Top 10 Business Logic Attack Vectors

Attacking and Exploiting Business Application Assets and Flaws – Vulnerability Detection to Fix

Whitepaper

RAPID7

Contents

Introduction

Significant attention has been given to codified, groups of standard syntax-based web application attacks such as SQL Injection (SQLi), Blind SQL Injection (BSQLi), and Cross Site Scripting (XSS). These classes of attacks are easier to understand and test for, because they can be broken down into a series of attack types that are somewhat uniform across applications. Another class of attacks, business logic flaws, defy easy categorization and can be more art than science to discover. The purpose of this paper is to give an overview of several types of business logic attacks as well as some tips to pen testers on how to test for these types of vulnerabilities.

Business Scenarios and Issues

As more and more business processes have migrated to web applications, web applications have become the core mechanism for effecting business processes over the Internet.

Before launching a web application into production on the Internet, it is imperative to test it thoroughly from a security standpoint. The testing can be achieved largely through black-box testing. There are two black-box testing approaches:

- Using an automated software to scan for vulnerabilities
- Doing a manual review of the applications security and logic enforcement with human intelligence.

Automated scanners are great at finding syntax issues and can help in discovering injection vectors like SQLi or XSS, but when it comes to business logic flaws, automation has known limitations. Humans are better at identifying critical behavioral patterns. It is important to complement the automated testing process with a human discovery of security risks that can be exploited by manipulating the business logic. Business logic exploits can result in serious compromise of internal and external applications even in applications with safeguards such as, authentication and authorization controls. This paper describes how an application security effort can be more effective by augmenting automated vulnerability assessment solutions with in-depth manual penetration testing techniques. It also details the top 10 business logic attack vectors that humans should look for in their manual complementary tests and gives guidelines for this testing.

Example Business Scenarios

A business process is an action or a set of actions that facilitates how one internal or external entity transacts with another internal or external entity. Automating business processes such as customer purchase orders, banking queries, wire transfers or online auctions, for example, requires entities to have access to extremely sensitive information. The following are some illustrative examples of exploitable business processes.

Retrieving a Profile

For example, Jack's profile can be fetched with id=1001 and if this value changed to 1089 we get another user's information. A scanner may go on and change the value from 1001 to '1001 to find SQL injection, but not to 1089 and would miss deducing that the application is vulnerable to authorization bypass. By changing the "id" from 1001 to 1089, a pen tester can see that John's profile , rather than Jack's, is being displayed.

Shopping Cart

Let us consider an online store application where customers add items to their shopping cart. The application sends the customers to a secure payment gateway where they submit their order. To complete the order, customers are required to make a credit card payment. In this shopping cart application, business logic errors may make it possible for attackers to bypass the authentication processes to directly log into the shopping cart application and avoid paying for "purchased" items.

DEFINING THE USER

- Not just your in-house employees
- Not just execs
- Not just your contractors
- Don't forget administrators—they can specifically be targeted

Modeling a Business Process Flow

In order to test the business logic of a web application, it is necessary to first understand the steps that the application is going through. Outlined below is the modeling of a sample business process flow, i.e. modeling the events that occur to start a process, perform intermediate processes (and sub-processes), and the end results of the process flow.

Example:

Payment transaction flow (Payment Transactions between store and payment service provider)

1. Customer builds a shopping cart. i.e. adds, removes or updates items by browsing through the merchant's offerings and decides to place an order.
2. Customer submits order. Here, the web application invokes the Order capture process. The customer

provides payment and shipping data. The server approves payment with the payment service provider.

3. Payment authorization/approval is completed. The Credit Card authentication process is invoked i.e. credit card transaction approval is submitted for a merchant by the card-issuing bank. Payment transaction is either aborted or completed.

4. The payment is deposited, the order is finalized, the items are released from inventory and the order is shipped to the customer's shipping address.

