nccgroup
freedom from doubt

An NCC Group Publication

Common Security Issues in Financially-Oriented Web Applications

A guideline for penetration testers

Prepared by:

Soroush Dalili

Version 1.1

Contents

1 Introduction

Today it is often hard to find individuals who have not purchased something online or used online financial services. Online services offer ease of use and provide other value-add properties such as loyalty card schemes to attract and retain customers, thus ensuring market competiveness. Creating new online commercial services is an imperative for most organisations, but has to be done in a safe and secure manner to meet client, regulatory, and legal expectations. Ecommerce applications, due to the value of the products and services they offer, are valuable targets for threat actors who are looking for financial gains or wish to damage a company's brand or reputation.

This document summarises NCC Group's experience of assessing ecommerce and financial services applications, providing a checklist of common security issues seen in financial services web applications.

Security assessments of ecommerce applications and financial services require specific security-minded test cases to be developed. These tests have to cover logical security issues or rare vulnerabilities that are usually not found through conventional security penetration or functional testing. Vulnerabilities such as price manipulation, buying items at a reduced price or even for free, or earning free money are the most interesting ones; however, these vulnerabilities don't represent all the possible attacks. Unfortunately, many application-specific ecommerce security issues cannot be identified by static or dynamic automated security scanners, or even in a manual source code review, if the reviewer does not have a complete understanding of the application rules, business processes, and threat scenarios.

In NCC Group's experience, one of the best ways to identify the business logic and application-specific security issues early in the development lifecycle is to write down all the rules (dos and don'ts) both for the business processes and the supporting software and systems. These rules can then be used to create a threat model. Specific security-focused test cases, scenarios, or checklists can then be designed based on this threat model, and used to identify vulnerabilities and verify the correctness of the implementation. Security-focused code reviewers and penetration testers benefit from these documents, as they provide information about the expected behaviour of the system and the thought patterns that guided its design. Automated security scanners (especially static analysis tools) can also have their performance improved, by defining new rules to detect specific issues once a detection pattern has been developed.

This whitepaper discusses the commonly-seen security issues that NCC Group has found over the last fifteen years of performing security assessments of real ecommerce and financial services web applications. The resulting checklist can be used as an additional tool for penetration testers when assessing ecommerce applications.

2 Common Vulnerability Classes in Financially-Oriented Web Applications

In this section we introduce you to the vulnerability classes, providing an overview of each and examples of how to test their presence.

We have omitted generic web application issues, such as those involving authentication, authorisation, and input validation; instead the issues discussed in the following sections are those that have specific relevance to financially-oriented web applications. Where possible we have mapped these into the categories used in the Common Weakness Enumeration. [10]

2.1 Time-of-Check-Time-of-Use (TOCTOU) and Race Condition Issues

CWE: 367 and 557

TOCTOU is a software bug that occurs when an application checks the state of a resource before using it, but the resource's state changes between the check and the use in a way that invalidates or changes the results of the check.

Time and order sequence are crucial to correct financial software operations. Many financial transactions rely on checking balances and values (sometimes in real time) before processing. If there is latency, delay, or opportunity to modify values between these checks, or if resource coordination is not properly implemented around multi-threaded solutions, then there may be scope for manipulating application logic, perhaps for financial gain.

2.1.1 Transferring Money or Points, or Buying Items Simultaneously

This is a common flaw within ecommerce applications that keep users' balances and allow money transfer or simultaneous purchases.

Consider the following example, commonly seen by NCC Group [1]. A user is authenticated to a financial application from two different devices in the same session. A transaction is performed seeking to transfer money from account number 1019 to account number 9823 for the amount of £100.

Suppose the server-side code is as follows, and that the user's account balance is £100:

```
1: if (amount <= account_balance) {
2:     account_balance = account_balance – amount
3: }
```

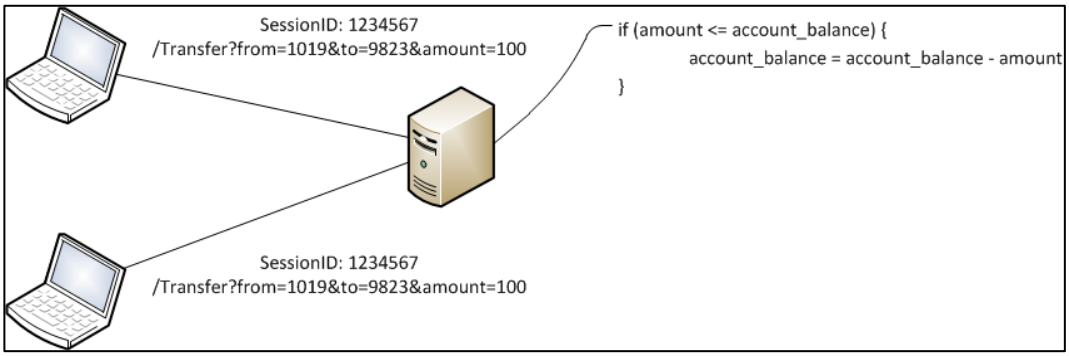

Figure 1 - Concurrent sessions can lead to race conditions

If the transfer request can be fired twice into the web application at the same time, there is the potential that line 1 in the code to be executed twice before line 2 is executed. If this occurs, then the first check that £100 is <= £100 holds true twice, and so when the *if* statement block executes, the `account_balance` value is decremented by £100 twice, meaning that the user has been able to transfer more money than allowed (as dictated by the *if* statement in the source code).

