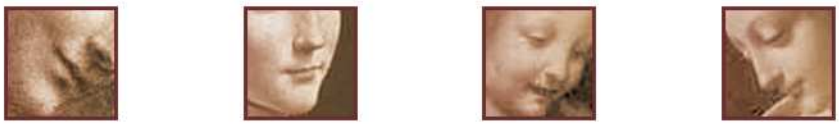

CORSAIRE WHITE PAPER

BREAKING THE BANK

VULNERABILITIES IN NUMERIC PROCESSING WITHIN FINANCIAL APPLICATIONS

Project Reference	080715 Corsaire Whitepaper - Breaking the Bank - Numeric Processing.doc
Authors	Adam Boulton, Stephen De Vries, Kevin O'Reilly
Date	15 July 2008
Distribution	General release

Copyright © 2008 Corsaire Limited. All Rights Reserved.

Breaking the Bank

Executive Summary

Attackers will go to extraordinary lengths to compromise a financial application; the spoils are just too irresistible. So unsurprisingly these applications have higher requirements for data confidentiality, transaction integrity and service availability than many other web applications.

Brand damage, loss of client and corporate data, fraudulent transactions and loss of revenue are just some of the direct local impacts associated a security breach of a financial application.

"Security compromise of a financial application has far-reaching and serious implications for the business"

This is, however, well trodden ground in the security arena and the purpose of this paper is not to play on the fears associated with the threats facing financial applications.

Clearly, a compromise of the integrity of financial data can have severe repercussions. Aside from the direct impact from deliberate fraud, organisations can also find themselves subject to fines and sanctions issued by bodies such as the UK's Financial Services Authority (FSA) and the Gambling Commission, or Australia's Australian Prudential Regulation Authority (APRA); along with potential prison sentences associated with breaches of Sarbanes Oxley.

"The accuracy and integrity of numeric calculations is of the utmost importance for applications dealing with financial data, but is frequently overlooked during security assessments"

The inappropriate handling of numeric calculations may be for many reasons; some assessment providers are simply unaware of the intrinsic programmatic risks associated with numerical processing, others are focussed on more easily identifiable issues associated with applications in general. The continued observation of such flaws by Corsaire's consultants suggests that there is still a lack of awareness of these issues, and they remain misunderstood and overlooked both from development and security assessment perspectives.

There are, of course, many other security threats facing financial applications including areas such as strong authentication, data validation and accountability, for example.

This paper focuses on technical issues associated with common programming languages and API's that present a security threat, and how to mitigate the associated risks.

While banking, trading, e-commerce and electronic gaming applications are likely to be some of the most effected by such flaws, these issues are applicable to any application where critical numeric calculations are made and relied upon.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Breaking the Bank

Table of Contents

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Breaking the Bank

Overview

This paper draws attention to how the use of common programming APIs and practices could lead to flaws in the processing of numeric data, which could allow attackers to manipulate the outcome of transactions or otherwise interfere with the accuracy of calculations.

It discusses the technical vulnerabilities typically observed in both the validation and processing of numeric data that could expose an organisation to unmanaged risk. It is intended for a technically literate audience involved in developing or testing financial applications, and to provide technical insight to those responsible for their management. The vulnerabilities are presented with source code examples, suggestions on how to identify the flaws during the testing phases and recommendations for mitigating the risk.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Breaking the Bank

Technical Vulnerabilities

Introduction

The technical vulnerabilities discussed within this section of the document are based on real world scenarios observed by Corsaire's consultants. These vulnerabilities can lead to significant discrepancies relating to the processing and delivery of financial data, which can have a major impact on the organisation.

All Java code examples were compiled using the Sun JDK 6. The .NET code examples were compiled under version 3.0 of the .NET framework. Additional examples are provided in VBScript, where appropriate.

Use of Inappropriate Data Types

The float and double data types are based on the Standard for Binary Floating-Point Arithmetic (IEEE 754)/Binary floating-point arithmetic for microprocessor systems (IEC 60559:1989). This standard has acknowledged issues and short-comings relating to rounding errors, comparisons and the overall accuracy of results when performing floating point arithmetic. Floats and doubles are designed for scientific and engineering calculations, where some small loss of accuracy is acceptable. Their use in financial applications should be avoided as they result in approximations instead of exact results.