When manually testing business logic, the tester must consider the possibility of one or all of the intermediate steps can be bypassed? At stake, is a treasure trove of information – the customer's personal and financial details – that is available to an attacker should an application vulnerability that is exploitable, exist. The operative word, here, is "exploitable". If, indeed, the "attacker" is able to exploit the existing vulnerability, the following scenarios are possible:

1. The "attacker" gets direct access to orders that have been received, processed, shipped.

2. The "attacker" gets access to payment transaction details, such as shopper credit card data, merchant information.

In the end, the attacker avoids paying for "purchased" items.

Attack Vectors for Business Logic

The following are 10 of the most common business logic exploits that we test for as a part of manual penetration tests. While the specific tests may vary from application to application, these should give the reader a good starting place as well as a toolkit of ideas for additional testing.

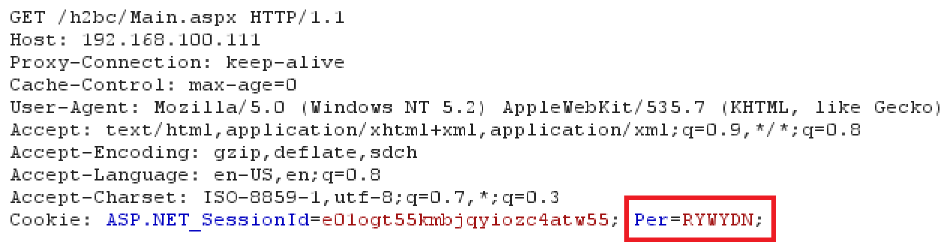
AV1 - Authentication flags and privilege escalations at application layer

Applications have their own access control lists (ACLs) and privileges. The most critical aspect of the application related to security is authentication. An authenticated user has access to the internal pages and structures that reside behind the login section. These privileges can be maintained by the database, LDAP, file etc. If the implementation of authorization is weak, it opens up possible vulnerabilities. If these vulnerabilities are identified during a test, then there is the potential for exploitation. This exploitation would likely include accessing another user's content or becoming a higher-level user with greater permissions to do greater damage.

Example:

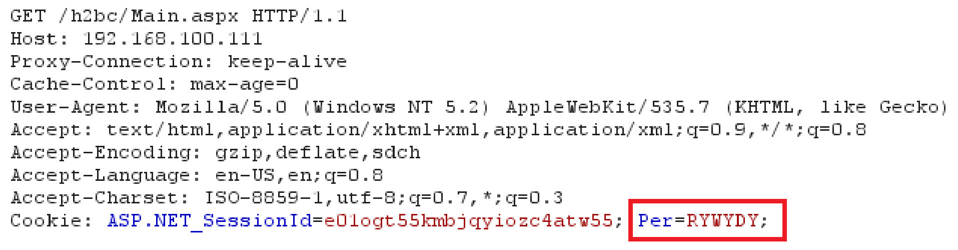
An application is maintaining ACLs and passing them as cookies at the time of authentication. This cookie can be reverse engineered by a malicious attack and an escalation exploit can be crafted. In some cases with Web 2.0 applications, these routine are done in JavaScript which can be easily deciphered in some cases and exploited. If business logic is buried in a Flash or Silverlight application component then it is possible to access the authorization information and attempt to manipulate it into an exploit.

Once you see the names of the parameters and cookies, you can determine if privilege information is being mapped to the parameters. HTTP protocol analysis and application behavior can help in identifying this loophole. This is not only the most common logical problem, but it also relatively easy to identify. For example, as shown below, a cookie flag contains ACL settings which have been defined by the developer. The last right character of the cookie value RYWYDN is "N". If this is changed to "Y", it grants write access on the resource.



```
GET /h2bc/Main.aspx HTTP/1.1
Host: 192.168.100.111
Proxy-Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 5.2) AppleWebKit/535.7 (KHTML, like Gecko)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: ASP.NET_SessionId=e01ogt55kmbjqyiozc4atw55; Per=RYWYDN;
```

Here, manipulated the new request will grant write access.



```
GET /h2bc/Main.aspx HTTP/1.1
Host: 192.168.100.111
Proxy-Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 5.2) AppleWebKit/535.7 (KHTML, like Gecko)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: ASP.NET_SessionId=e01ogt55kmbjqyiozc4atw55; Per=RYWYDY;
```