This problem can be exploited by writing a simple piece of multithreaded code that uses multiple valid sessions for the same user (for example by simulating concurrent login on multiple devices).

The implications of this could be severe, depending on the nature of the application, as it opens up the potential for theft or use of unauthorised amounts of money, and provides a potential mechanism for fraud and other laundering activities.

This issue can be found in many financial applications, such as a banking application that allows money transfer between multiple accounts, a shopping website in which a user can buy multiple items at the same time, or a commercial website that allows its users to earn and transfer their reward points.

Some applications may even prevent a user from having a negative value in their account by replacing negative values with 0. If the application in the above example had this feature, the user could potentially gain £100.

Concurrency issues may also affect discount voucher codes which can only be used once or twice.

2.1.2 Changing the Order upon Payment Completion

Applications that allow users to change their order while paying for an item can also be vulnerable when there is no verification at the end of the process. Although race conditions in changing the shopping basket while payment is being processed seem a little tricky, often there is no need for this, especially if the payment page is not part of the application but is a third-party website or an external module. In this case, the order can be changed while the user is on the payment page and before clicking the “pay” button to complete the payment. Changing items in the basket, shipping method and posting address, quantity of items, and so forth can affect the final price while the application still uses the initial cheaper price.

The following case shows an example of this vulnerability which was seen in a production website.

A cheap item was selected and added to the basket. The user then went to the checkout page to pay for the selected item. At this stage, without closing the payment page, the user opened the main website in a new browser tab (to use the same session token) and added other and more expensive items to the basket. After doing this, the user went to the initial checkout page, which was open in the previous browser tab, to complete payment for the initial item. When the order was completed, all the items in the basket were shown as paid in the final receipt. The user could buy additional items for free, while only paying for the initial item.

This vulnerability may also exist in the deposit process when an application can hold users’ balances.

NCC Group has also encountered a rare scenario in which an application validated the input values and stored them in the session regardless of the validation result. In this case, the application did not go to the next stage when an input value was invalid. However, if a user went to the next stage by providing valid values, and then replayed the previous request with invalid values, the application stored the invalid values in the session and did not validate them any more as the user had already passed that stage. This caused severe logical issues for the application.

2.2 Parameter Manipulation

CWE: 20 , 691 , 693 , 179 , 345 , 807 , 115 , 133 , 166 , 167 , 168 , 171

Parameter manipulation is a key technique for exploiting many of the security issues outlined in this paper. Below, we discuss the most interesting parameters that should be considered and tested during an assessment of a financial application.

2.2.1 Price Manipulation

Price manipulation is an important test for any ecommerce applications in which the user can purchase a product. Applications normally send the price data to the payment pages, especially when the payment module is not part of the web application and therefore does not have access to users’ sessions or the database. It is also possible to find applications that send the price data upon selecting an item to add it to the basket.

Sometimes it is possible to buy the same item cheaper or even for free by manipulating its price. Although nowadays it is very rare to find an application that accepts negative numbers via the price fields, this always needs to be tested as it may change the application flow completely.

The following interesting example has been seen by NCC Group in recent years:

The ecommerce site's "add to basket" mechanism contained a "price" parameter in a hidden field, but the application ignored a manipulated price in the request and used the correct value instead. However, it was found later that by adding a number of sale items (items with additional discounts) to the basket, the application started using the price parameter within the request, and allowed price manipulation and negative values (see the "Dynamic Prices, Prices with Tolerance, or Referral Schemes" section for more information).

Sometimes, when the application is badly implemented, it is possible to change the price value on the callback from the payment server (which goes through the user's browser and not via the backend APIs). In this case, the user can alter the price before going to the payment page, and after completing the transaction the price in the callback URL will be changed to reflect its initial value. The user could later ask for a refund and gain this money. Although it is rare to see a vulnerable application like this nowadays, it is always worth checking it.

2.2.2 Currency Manipulation

Although an ecommerce website may not accept different currencies, payment applications normally accept them, and they generally require the currency parameter to be specified in the initial request. If a website does not validate the currency parameter upon completion of a transaction, a user can cheat by depositing money in a currency which has a much lower value than the requested currency. The following example shows a badly-implemented PayPal payment method that could be exploited:

The user wanted to make a payment of £20 to a website, using the PayPal payment option. The request that the website sent to the PayPal website was intercepted and the currency parameter was changed to "INR" (Indian rupee) from "GBP" (British pound). After completing the transaction on the PayPal website with 20 Indian rupees, the website authorised the transaction without checking the currency, and £20 was deposited in the user's account while only £0.22 was withdrawn from the PayPal account.

