder debugging when required [17]. Consider the following statement

```
int c = 10 * a / b++ ;
int z = 100 + (10 * x / y) ;
if ( c >= x)
        // some operation
else
        //some other operation
```

This is a better written code. Here if there was a problem with division in assignment to variable ‘c’ which we might have easily captured. But consider the following code where, the expression is written in a complicated way mixing up the individual operations.

```
if ( (10 * a / b++) >= (100 + (10 * x / y)) )
      //some operation
else
      //some other operation
```

In the above code, it would be harder to interpret and debug when it is necessary.

## 3.18 Cryptography

It is very bad practice to store the secret data in the programs itself. It should be also avoided to used ones own cryptography algorithm. The simplest way to get the stored secret data is by guessing the variable names which in case of passwords may be ‘pwd’, ‘Password’, ‘crypt’, ‘salt’ and so on. So the attacker can first look for variable names and functions with such names. All such hits, even though they may give false positives can be further analyzed among which some may be interesting and yield embedded secret data for cryptography systems. When searching for cryptographic algorithms, the simplest possible way to do is to search for XOR operations which will be usually there in crypto systems. So one should be careful in designing such systems [3].

## 3.19 Miscellaneous practices

[22]

- Cyclomatic complexity exceeds specified limit. This makes analysis difficult.
- A class should have no public fields except ‘final’ or ‘static final’.
- Synchronize at the block level rather than the method level.
- Check to see if the for statement has an update clause. Otherwise, it may tend to give to some infinite loop. If there is no updates, then better to use while loop.
- Too many parameters make the function complex.
- Avoid expressions like ? true : false
- Limit the amount of logical nesting.
- It is good practice to call in any case super() in a constructor.
- Make inner classes ‘private’.

# 4 Conclusions

This report tried to illustrate many common bugs which are generally committed by the programmers.We hope that awareness about these will help programmers prevent such bugs. We have also discussed some programming practices that will help write good code.

## References

[1] Stackshield: A ”stack smashing” technique protection tool.

[2] Heap Overflow Attacks. Homepage : http://www.w00w00.org/files/articles/heaptut.txt.

[3] Michael Howard and Keith Brown. ”defend your code with top ten security tips every developer must know, homepage http://msdn.microsoft.com/msdnmag/issues/02/09/securitytips/”.

[4] IEEE Std 1044-1993. Ieee standard classification for software anomalies, institute of electrical and electronics engineers.

[5] Boris Beizer. *Software testing techniques (2nd ed).* Van Nostrand Reinhold Co, 1990.

[6] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[7] Cowan C., Wagle F., Calton Pu, Beattie S., and Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade, 2000.

[8] H. Chen, D. Wagner, and D. Dean. Setuid demystified, 2002.

[9] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification – A concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, November 1992.

[10] Cyrelli Artho. Jlint homepage: http://artho.com/jlint.

[11] D. Cok and J. Kiniry. Esc java homepage : http://www.cs.kun.nl/sos/research/escjava/index.html.

[12] D Ddyer. The Top 10 Ways to get screwed by the ”C” programming language http://www.andromeda.com/people/ddyer/topten.html.

[13] David Hovemeyer. Findbugs homepage: http://findbugs.sourceforge.net/.

[14] Dawson Engler. Racerx: Effective, static detection of race conditions and deadlocks.

[15] Crispan Cowan et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[16] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[17] Martin Fowler. *Refactoring- Improving the design of existing code.* Addison Wesley Publications, 2003.

[18] NASA group. Issues and Comments about Object Oriented Technology in Aviation .

[19] David Hovemeyer and William Pugh. Finding bugs is easy. In *In proceedings of OOPSLA 2004*, 2004.

[20] Michael Howard and David LeBlanc. *Writing Secure Code (2nd ed.).* Microsoft Press, 2002.

[21] MSDN. Expert Tips for Finding Security Defects in Your Code Homepage: http://msdn.microsoft.com/msdnmag/issues/03/11/SecurityCodeReview/.

[22] Anonymous programmers and researchers.

[23] Various Researchers. MSDN library.