How to test for this business logic flaw:

- During the profiling phase or through a proxy observe the HTTP traffic, both request and response blocks.
- POST/GET requests would have typical parameters either in name-value pair, JSON, XML or Cookies. Both the name of the parameter and the value need to be analyzed.
- If the parameter name is suspicions and suggests that it has something to do with ACL/Permission then that becomes a target.
- Once the target is identified, the next step is evaluating the value, it can be encoded in hex, binary, string, etc.. The tester should do some tampering and try to define its behavior with bit of fuzzing.
- In this case, fuzzing may need a logical approach, changing bit patterns or permission flags like 1 to 0 or Y to N and so on. Some combination of bruteforcing, logical deduction and artistic tampering will help to decipher the logic. If this is successful then we get a point for exploitation and end up escalating privileges or bypassing authorization.

AV2 - Critical Parameter Manipulation and Access to Unauthorized Information/Content

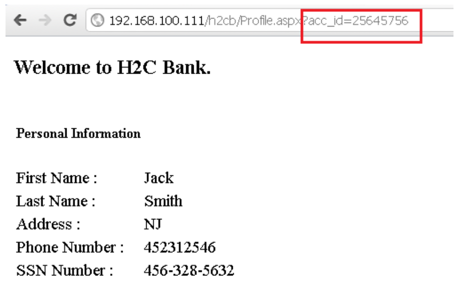
HTTP GET and POST requests are typically accompanied with several parameters when submitted to the application. These parameters can be in the form of name/value pairs, JSON, XML etc. Interestingly, these parameters can be tampered with and guessed (predicted) as well. If the business logic of the application is processing these parameters before validating them, it can lead to information/content disclosure. This is another common business logic flaw that is easy to exploit.

Example:

A banking application is authenticating a user and then allowing the user to access his/her last 10 transactions. While making this request, several parameters are going to the application but one of these parameters is the "accountid" parameter. If this parameter is not truly random and easily guessable, then an attacker can inject another user's account number. At this point, if business logic is not going back and mapping the existing session to the original account that was logged into the application, then the other user's account information gets disclosed. This can be a lethal blow for an application. This can also happen with a shopping cart system where you see your own order based on a parameter called "orderid". By modifying this parameter you may end up seeing another user's orders. Again, parameter analysis and behavior can help in identifying this attack vector.

Here, a customer of a bank opens his/her personal information page. The URL shows that the customer's id is being passed in a query string.

An attacker can tamper with the customer id parameter and can try various permutations of numbers and combinations.



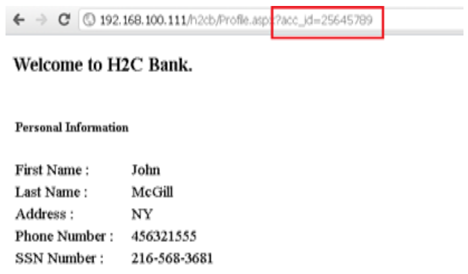
192.168.100.111/h2cb/Profile.aspx?acc_id=25645756

Welcome to H2C Bank.

Personal Information

First Name :	Jack
Last Name :	Smith
Address :	NJ
Phone Number :	452312546
SSN Number :	456-328-5632

Here, you can see that an attacker has obtained access by using a different user's ID and has successfully guessed a correct customer id (other than their own). Hence, the attacker can now access other user accounts based on simple guesses or brute force techniques.



192.168.100.111/h2cb/Profile.aspx?acc_id=25645789

Welcome to H2C Bank.

Personal Information

First Name :	John
Last Name :	McGill
Address :	NY
Phone Number :	456321555
SSN Number :	216-568-3681

How to test for this business logic flaw:

- During the profiling phase or through a proxy, observe HTTP traffic, both request and response blocks.
- POST/GET requests would have typical parameters either in name-value pair, JSON, XML or Cookies. Both the name of the parameter and the value need to be analyzed.
- Observe the values in the traffic and look for incrementing numbers and easily guessable values across all parameters.
- This parameter's value can be changed and one may gain unauthorized access.