2.2.3 Quantity Manipulation

Websites calculate a final price based on the quantity of items purchased. Therefore, it may be possible for this parameter to be manipulated to contain small or negative values, to affect the price on the final payment page.

The website may remove items that have zero or negative values within the quantity parameters. In this case, decimal values such as "0.01", "0.51", or "0.996" can be tested to see if they have any effects on the final price. This method can be more dangerous when used on items which are not normally manually reviewed.

2.2.4 Shipping Address and Post Method Manipulation

Changing the shipping address and the posting method may change the cost. Therefore, it is important to manipulate them in the last stage of the payment process to check whether it changes the cost. It is sometimes possible to change the shipping address after placing an order and before receiving the invoice, by changing the user's profile address, so this needs to be tested as well. This can also be a TOCTOU issue – see that section.

The tax value can also be based on the address. This should be tested to ensure that it is not easy for an attacker to avoid required taxes, such as VAT, by manipulating the address in the process.

2.2.5 Additional Costs Manipulation

Any additional parameter that can affect the final cost of a product, such as delivery at a specific time or adding a gift wrap, should also be tested, to ensure it is not possible to add them for free at any stage of the payment process.

2.2.6 Response Manipulation

Sometimes application payment processes, application license checks, or in-app asset purchases can also be bypassed by manipulating the server's response. This threat normally occurs when the application does not verify the response of a third party and the response has not been cryptographically signed.

As an example, there are applications with a time-restricted trial version which do not cryptographically validate the server's response upon purchasing a license. As a result, it is possible to activate the application without paying any money, by intercepting and manipulating its server's response to a license purchase request.

Other examples include mobile games which download users' settings from a server after opening the app. For vulnerable applications it is possible to manipulate the server's response to use non-free or locked items without paying any money.

2.2.7 Repeating an Input Parameter Multiple Times

This is very rare, but repeating an input parameter within a request that goes to the application or to the payment gateway may cause logical issues, especially when the application uses different codebases or different technology to parse the inputs on the server side.

Different technologies may behave differently when they receive repetitive input parameters. This becomes especially important when the application sends server-side requests to other applications with different technologies, or when customised code to identify the inputs is in place.

For example, the "amount" parameter was repeated in the following URL:

```
/page.extension?amount=2&amount=3&amount[]=4
```

This has different meaning for code written in ASP, ASP.Net, or PHP, as shown below:

```
ASP → amount = 2, 3
ASP.Net → amount = 2,3
PHP (Apache) → amount = Array
```

This test shows a classic example of HTTP parameter pollution [10]. However, repeating input parameters is not only limited to normal GET or POST parameters, and could be used in other scenarios such as repeating a number of XML tags and attributes in an XML request, or another JSON object within the original JSON objects.

2.2.8 Omitting an Input Parameter or its Value

Similar to repeating input parameters, omitting parameters may also cause logical issues when the application cannot find an input or sees a null character as the value.

The following cases can be tested for sensitive inputs to bypass certain protection mechanisms:

- Removing the value
- Replacing the value by a null character
- Removing the equals-sign character after the input parameter
- Removing the input parameter completely from the request

2.2.9 Monitor the Behaviour while Changing Parameters to Detect Logical Flaws

Just as when testing non-financial applications, all the input parameters within the payment process should be tested separately in order to detect logical flaws. In the example below, the payment process flow could be changed by manipulating certain parameters.

In a web application, there was a parameter which was used to tell the server to use the 3D-Secure mechanism, which could be manipulated to circumvent this checking process.

Sometimes web applications contain a parameter which shows the current page number or stage. A user may be able to bypass certain stages or pages by manipulating this parameter in the next request.

It is not normally recommended to change more than one parameter during limited time frame of testing; however, some logical flaws can be found only by changing more than one parameter at a time. This is useful when an application detects parameter manipulation for parameters such as the price field. Although it may not be feasible to test different combinations of all the input parameters, it is recommended to modify at least a couple of the interesting inputs at the same time. In order to automate this test, the target field such as the price or the quantity parameter can be set to a specific amount that is not normally allowed, and then other parameters can be changed one by one to detect any possible bypass of current validation mechanisms when the application accepts the manipulated items.

The following shows an example of this kind of vulnerability.

Suppose the server-side code is as follows:

```
1:     Try
2:           ' Delivery type should be an integer
3:           deliveryType = Int(deliveryType)
4:           ' Quantity should be an integer
5:           quantity = Int(quantity)
6:     Catch ex As Exception
7:           ' Empty catch!
8:     End Try
9:     ' Continue ...
```

This code will make sure that the “deliveryType” and the “quantity” parameters contain integer numbers. However, due to an empty `Catch` section in line 7, the “quantity” parameter can still contain a decimal number such as “0.1” when the “deliveryType” parameter contains a string such as “foobar”. In this case, the application jumps to the `Catch` section due to an error in converting a string value to an integer, without converting the “Quantity” parameter to an integer.