Java

For example, consider the following Java code:

```
System.out.println(2.00 - 1.10);
```

While it would be reasonable to expect the output to be "0.9", it is actually "0.8999999999999999". It is not possible to represent fractions exactly, as the IEEE 754 standard explains; floats and doubles are stored internally as 32 and 64-bit numbers, with inherent limits in the accuracies they can store. For example, it is impossible to represent any negative power of 10 as a float or double exactly, illustrated here when performing a calculation with 0.1:

```
double fraction = 0;

for (int i=0; i<10; i++)
{
   fraction += 0.1;
   System.out.println(fraction);
}
```

Produces:

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
```


Copyright © 2008 Corsaire Limited.
All Rights Reserved.

```
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

Rounding will continue to cause further inaccuracies as can be seen in the following example:

```
double d = 29.0 * 0.01;
System.out.println(d);
System.out.println(d * 100); //29.0 would be expected
System.out.println((int) (d * 100)); //Cast to integer
```

This produces the output:

```
0.29
28.999999999999996
28
```

In the cast operation, the mantissa is simply stripped without consideration for the overall result.

Clearly, various mathematical operations conducted on the floats and doubles resulted in notable variations and increasingly significant errors that would affect the integrity of a financial application.

C#

In C# floating points use 128 bits to represent values within the range 1E-28 to 7.9E+28. As can be seen throughout the paper, floating point arithmetic suffers from serious rounding errors when dealing with decimal values.

A similar effect as the above Java example can be observed via the following C# code:

```
int i;
float fraction = 10000;

for (i = 0; i < 101; i++)
{
    fraction = fraction+0.1f;
    Console.WriteLine(fraction.ToString("N2"));
}
```

This produces the following (truncated) output:

```
10,000.10
10,000.20
10,000.30
10,000.40
...
10,009.76
10,009.86
10,009.96
10,010.06
```

After one hundred iterations, a significant error can be seen.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

VBScript

In VBScript, this situation (which is inherent to all floating point numbers) can be masked by a feature of the VBScript engine:

```
print(919.9999999999999);
print(920.0000000000001);
```

Produces the output:

```
920
920
```

Here the VBScript interpreter assumes that the input values have already suffered from floating point rounding errors, and rounds them in an attempt to correct this phenomenon. However, as we can see this correction is performed even when no error has actually occurred – this could lead to erroneous results.

Testing

If a site permits a user the ability to manipulate decimal values then it can be tested by performing a calculation which would result in a rounding error, for example, in a banking application perform the following tests:

1. Modify an account so that it only has 2.00 as the available balance.

2. Transfer 1.10 from the account.

If the balance is 0.8999999999999999 then the application is prone to serious miscalculations and is a likely indication that the IEEE 754 standard has been used for monetary calculations.

Recommendation

When dealing with monetary values in Java and .NET languages, it is recommended that the available classes that allow arbitrary precision, and their associated methods, be used for calculations instead of the standard mathematical operators. For Java, the java.math.BigDecimal class is available, while the System.Decimal class is available for C#.. However, these classes also have their peculiarities which should be taken into account (see *Converting between Floating Points* and *Object Equality*).

A combination of data types may be required in a financial application, for example, if a language supports primitives then their use may be considered when optimal performance is essential and the use of the BigDecimal class to avoid the overhead of manually managing the decimal point. In either solution, thorough testing should be performed to ensure that calculations provide the correct and expected result.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Converting between Floating Points

Java

There are also a number of pitfalls to be aware of when using floating points in combination with the BigDecimal class. BigDecimal has several factory methods and overloaded constructors which take floating-points as an argument; these could cause approximation errors. Depending on the value passed into the constructor or factory methods the result may not be as expected due to IEEE 754, as discussed above.