In the above case we were able to access other users profiles.

AV3 - Developer's cookie tampering and business process/logic bypass

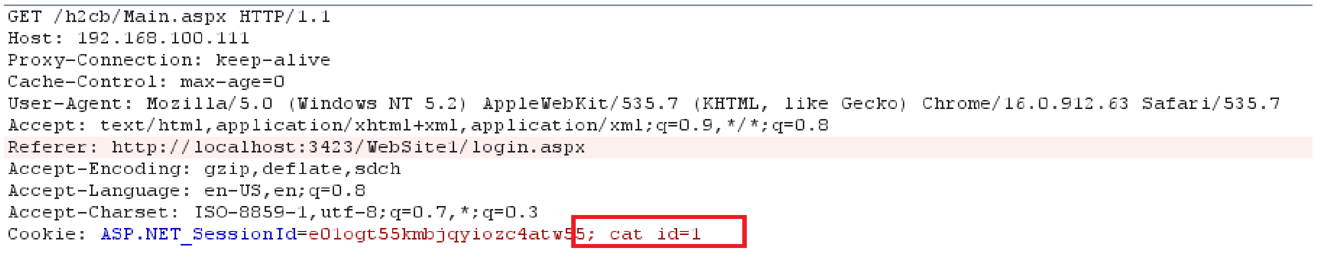
Cookies are an essential component to maintain state over HTTP. In many cases, developers are not using session cookies only, but instead are building data internally using session only variables. Application developers set new cookies on the browser at important junctures which exposes logical holes. After authentication logic sets several parameters based on credentials, developers have two options to maintain these credentials across applications. The developer can set the parameters in session variables or set cookies in the browser with appropriate values. If application developers are passing cookies, then they might be reverse engineered or have values that can be guessed/deciphered. It can create a possible logical hole or bypass. If an attacker can identify this hole then they can exploit it with ease.

Example:

A trading portal allows a user to log in and then set a few values as cookies on the browser. One of these cookies is storing the user's membership category as silver, gold or platinum.

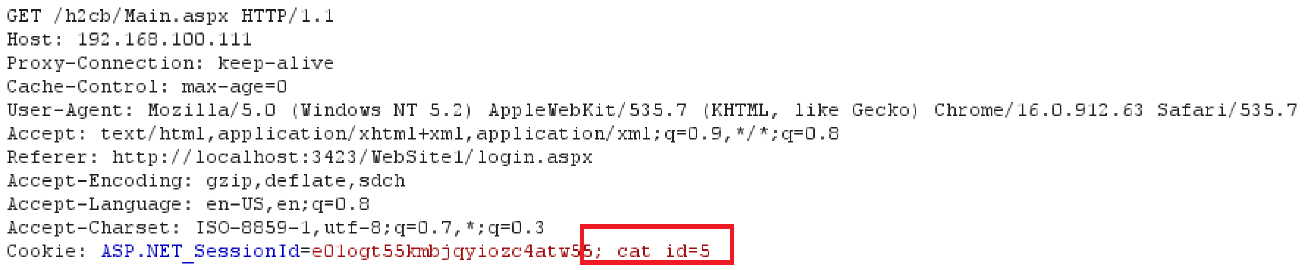
Now this cookie defines the trading limit on the application layer. If an attacker deduces this information and manipulates this cookie, then the attacker will be able to upgrade their membership. There are several scenarios where these cookies can be exploited. If the application is giving more than one cookie then it is maintaining some logic based on cookie and not relying truly on session cookie only. This scenario is becoming increasingly complex with "single sign on" where cookies are accessible and applicable across domains.