2.3 Replay Attacks (Capture-Replay)

A replay attack occurs when all or part of a message between the client and the server are copied and replayed later. The parameters can also be changed when no parameter manipulation prevention technique such as message signature validation is present on the server side. Although a message can be signed or encrypted to prevent parameter manipulation, this will not stop replay of a message which was originally created by a trusted party.

An application can be vulnerable to serious security issues when it trusts replayed requests without performing any further validation to check whether they have already been received or sent in the right order.

2.3.1 Replaying the Call-back Request

It is quite normal for the payment systems to redirect the user to a specific page when a payment has successfully been processed or failed. Sometimes it is possible to replay a request which was for a successful payment, to authorise a transaction which has not yet been processed.

For example, a website signed all the input parameters except the “transaction-id” parameter in a successful callback request. This parameter could be replaced with a new transaction-id to complete a payment without spending any money.

2.3.2 Replaying an Encrypted Parameter

Sometimes websites encrypt some of the important parameters without creating a mechanism to detect replay attacks. For example, there was a website which encrypted price values on the server side to include them in hidden input fields. Although direct price manipulation was not possible when price parameters were encrypted, it was still possible to use the encrypted price parameter of cheaper items to buy more expensive items.

2.4 Rounding Issues

CWE: 187 and 681

Numerical values can be stored in integer or float variables. Although float variables can contain numbers with some digits after the decimal point, the number of digits is still finite and based on the variable type and its precision. Integer variables can only contain numerical values which do not have any digits after the decimal point.

When a mathematical value is stored in a numerical variable, it needs to be rounded based on the precision of the variable type. As a result, the new stored number can be slightly greater or smaller than the original value. This normal behaviour can sometimes be abused by attackers.

2.4.1 Currency Rounding Issues

The following images show an example of exchange rates (USD to/from GBP) in Google at one time:

Figure 2- Exchange rate from USD to GBP in Google (rounded by two digits after the decimal point)

Figure 3- Exchange rate from GBP to USD in Google (rounded by two digits after the decimal point)

As Google rounds the numbers to two digits after the decimal point, someone could convert $0.20 to £0.14 (something like £0.1352 before rounding) and then convert £0.14 to $0.21 (something like £0.2070 before rounding) with a profit of $0.01. By doing this a hundred times, a dollar could be created. However, the following images show the exchange rate with four digits after the decimal point in another website at the same time (LikeForex.com):

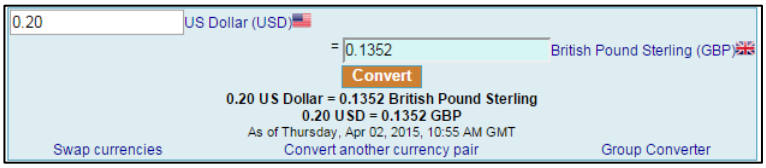

Figure 4- Exchange rate from USD to GBP in LikeForex (rounded by four digits after the decimal point)

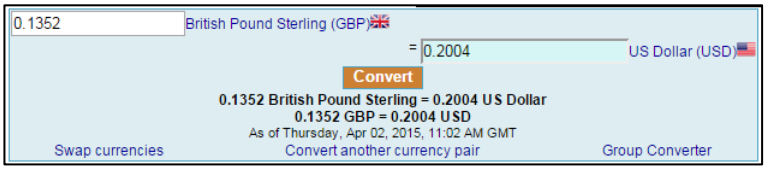

Figure 5- Exchange rate from GBP to USD in LikeForex (rounded by four digits after the decimal point)

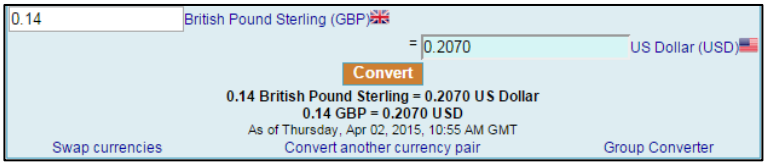

Figure 6- Exchange rate from GBP to USD in LikeForex (rounded by four digits after the decimal point)

In this case, someone could exchange $0.20 for £0.1352 and then exchange £0.1352 for $0.2004, giving a small profit of $0.0004. These exchanges needed to be done 2500 times to create one dollar.

If a real financial application converts different currencies to each other without a commission fee or without different buy and sell rates in favour of the company, this can lead to a financial gain for an attacker [2].

Shopping applications that support multiple currencies can also become victims of currency rounding issues, when a user can buy an item with one currency and refund it with another.

In addition, applications in which users can deposit money into their accounts (such as banks, international calling card companies, or gambling websites), can become vulnerable if they support multiple currencies with different exchange rates and a user can withdraw the deposited money from their accounts immediately without any cost. Changing the currency of the account after the first deposit can also lead to this vulnerability. This can be more problematic when the application uses a different exchange rate than the payment gateway (see the “Currency Arbitrage in Deposit/Buy and Withdrawal/Refund” section).