For example, consider the following Java code:

```
double d = 1;
for(int i=0; i<10; i++)
{
        d += 1.1;
        BigDecimal bd = new BigDecimal(d);
        System.out.println(bd);
}
```

Running the code results in the following output:

```
2.1
3.2
4.300000000000001
5.4
6.5
7.6
8.7
9.799999999999999
10.899999999999999
11.999999999999998
```

C#

The same is also true for C#: The Decimal class can suffer from similar inaccuracies when constructing a Decimal using floating point values:

```
float f = 1;

for (int i = 0; i < 10; i++)
{
    f += 1.1F;
    decimal dec = new decimal(f);
    Console.WriteLine(dec.ToString("N6"));
}
```

This gives as output:

```
2.100000
3.200000
4.300000
5.400000
6.500000
7.599999
8.700000
9.800000
10.900000
12.000000
```


Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Testing

During a security code review, check for the use of the BigDecimal or Decimal class which is constructed using floating point values and ensure adequate validation has been performed.

During a black box assessment, attempt to set a value to “Infinity” or “NaN”. While these are valid values for the Double class they will cause runtime exceptions (NumberFormatException in Java and FormatException in C#) if used to construct a BigDecimal / Decimal, which can therefore cause the currently executing thread to crash if not handled correctly.

Recommendation

A BigDecimal should be constructed using a String value and should be validated beforehand, as a String, in order to prevent runtime exceptions. Using a floating point to instantiate a BigDecimal should be avoided. Floating points also have a limit of 16 significant digits which results in size restrictions since BigDecimal supports more. One of the most important reasons for using BigDecimal is that it gives developers complete control over rounding and scale manipulation.

Care should be taken when using built in numeric validation functions as these might permit non-numeric data to be accepted when it should be rejected.

Bypassing Validation through Exponential Notation

Floating point values can often be expressed in exponential notation. If validation is performed on String values before they are converted to floating points, then exponential notation could be used to bypass the validation facilities.

The following VBScript code illustrates this vulnerability:

```
Dim value As String
Dim number As Double
value = "9E-4"
If (IsNumeric(value)) Then
   'Check if the number contains at most 2 decimal places
   If ((InStr(value, ".") = 0) Or (Len(value) - InStr(value, ".") <= 2)) Then
      number = CDbl(value)
      MsgBox("The value contains at most 2 decimal points.  Value=" & number)
   End If
End If
```

Produces the output:

```
The value contains at most 2 decimal points.  Value=0.0009
```

Exponential notation could also be used to bypass length restrictions, consider the following VBScript:

```
String value = new String("9E+6");
Float number;
number = Float.valueOf(value);
```


Copyright © 2008 Corsaire Limited.
All Rights Reserved.

```
if (value.length() <= 4) {
   System.out.println("Value is less than 4 digits long. Value="+number);
}
```

Produces the output:

```
Value is less than 4 digits long.  Value=9000000.0
```

Both Java and C# have methods to convert String values in exponential notation to the equivalent floating point values as shown in the VBScript example above.

Testing

Test the application for support of exponential notation. For example, by attempting to bypass limits using values such as:

- 9E+1
- 9E1
- 9E-1
- 0.99e-4
- 0.99E+6

Recommendation

Validate string values using strict rules to ensure that only digits, and if required, the decimal point is permitted. Verify that the validation was correctly performed by re-validating the numeric value after it is converted to a numeric data type.

Bypassing Validation through Reserved Words

There are a number of reserved String words which can be recognized as legitimate numeric values by a language. These values can be used to bypass validation and cause serious data corruption issues. In Java and C# the reserved values are:

- NaN
- Infinity
- -NaN
- -Infinity

Java

Consider the following Java code:

```
String value="NaN";
Float number, balance,result;
balance = 10.0F;
```


Copyright © 2008 Corsaire Limited.
All Rights Reserved.

```
try {
        number = Float.valueOf(value);
        result = balance - number;
        System.out.println(result+" is a valid number.");
} catch (NumberFormatException e) {
        System.out.println("The value: "+value+" is not a valid number.");
}
```

This produces the output:

```
NaN is a valid number.
```

"NaN" is a Java reserved String which stands for Not a Number, but is itself regarded as a valid number by the language. A NumberFormatException would be thrown for other strings which are not regarded as numbers.