Again, here is a cookie that you may end up deciphering because it is too simple and similar.



```
GET /h2cb/Main.aspx HTTP/1.1
Host: 192.168.100.111
Proxy-Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 5.2) AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.63 Safari/535.7
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost:3423/WebSite1/login.aspx
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: ASP.NET_SessionId=e0logt55kmbjqyiozc4atw55; cat id=1
```

Again, here is a cookie that you may end up deciphering because it is too simple and similar.



```
GET /h2cb/Main.aspx HTTP/1.1
Host: 192.168.100.111
Proxy-Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 5.2) AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.63 Safari/535.7
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost:3423/WebSite1/login.aspx
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: ASP.NET_SessionId=e0logt55kmbjqyiozc4atw55; cat id=5
```

Simple sequential values are changed. Developer cookies are vulnerable to few logical attack vectors.

How to test for this business logic flaw:

- During the profiling phase or through a proxy observe the HTTP traffic, both request and responseblocks.
- Analyze all cookies delivered during the profiling, some of these cookies will be defined by developers and are not session cookies defined by the web application server.
- Observe cookie values in specific, look for incrementing easily guessable values across all cookies.
- Cookie value can be changed and one may gain unauthorized access or logical escalation.

AV4 - LDAP Parameter Identification and Critical Infrastructure Access

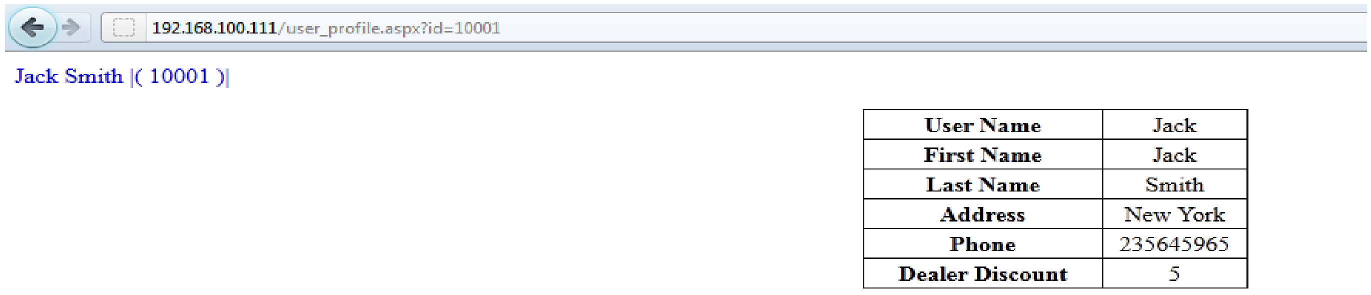
LDAP is becoming an important aspect for large applications and it may get integrated with "single sign on" as well. Many infrastructure layer tools like Site Minder or Load Balancer use LDAP for both authentication and authorization. LDAP parameters can carry business logic decision flags and those can be abused and leveraged. LDAP filtering being done at the business application layer enable logical injections to be possible on those parameters. If the application is not doing enough validation then LDAP injection and business layer bypasses are possible.

Example:

A large telecom company is leveraging LDAP services for their authentication and authorization on large infrastructure applications and users can login from mobile devices, tablets, and traditional web browsers. LDAP is also integrated into the company's web services for business-to-business communication. The LDAP parameter uses "email id" to authenticate and pass it to an LDAP server with other tree nodes (ON, CN etc.). This parameter allows rights for read/write/delete based on rules defined on the LDAP server. The application is allowing "*" and that ends up giving full access.

Because of this full permission set, the attack can go ahead and change their settings. For example, a user can change their access plans, billing information etc. This can be a big hole at the application layer. Since LDAP is being used on more than one application, all of the applications can be attacked and exploited using LDAP bypasses. It is relatively common with infrastructure related applications and intranet apps. For example, over here we get Jack's profile and a filter is passing a name to an underlying LDAP request.

For example, over here we get Jack's profile and a filter is passing a name to an underlying LDAP request.