2.4.2 Generic Rounding Issues

Rounding issues are not always limited to currency exchange. Even shopping applications which only support one type of currency can be affected by inconsistencies between different parts of the application.

The following is an example of this type of inconsistency, which should be tested for:

The user chooses to deposit £10.0049 to a website that can hold the user’s balance; the website keeps this money in the database to authorise it and adds it to the user’s balance when the money transfer from the bank is completed. However, the banking API only accepts numbers with two digits after the decimal pointer based on its standard. Therefore, the application converts the money to £10.00 and waits for the user and the payment gateway to complete this transaction. After the transaction is completed, £10.00 will be deducted from the user’s bank account but £10.0049 will be deposited into the site’s balance. After repeating this process 205 times, the user can gain £1.00.

The same problem arises when the monetary calculation within the same application is done by different applications or different codes. One example can be the use of database stored procedures for some of the calculations (for example money transfer) and C# code with different rules for other monetary calculations (such as money withdrawal or cancelling the money transfer).

2.5 Numerical Processing

CWE: 189

Obviously numbers play an important role in financial systems [4]. Manipulating numbers for ecommerce applications can lead to different logical issues and money loss in severe cases. Therefore, different test-cases should be designed to test numerical parameter manipulation in numerical fields such as price, quantity, voucher codes and so on.

2.5.1 Negative Numbers

Negative numbers can lead to a number of logical issues. Most of the time, they reverse the application logic so, for example, a user may be able to deposit “£100” by refunding “-£100” from the system. Any coefficient value such as the quantity parameter can also be used for this purpose.

As the application logic is reversed, transferring “-£100” into another account can be like transferring money from the target account to steal their money. The same logical issue applies to reward points or within gaming applications in which chips or other virtual currencies are used instead of money to buy virtual items.

Although using negative numbers in different parameters does not always reverse the application logic, it can cause other useful logical flaws and it should always be tested.

The “-1” value should also be tested separately, as it can have a specific meaning for the application, as developers often use it to initialise numerical parameters or when a condition has not been met.

2.5.2 Decimal Numbers

In addition to the rounding issues which were discussed earlier, decimal numbers can cause logical issues for applications, especially when a parameter such as quantity should only accept integer values. Decimal values can also be used to exploit rounding issues – see that section. An additional use of decimal values is to create the same transactions multiple times when there is a restriction on uniqueness of items in an order; in this case, it can be used in numerical id parameters to point to the same item multiple times by having values such as “1234”, “1234.00”, or “01234.000001”, which can have the same meaning when processed by the payment system or the database.

2.5.3 Large or Small Numbers

Range validation check is an important test, which should be done using a value slightly larger or smaller than the maximum and minimum values (decimal numbers can be used here as well).

2.5.4 Overflows and Underflows

A numeric overflow or underflow can occur when a value or the result of a calculation is bigger or smaller than what can be stored for that variable type in the memory or the database.

For example, in Java or C#, if an integer value reaches the maximum value (“2^{31}-1 = 2147483647”) and is incremented, an overflow occurs, without causing any error, which causes the value to roll-over into the smallest minimum value (“-2^{31} = -2147483648”). These numbers can be used to bypass some validations.

2.5.5 Zero, Null, or Subnormal Numbers

“0”, “NaN”, or null characters can be used in different contexts, especially for price manipulation. Non-zero numbers with magnitude smaller than the smallest normal number and which are nearly equal to zero, such as “0.00000000000000000000000000000001” or “1e-50”, should also be tested.

2.5.6 Exponential Notation

Exponential notations are quite useful for bypassing length restrictions in which the numerical values cannot contain certain number of digits.

For instance, when only four characters are allowed, the following notation can bypass the “9999” restriction as the maximum value:

```
9e99 = 9 * 10^99 → 100 digits
```

Another example is when the dot character (“.”) is not allowed to create decimal numbers:

```
1e-1 = 0.1
```

2.5.7 Reserved Words

The following reserved words can be used in Java and C# applications to represent a number, which can cause serious logical issues:

```
NaN
Infinity
-NaN
-Infinity
```

2.5.8 Numbers in Different Formats

Numbers in different technologies can be written in different formats to bypass validation mechanisms. For instance, when sending “0” as a value is restricted, “0.00”, “-0.00”, or even “$0” or “£0” could be allowed.

The following table shows response of different functions within ASP Classic (VBScript), C# .NET, Java, and PHP to several presentations of numbers.

Columns Description:

A. VBScript – ASP Classic `IsNumeric` function
B. C# – .NET `IsNumeric` function
C. C# – .NET `Double.TryParse` function + result value
D. Java – `Float.valueOf` function + result value
E. PHP – `is_numeric` function
F. PHP– `floatval` function + result value

String	A	B	C	D	E	F	Comment
001.0000	**True**	**True**	**True** (1)	**True** (001.0000)	**True**	**True** (1)	Decimal symbol with leading zeros based on the regional settings of the server
$10	False	**True**	False (10)	False	False	False	Currency symbol based on the regional settings of the server (culture format).
1,,2,,,3,,	**True**	**True**	**True** (123)	False	False	**True** (1)	Digit grouping symbol based on the regional settings of the server (culture format). Can be created by HPP too.
-10.0	**True**	**True**	**True** (-10)	**True** (-10.0)	**True**	**True** (-10)	Negative symbol based on the regional settings of the server. It could be a positive sign.