C#

The following C# code illustrates the same principle:

```
String value = "-Infinity";
double number, balance;
balance = 10.0F;
try
{
   number = Convert.ToDouble(value);
   double result = balance - number;
   Console.WriteLine("Results=" + result);
   Console.ReadLine();
}
catch (FormatException fe)
{
   Console.WriteLine(value + " is not a number.");
}
```

As the code illustrates, these values can take part in numeric calculations, the results of which could skew financial calculations, for example:

- 10.0F – (-Infinity) = Infinity
- 10.0F / (-Infinity) = -0.0
- Any calculation involving "NaN" produces "NaN" as a result.

Testing

Test all of the listed reserved words in numeric fields. If an exception from a casting operation is the only validation performed on a value, then this validation could likely be bypassed and invalid numeric data entered.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Recommendation

Validate values as Strings before they are converted to numeric data types using a strict white list. Additionally, re-validate the numeric values after conversion and take care when using built in numeric validation functions.

Bypassing Validation through Overflows and Underflows

Numeric overflows occur when a value is too large for the number of bytes allocated for the type. For example, consider the primitive int data type. This is a 32-bit signed integer. An int therefore has a minimum value of -2^{31} and a maximum value of $2^{31}-1$. In Java and C#, if an integer value reaches the maximum value and is incremented, an overflow occurs which causes the value to 'roll-over' into the smallest minimum value; no runtime exceptions are thrown when the overflow occurs. The overflow condition exists for all the primitive data types and primitive wrapper classes in Java as well as the corresponding C# classes.

Java

The following Java code demonstrates a numeric overflow:

```
Integer intOverflow = Integer.MAX_VALUE + 1;
System.out.println("intOverflow = " + intOverflow);

Double positiveInfinity = Double.MAX_VALUE + Double.MAX_VALUE;
System.out.println("doubleOverflow = " + positiveInfinity);

Float negativeInfinity = -Float.MAX_VALUE / 0;
System.out.println("floatOverflow = " + negativeInfinity);
```

This produces the output:

```
intOverflow = -2147483648
doubleOverflow = Infinity
floatOverflow = -Infinity
```

C#

Similarly in C#:

```
Int32 intOverflow = Int32.MaxValue + 1;
Console.WriteLine("intOverflow = " + intOverflow);

Double positiveInfinity = Double.MaxValue + Double.MaxValue;
Console.WriteLine("doubleOverflow = " + positiveInfinity);

Float negativeInfinity = -Float.MaxValue / 0;
Console.WriteLine("floatOverflow = " + negativeInfinity);
```

This produces the output:

```
intOverflow = -2147483648
doubleOverflow = Infinity
floatOverflow = -Infinity
```


Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Testing

If a site permits the ability to manipulate numeric values, such as a shopping cart feature, it can be tested as follows:

1. Add 214748367 items in the shopping cart.

2. Then add one more of the same item to cause an overflow.

If the shopping cart registers as having "-2147483648" quantity it is possible to infer that the quantity is being stored using a 32-bit value of the Integer / Int32 data type and that no validation has been performed on the result.

Recommendation

If a language supports unsigned values (such as the UInt32 in C#) then these data types should be used when expecting only positive values, such as an online shopping cart.

Validation should be performed thoroughly on values prior to them being used in a calculation as well as the end result being validated to avoid results which may have overflow or underflow.

Bypassing Validation through API misuse

Applications sometimes perform validation on numeric data, before the data has been converted from a string data type. This can lead to unexpected errors, and in some cases allow attackers to bypass this validation.

VBScript: Implicit conversion

The VBScript "isDigit()" function expects a character argument and returns a Boolean based on whether the character is a digit or not. If the argument is a String, then it is cast as a character by taking the first character in the string. No warnings or error messages are generated during this cast. Inadvertent use of strings instead of characters could allow non-numeric values to bypass validation routines.

This means that the results of the following calls are all true:

```
Char.iSDigit("2E-4")
Char.isDigit("2hello")
```

Depending on how the validation is performed, this could lead to non-numeric data finding its way into the business logic, which could lead to unpredictable results and failure of the application or applications that depend on the data.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

VBScript: isNumeric

The “isNumeric()” functions is used to determine whether a given string value is numeric or not. But this function will return true for a wide range of characters which are not necessarily valid digits. This can be used to bypass some forms of validation, for example:

```
Dim value As String
Dim number As Double
value = "$-12"
If (IsNumeric(value)) Then
  number = CDbl(value)
  'Checks if the first character is a negative
  If InStr(value, "-") = 1 Then
      MsgBox("Negative.  Value=" & number)
  Else
      MsgBox("Not a negative.  Value=" & number)
   End If
End If
```

This produces the output:

```
Not a negative.  Value=-12
```

Testing

Insert a dollar sign in front of negative numeric values to attempt to bypass negative value checks. Where the number of decimal places is being restricted through validation, attempt to enter the values as exponential notation.