[24] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector, 2004.

[25] Silberschatz, Galvin, and Greg Gagne. *Operating System Concepts, 6th edition.* John Wiley Sons Inc., 2001.

[26] Microsoft Testing Unit. Common coding errors.

[27] unknown. MSDN,Exception handling Homepage: http://msdn.microsoft.com/library/default.asp?url=/library/e
us/dv_vstechart/html/exceptions2.asp.

[28] Yichen Xie and Dawson Engler. Using redundancies to find errors. *SIGSOFT Softw. Eng. Notes*, 27(6):51–60, 2002.

## 5 APPENDIX

The following tabular will illustrate the relationship between the bad programming practices and the real bugs. Each bug can be associated with a programming practice, which may not be necessarily among the listed. And each bad practice can cause and error which may be not necessarily in enlisted. Here we tried to capture some common mapping. In the table below, OTH refers to other possible bugs/practices which are not listed here.

| **ID** | **Coding Error** | | **(Bad) Coding Practice** | |
|---|---|---|---|---|
| 1 | MEM LEAKS | 1,8,9,13, 15,OTH | SWITCH NO DEFAULT | 1,4,12, 14,15,21 |
| 2 | TEMP RETURN | OTH | EMPTY CATCH | 12,15,22, 31,OTH |
| 3 | FREED FREE | 9,OTH | EMPTY CONDITIONAL | 3,OTH |
| 4 | NULL DEREF | 1,5,9,10, 11,13,15, OTH | FOR SHOLD BE WHILE | OTH |
| 5 | EXPOSE TO UNTRUSTED | OTH | CHECK READ RETURN | 4,12,14,15, 17,21,24 |
| 6 | ALIASING ERRORS | 1,5,8, 11,OTH | UNNCECESSARY TEMPS | OTH |
| 7 | DEADLOCKS | OTH | EMPTY LOOPS | OTH |
| 8 | RACES | OTH | CLOSING STREAMS | 1,4, 6,OTH |
| 9 | INCONSISTENT SYN | OTH | LOOP CONDITION CONFUSED | 3,4,8,14, 21,26,OTH |
| 10 | NAKED NOTIFY | OTH | CORRELATED PARAM | 4,14,15,21, 13,24,OTH |
| 11 | SPINNING ON FIELD | OTH | TRUSTED SOURCES | 4,12, 21,22 |
| 12 | UNINITIALISED USE | 1,2,5, 7,8,13, 15,OTH | CHECK OBJECTS | OTH |
| 13 | RANGE INCONSISTENCY | 1,OTH | DEAL ALL EXCEPTION CASES | 4,6,8, 12,14,15, 21,25,OTH |
| 14 | BUFEFR OVERFLOW | 1,2,9,10, 11,13,OTH | USE FINALIZE | OTH |
| 15 | ARITHMATIC EXCEPTIONS | 1,9,10, OTH | SIMPLER METHODS | OTH |
| 16 | OFF BY ONE | OTH | COMPLECATED EXCEPTIONS | OTH |
| 17 | ENUMERATED TYPE ERRORS | OTH | CTRYPTOGRAPHY | OTH |

| | **Coding Error** | | **(Bad) Coding Practice** | |
|---|---|---|---|---|
| 18 | OPERATOR PRECIDENCE ERRORS | OTH | | |
| 19 | SIDE EFFECT MISCONCEPTION | OTH | | |
| 20 | USE OF & | OTH | | |
| 21 | STRING ERRORS | 5,8,9, 10,11,OTH | | |
| 22 | MALICIOUS CODE EXECUTION | 2,11,OTH | | |
| 23 | GAIN PREVILEGES | OTH | | |
| 24 | HEAP OVERFLOWS | OTH | | |
| 25 | DEALING VOLATILE OBJECTS | 13,OTH | | |
| 26 | REDUNDANT OPERATIONS | OTH | | |
| 27 | INHERITENCE RELATED ERRORS | OTH | | |
| 28 | USELESS FLOW | OTH | | |
| 29 | UN CLOSED STREAMS | 8, OTH | | |