String	A	B	C	D	E	F	Comment
(10)	**True**	**True**	False (-10)	False	False	False	Negative symbol based on the regional settings of the server.
10-	**True**	**True**	False (-10)	False	False	True (10)	Negative symbol based on the regional settings of the server. It could be a positive sign.
1e2	**True**	**True**	**True** (100)	**True** (1e2)	**True**	**True** (100)	String length can be less than the number's length
%20%091	**True**	**True**	**True** (1)	**True** (1)	**True**	**True** (1)	Space characters (09-0D and 20) Space characters (09-0D and 20) %20=Space %09=Tab
1%20%00%00	**True**	**True**	**True** (1)	**True** (1)	False	**True** (1)	Space characters (09-0D and 20) followed by Null Character(s)
&hff	**True**	**True**	False (255)	False	False	False	&h and &o can be used in VBScript to represent a number in Hex or Octal.
Infinity	False	**True**	**True** (Infinity)	**True** (Infinity)	False	False	Infinity: a reserved Word for C# and Java
NaN	False	**True**	**True** (NaN)	**True** (NaN)	False	False	NaN (not a number): a reserved Word for C# and Java
0x0A	False	False	False	False	**True**	False	Hex format
An Array	False	False	False	False	False	**True** (1)	Providing an input as an array. e.g.: p.php?in[]=val
%0B%09%20-0001,,,,2.8e0002%09%20%0C%00%00	**True**	**True**	**True** (-1280)	False	False	**True** (-1)	An example using the above notations
%0B$%09%20(0001,,,,2.8e0002%09%20)%0C%00%00	False	**True**	False (-1280)	False	False	False	An example using the above notations

Note 1: "`Integer.parseInt`" in Java cannot convert any of the numbers in the above table.

Note 2: "`Convert.ToInt32("0X0A", 16)`" in C# .Net returns "10". This function cannot convert other numbers in the above table though.

Note 3: PHP 5.4 supports a binary prefix ("0b") that can be used to create a number as well.

2.6 Card Number-Related Issues

Payment card numbers are some of the most attractive data for attackers. In addition to being used for online shopping, they can be sold in black markets even without the card verification code or value (three-digit or four-digit number printed on the front or back of a payment card).

Nowadays many ecommerce websites are compliant with the Payment Card Industry Data Security Standard (PCI DSS) [8], making them more secure, in order to attract more suppliers and customers and to reduce the risk of card data breaches. As a result, they must not permanently store the card verification code used to verify card-not-present transactions. In addition, they must encrypt the card numbers in their storage.

The following examples discuss two different security issues to which PCI-compliant web applications can still be vulnerable.

2.6.1 Showing a Saved Card Number during the Payment Process

Ecommerce websites may reveal users' saved bank card numbers during the checkout process. Most of the time, this occurs due to a bad implementation, and the card number is not required to be displayed. Sometimes, however, the card number should be decrypted on the payment page; for instance if it is to be sent to a 3D-Secure authentication website.

This can be problematic, as an attacker who has hijacked a user's session or credentials or is exploiting a cross-site scripting (XSS) issue can obtain the card numbers.

The risk can be mitigated if card numbers are only displayed when necessary, the pages which contain the card numbers are password protected, and the 3D-Secure authentication process or similar mechanisms cannot be activated directly by accessing those pages when they are not required.

NCC Group also often finds unsaved card numbers in HTTP responses after using a card number in a payment process and before logging out of the website. This behaviour can also be dangerous, especially when the website is vulnerable to XSS or session-hijacking attacks.

It should be noted that the CVV (CV2) numbers (the card verification code) must not be seen in any of the responses from the server at any time.

2.6.2 Card Number Enumeration via Registering Duplicate Cards

Some websites do not allow their customers to save the same card number in multiple accounts. One of the reasons is to detect duplicate accounts or to stop abusing first-time buyers' offers.

This functionality, when it is badly implemented, can be abused to brute-force other users' card numbers which are registered on the website.

2.7 Dynamic Prices, Prices with Tolerance, or Referral Schemes

CWE: 840

Sometimes prices and discounts can be dynamic because of currency exchange rates, number of sold items, referral schemes, and delays in submitting a price in dynamic trading systems.

Therefore, the application specification should be reviewed to see if it supports dynamic prices. Most of the time, an additional input parameter helps the application to recognise the use of dynamic prices. For instance, the system may start using dynamic prices when the application does not use the default currency or when a customer uses a mobile device or resides in a certain country which can have a slower Internet speed. It may also consider using submitted prices when a referral header or a referral parameter is available. In order to find these systems, a number close to the original price (price ± 0.01) should be submitted while changing the other parameters.