Recommendation

Avoid using the “isNumeric()” function to perform validation; instead, perform stronger and tighter validation through “isDigit()” and thorough String validation. When using “isDigit()”, ensure that the parameter is a char value, and not a String. In addition to String validation it is also recommended that numeric values are validated after they have been cast as numerics.

Object Equality

Using incorrect methods to determine value and object equality can result in inaccurate results and inconsistent business logic.

Java

Most classes have an overridden “equals()” method (derived from Object) which is used to compare values between objects. However, when dealing with monetary values and using the BigDecimal class the “compareTo()” method should be used. The following code illustrates the difference:

```
BigDecimal bigDec = new BigDecimal("100.0");
BigDecimal bigDec2 = new BigDecimal("100.00");
System.out.println(bigDec.equals(bigDec2));
System.out.println(bigDec.compareTo(bigDec2));
```


Copyright © 2008 Corsaire Limited.
All Rights Reserved.

The output shows that there is a very distinct difference:

```
false
0
```

The “0” returned by “compareTo()” indicates that the values are equal – even though they have a different scale they are equivalent in value. In order for the “equals” to return true the value and scale must be equal.

C#

The following C# code indicates how the “equals()”, “compare()” and “compareTo()” methods behave differently when compared to the equivalent Java implementation above. Functionally there is very little difference between these methods as the “compare()” and “compareTo()” methods are merely wrappers around the “equals()” method. All of these methods are used to compare the values between instances of Decimal objects. However, the scale is not taken into consideration when determining the equivalence:

```
Decimal bigDec = Decimal.Parse("100.0");
Decimal bigDec2 = Decimal.Parse("100.00");
Console.WriteLine(Decimal.Equals(bigDec, bigDec2));
Console.WriteLine(Decimal.Compare(bigDec,bigDec2));
Console.WriteLine(bigDec.CompareTo(bigDec2));
```

Results in the following output:

```
True
0
0
```

Caching in Java

Since Java 5, primitive wrapper class caching was introduced which may cause further mistakes when comparing values. The Integer, Byte, Short and Long classes contain nested inner classes which create 256 instances of the instantiated Object using values in the range of -128 to 127. This can cause confusion and lead to potential errors when using the “==” operator. While “==” is used to compare primitive values it should never be used when attempting to compare the values of the wrapper classes as the values are not unboxed and instead compare object equality. The following Java code demonstrates the caching mechanism:

```
Integer a = 5, b = 5;
Integer c = 200, d = 200;

System.out.println(a == b);
System.out.println(c == d);

System.out.println(a.equals(b));
System.out.println(c.equals(d));
```

Produces the output:

```
true
false
true
true
```


Copyright © 2008 Corsaire Limited.
All Rights Reserved.

The first condition returns “true” because using the “==” operator will always check Object equality and not value, unless comparing primitives. Object “a” and Object “b” will point to the same object on heap. However, for values which lie outside the caching range, a new object is created on the heap which causes the second condition to result in false despite the Integer Objects containing the same value.

BigDecimal also uses caching but creates the cache using a slightly different mechanism. Instead of using a nested inner class, the caching is pre-defined in a static array and only covers 11 numbers, 0 to 10.

Further issues are introduced when using BigDecimal in SortedMap or SortedSet Java classes. Since BigDecimal's natural ordering is inconsistent with “equals”, great care should be taken if BigDecimal is to be used as a key within these classes.

Caching in C#

This type of caching does not exist in C# as can be seen from the equivalent code example:

```
Int32 a = 5, b = 5;
Int32 c = 200, d = 200;
Console.WriteLine(a == b);
Console.WriteLine(c == d);

Console.WriteLine(a.Equals(b));
Console.WriteLine(c.Equals(d));
```

Produces the output:

```
true
true
true
true
```

Testing

During a white-box assessment the code should be checked to ensure the correct methods have been used to compare values. For example, in Java ensure that “equals()” has not been mistakenly used in place of “compareTo()”. The code should also be checked to ensure that “==” has not been used to compare values.

When conducting a black-box assessment, test operations where comparisons may have been used (such as when comparing balances or limits) by noting any differences in the application when handling values with one or two decimal places. For example, if the software states that a transaction must be 5GBP but 5.00GBP has been submitted and fails it may be possible to infer that the BigDecimal comparison is being done using the “equals()” method.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Recommendation

Never use the "==" operator to determine it two objects contain the same value, instead use the appropriate methods of the given class. In Java's BigDecimal, use the "compareTo()" method to compare two objects instead of using the "equals()" method.

Rounding Errors in Currency Conversion

Applications often have to express values with a fixed number of decimal places, and numbers that would normally occupy a greater number of decimal places must be rounded either up or down in order to fit. The *round-to-nearest, ties away from zero* rounding mode is the convention most commonly applied; where 0.4 is rounded down but 0.5, representing a 'tie' in distance from 0 and 1, is rounded up. This is perfectly adequate for most normal applications. However, within the arena of financial transactions, rounding up or down may lead to situations where users exploit the calculations for financial gain.

The most common scenario where rounding of some sort can often not be avoided is when dealing with currency conversion. The exchange rate between two currencies is often expressed as a value with a large number of decimal places. In this scenario, rounding of some sort may simply be inevitable, yet care must still be paid as to how the conversion is implemented, to avoid scenarios where either money can be erased, or more worryingly for financial institutions, where money can be created by rounding values upwards.

Although the largest amount of money that can be erased or created by rounding is the smallest unit of a given currency, if care is not paid to other factors such as the number of transactions permissible in a given time period, repeated transactions involving rounding can soon lead to situations where large quantities of money are involved.

For example, given an exchange rate of 0.745 from Euros into Sterling, a Euro would result in 75 pence if rounding up:

€1 * 0.745 = £0.75 (rounded to 2 decimal places)

Thus one thousand repeated transfers like this would give £750. However moving this back into Euros in a single transfer with the same rate would give :

£750 / 0.745 = €1006.71 (rounded to 2 decimal places)

Thus €6.71 can be created by performing many small transactions that exploit the rounding in these calculations.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Testing

Testing for such issues involves ascertaining if the rounding convention used implements rounding up by performing a transaction with appropriately chosen numbers. For example, with an exchange rate of, say, 2.03679 between Dollars and Sterling:

2 * 2.03679 = 4.07358

Therefore:

£0.20 * 2.03679 = $0.407 (to 3 decimal places)

If when testing the application, a result of $0.41 is given, it is clear that rounding up has occurred.

A key factor in practically exploiting these issues is that the application allows a user to easily perform many transactions using automated means (such as a script). The protections implemented which prevent automated submission should be taken into account when evaluating the risk posed by this vulnerability.

Recommendation

A possible defence to such an attack would be the creation of a limit to the number of transactions a given user is allowed to perform in a given time frame. While not overly restrictive for legitimate customers, it would thwart large scale fraud by placing a small upper limit on the amount of money that might be created or lost based on manipulation of rounding. Additionally, strategies to detect aberrant behaviour in relation to such transactions may also be advisable if not already in place within anti-fraud and other monitoring and analysis procedures. This could be used to assist in the detection of slow attacks, which a determined attack may employ to avoid daily transaction limits.

By simply implementing a system of commission for the institution, the issue is conveniently avoided altogether by reversing the small potential gain of a single transaction. If the functional specification of an application does not include a provision for a commission mechanism, to deal with this issue one might choose to always round down to avoid loss. However this might be dimly viewed by customers as it is they who thus lose on the transaction. Rounding of values in favour of the organisation may raise legal concerns if not conducted in an appropriate manner. Where necessary, seek clarification from the appropriate standards and legal bodies (e.g. regulators).