Other parameters that affect the final price may also be dynamic or have a margin of threshold. For example, it is quite normal to see this behaviour in the "odds" parameter of a live betting application.

The application policy should be reviewed whenever dynamic prices are found, to ensure that the changed prices are within the allowed margin. In addition, a secure cryptography method should be used when the prices are generated by a trusted party or even by the website itself, in order to identify any manipulation by untrusted parties.

2.8 Discount Codes, Vouchers, Offers, Reward Points, and Gift Cards

Discount codes and vouchers which can be used to reduce the final price should be tested to ensure they are not predictable and cannot be easily enumerated, and that it is not possible to use them to buy illegitimate items (for instance using new product discount codes to extend old services). The refund process should also be tested to ensure that the discounted values are not refunded.

The application behaviour after applying any discount method should be reviewed to see if there is any interesting parameter that can be manipulated or replayed to use a discount code for different products, after a certain date when it is expired, or multiple times when it should expire after the first use (concurrency issues can also be tested here).

In addition, applications need to be tested to ensure that the discount values are calculated at the last stage of purchase when the user changes the initial order in any way (adding/removing items or changing the quantities).

Offer schemes such as buy-one-get-one-free in which the user only pays for the most expensive item can also be abused to buy inapplicable items for free or to pay for the cheapest item to get the expensive one for free. In addition, the refund process needs to be tested to ensure that all the free items are returned along with the paid items.

Additional tests need to be performed to bypass any available restrictions such as limited quantity of specific items in sale.

Users can earn reward points in many ecommerce applications when the points can be used to purchase items, they should be treated and tested exactly like the user's balance. Therefore, negative number issues, rounding issues, concurrency issues, and so on should all be tested. The refund process should also be tested to ensure that a user cannot earn free points by buying and refunding items. The points might be spent in between buying and refunding items; in this case, users may not have enough points in their reward cards when refunding an item, and an appropriate policy should be in place to recover the lost points.

If users receive reward points by referring someone else to register or by registering themselves for the first time, they can abuse the reward point scheme by using point transfer functionality. Although point transfer functionality may not be directly accessible to the users in an application, it can be available upon closing an account or when a loyalty card has been lost or stolen.

Gift or loyalty card numbers should be unpredictable and very difficult to enumerate, otherwise an attacker can create a duplicate card to use a victim's balance. When these cards carry a spendable balance, they should be treated similarly to bank card numbers and should be protected by PIN codes or passwords.

2.9 Cryptography Issues

CWE: 310

Cryptography methods such as encryption, encoding, signing, and hashing are often seen within payment systems. However, design errors and implementation mistakes, due to human error, or the lack of attack vector knowledge, are quite commonly seen in this area, especially in applications which implement their own cryptography methods rather than using well-known pre-implemented libraries.

For example, when an application hashes some of the known parameters with a short and insecure secret key, this key can easily be brute-forced when the algorithm is known. Sometimes applications do not use long and strong secret keys when the implementation does not enforce it.

Another example is length-extension attack, in which the hash of a secret key which is concatenated with other values can be exploited to add data to the original request by padding the original data and calculating a new hash (see [5] and [6] for more details).

When the encrypted values are used in multiple places within the input parameters (in cookies, or POST/GET requests), the application often decrypts them in multiple places as well. The user may be able to use those pages to decrypt unknown encrypted values in order to understand how the application works. The problem can be severe if a user can shape and encrypt arbitrary data by using the provided input parameters in order to replace the current encrypted parameters.

As was discussed in the “Replay Attacks” section, sometimes there is also no need to break the cryptography methods, as they can be replayed.

2.10 Downloadables and Virtual Goods

CWE: 425

Ecommerce applications which sell virtual goods such as application files, MP3s, streaming videos, or PDF and document files can often be vulnerable to direct object reference attacks. In this case, an attacker can download or use non-free materials for free just by guessing or finding the actual URLs of the virtual products.

2.11 Hidden and Insecure Backend APIs

CWE: 656

Backend APIs which are used by electronic point of sale systems or payment servers are often old and insecure, as they are not directly accessible to the users. Sometimes even mobile or tablet application APIs are also insecure, as the developer did not think about security in the server side application layer when implementing them.

Some of these APIs and web services do not have any protection against many of the described attack techniques, and some of them even suffer from access control issues, allowing an attacker to perform administrative tasks such as balance adjustment.

2.12 Using Test Data in Production Environment

CWE: 531

In order to implement an ecommerce application, test payment methods and dummy card data are normally used in the testing or staging environments to prevent sending test requests to the live payment APIs or banks. Developers often miss removing a code from the production environment that is supposed to be only available in the testing environment. As a result, it is sometimes possible to change some of the parameters in the request to force a live application to use the test data. In addition, an ecommerce application may not show all of its payment methods, especially when they are not enabled for a specific user or when they are not fully implemented. Some of the test pages with which developers test and debug the functionality of third party APIs to ensure they work in the right way can also be available on ecommerce websites. These debugging functions and test pages can put the website in danger when found by an attacker.