A better method would be to quote different exchange rates depending on the direction of the transaction between two currencies that are both slightly adjusted to counteract rounding in favour of the institution. Whilst minimising any loss to the customer, this system leaves the customer free to decide against proceeding with the transaction by evaluating the fairness of the rates that are offered.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Upon comparing the rates, the difference will appear minimal, but it should be clear from the outset that they fairly exclude situations where any user might make money out of the system.

Conclusions

Accurately processing numeric data is of paramount importance to organisations that depend on the accurate management of financial data. Financial applications must be designed and implemented with accuracy and correctness in mind to avoid direct financial loss and to comply with relevant regulatory requirements. Achieving this goal, however, can be problematic due to the way common programming languages deal with numeric data, especially floating point values. This is of particular concern during the validation stage, where the application determines whether numeric data is valid or not. The accuracy of rounding floating point values can also lead to exploitable vulnerabilities which attackers could use to manipulate transactions to return favourable outcomes.

The risk posed by these vulnerabilities can be managed by understanding how to identify the issues and how to correctly use the applicable programming APIs.

References

SOX: What does it mean for UK companies? - http://www.continuitycentral.com/feature0203.htm

UK Financial Services Authority http://www.fsa.gov.uk/pages/Library/Communication/PR/2007/021.shtml

IEEE Standard for Floating-Point Arithmetic - http://grouper.ieee.org/groups/754/

Operational Risk in the FSA Handbook - http://fsahandbook.info/FSA/html/handbook/SYSC/13/7

Acknowledgements

This paper was written by Adam Boulton, Kevin O'Reilly and Stephen de Vries, with contributions from Glyn Geoghegan and David Ryan.

About The Authors

Adam Boulton is a Research Developer and Security Consultant for Corsaire. He has been involved in all aspects of the SDLC with a focus upon security. He graduated from Sheffield Hallam University with a 1st Class Software Engineering Degree and is also certified for secure code assessments.

Adam's past roles have included that of a Software Engineer for the Ministry of Defence and a Virus Analyst for Sophos. At both positions he was heavily involved in Software Development, Reverse Engineering and

Copyright © 2008 Corsaire Limited.
All Rights Reserved.

Implementation. He loves challenges, especially when it comes to making code run fast, and finding simplicity and elegance in what looks like intricate chaos.

Stephen de Vries is a Principal Consultant in Corsaire's Security Assessment team. He has worked in IT Security since 1998, and has been programming in a commercial environment since 1997. He has spent the last eight years focused on Security Assessment and Audit at Corsaire, KPMG and ISS. He was also a contributing author and trainer on the ISS Ethical Hacking course. He is currently leading the OWASP Java Project and regularly presents on secure programming and testing.

Stephen's past roles have included that of a Security Consultant at a leading City of London Financial institution and also Security Engineer at SMC Electronic Commerce. At both positions he was involved in corporate security at many levels and was responsible for consulting on the paper security policies and procedures, conducting vulnerability assessments, designing, deploying and managing the security infrastructure of the organisation.

Kevin O'Reilly is a Research Developer and Security Consultant for Corsaire. He has a background in malware analysis and security research which combines experience in reverse engineering and software development. He graduated from Oxford University with a Master's degree in Physics.

Kevin's past roles have included a Research Studentship for the Nuclear and Astrophysics Laboratory in Oxford, and a position as Virus Researcher with Sophos. The former post had a particular emphasis on programming and mathematics, with the latter focussing on reverse engineering, malware analysis and security research.

About Corsaire

Corsaire are experts at securing information systems, consultancy and assessment. Through our commitment to excellence we provide a range services to help organisations protect their information assets and reduce corporate risk.

Founded privately in the United Kingdom in 1997, we operate on an international basis with a presence across Europe, Africa and the Asia-Pacific rim. Our clients are diverse, ranging from government security agencies and large blue-chip FTSE, DAX, Fortune 500 profile organisations to smaller internet start-ups. Most have been drawn from banking, finance, telecommunications, insurance, legal, IT and retail sectors. They are experienced buyers, operating at the highest end of security and understand the differences between the ranges of suppliers in the current market place. For more information contact us at contact-us@corsaire.com or visit our website at http://www.corsaire.com.

Copyright © 2008 Corsaire Limited.
All Rights Reserved.