The following shows an example of this vulnerability:

A website sent a numerical payment type to the server, alongside the other parameters which were needed to complete a transaction. However, changing the payment type to other numerical values could force the application to use the test payment gateway that used testing accounts to simulate the live environment. This allowed an attacker to complete a transaction without spending real money, just by connecting the application to its testing environment.

The destination page and all the input data in a payment request should be examined and tested to make sure that is not possible to force a live application to use test data. For instance, sometimes changing the “HOST” header in the HTTP request to a known internal hostname that is used for testing can trigger this vulnerability.

In addition, payment-specific testing data should also be tested in order to make sure that it is not possible to use it in the live environment. For instance, in one application it was possible to use the Sage Pay test card data [9] in a real transaction.

2.13 Currency Arbitrage in Deposit/Buy and Withdrawal/Refund

If an ecommerce application supports different payment methods with different currencies, someone can potentially deposit money in one currency and withdraw it with another. Arbitrage occurs when the deposit and the withdrawal methods are different (such as using a credit card company to deposit money and PayPal for money withdrawal) and they use inconsistent exchange rates.

For instance, if US/EUR exchange rate is 3/2 (three dollars per two euros) in a bank, and 4/3 in PayPal, by withdrawing eight euros from the bank, twelve dollars will be deposited in the website, and by withdrawing twelve dollars from the website, nine euros with be deposited into the PayPal account, giving the user one additional euro.

A more complicated issue can be found when a financial application supports money transfer with different currencies, as multiple-currency arbitrage (such as triangular arbitrage) can be exploited when the commission fees are negligible.

It is rare to see this vulnerability among banking and trading applications, due to the use of high-speed computer networks which can alarm them to close the gap whenever arbitrage can happen [3]. However, an ecommerce application that updates its exchange rate slowly can be a victim of this exploitation technique.

3 Conclusions

In this paper, the following attack methods and testing methodologies were discussed against ecommerce, payment, and trading applications:

- Time-of-Check-Time-of-Use (TOCTOU) and race condition issues
  - Transferring money/points or buying items simultaneously
  - Changing the order upon payment completion
- Parameter manipulation
- Replay attacks
  - Replay the call-back request
  - Replay an encrypted parameter
- Rounding issues
  - Currency rounding issues
  - Generic rounding issues between different applications
- Numerical processing
- Credit card and other payment card related issues
  - Showing a saved card number during the payment process
  - Card number enumeration via registering duplicate cards
- Dynamic prices, prices with tolerance, or referral schemes
- Discount codes, vouchers, offers, reward points, and gift cards
- Cryptography Issues
- Downloadables and virtual goods
- Hidden and insecure backend APIs
- Using test data in production environment
- Currency arbitrage in deposit/buy and withdrawal/refund

These attack methods can also be used against other similar applications such as betting and gambling applications, or other financial services platforms.

In addition to the items which were discussed in this research, web applications should also be tested for common vulnerabilities to ensure comprehensive coverage. Organisations such as OWASP provide good advice on what to cover, and how to gain this coverage.

It is clear that while there are common factors in all web applications, understanding the supporting business process and thus specific threats is imperative in order to tease out certain vulnerabilities. It is for this reason that today humans can provide a more complete picture than automated tooling alone. In the future we can expect approaches such as expert systems to go some way to make up this ground, however today certain vulnerability classes, and thus threats, can only reliably be discovered by humans and manual tests.

4 References and Further Reading

The following references were used in the production of this whitepaper.

1. Research Insights Volume 1 - Sector Focus: Financial Services

   https://www.nccgroup.trust/media/481879/research-insights-vol-1_sector-focus-financial-services-mar2015-online.pdf
2. Is Your Online Bank Vulnerable To Currency Rounding Attacks?

   http://blog.acrossecurity.com/2012/01/is-your-online-bank-vulnerable-to.html
3. Currency Arbitrage

   http://www.investopedia.com/terms/c/currency-arbitrage.asp
4. Corsaire Whitepaper: Breaking the Bank

   http://lists.owasp.org/pipermail/webappsec/2008-July/000634.html
5. Ron Bowes – Crypto: You're Doing It Wrong

   https://www.youtube.com/watch?v=j3wXitDweC4#t=1411
6. Flickr API Signature Forgery

   http://netifera.com/research/flickr_api_signature_forgery.pdf
7. Don't trust a string based on TryParse or IsNumeric result

   https://soroush.secproject.com/blog/2012/10/dont-trust-a-string-based-on-tryparse-or-isnumeric-result-netvbscript/
8. The PCI Security Standards Council Website

   https://www.pcisecuritystandards.org/
9. Test Card Details for Your Test Transactions

   http://www.sagepay.co.uk/support/12/36/test-card-details-for-your-test-transactions
10. Common Weakness Enumeration

    https://cwe.mitre.org/