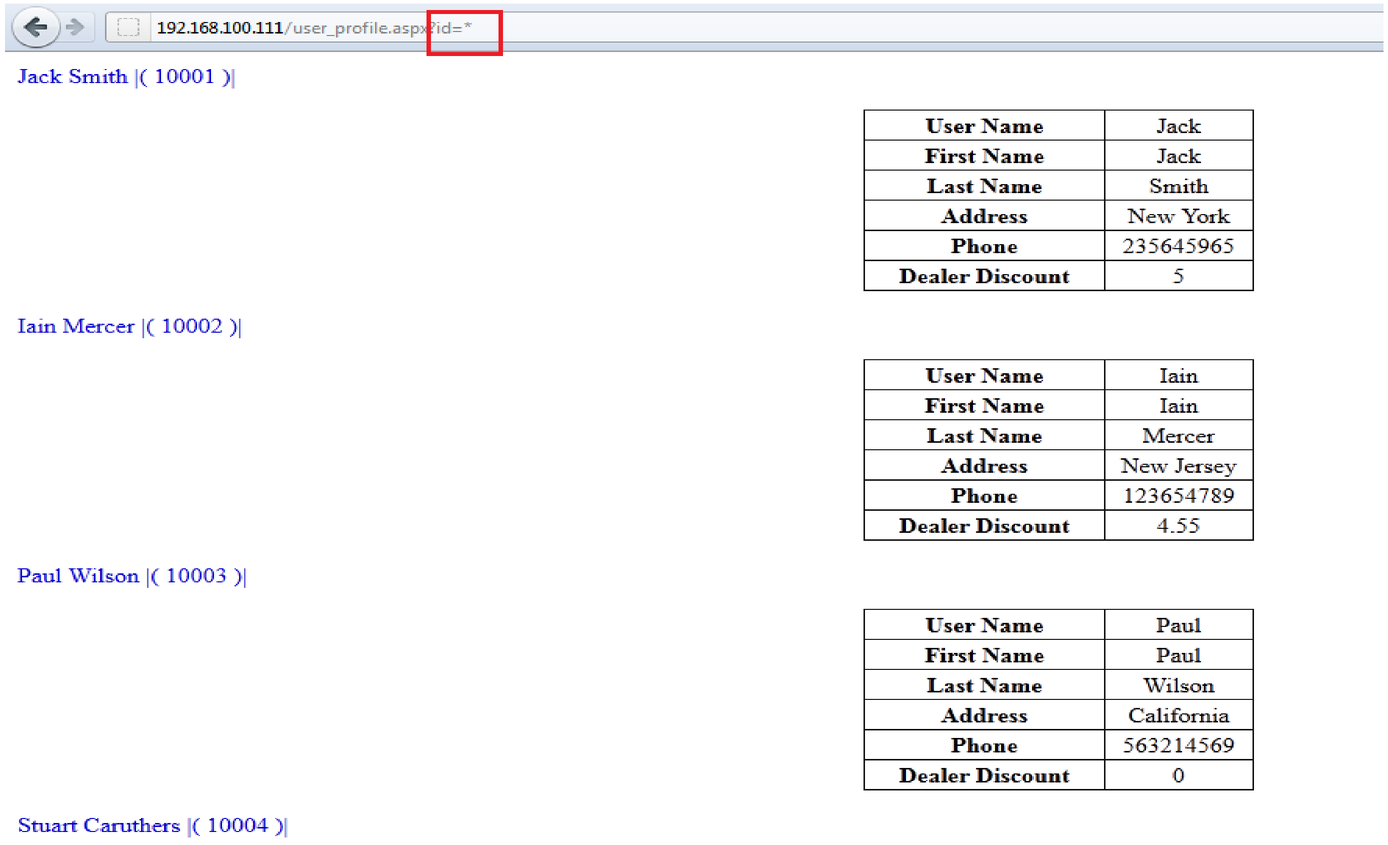


192.168.100.111/user_profile.aspx?id=10001

Jack Smith |(10001)|

User Name	Jack
First Name	Jack
Last Name	Smith
Address	New York
Phone	235645965
Dealer Discount	5

This can be bypassed if validation is not in place at the business logic layer of the application and "*" gets injected in to the LDAP filter.



192.168.100.111/user_profile.aspx?id=*

Jack Smith |(10001)|

User Name	Jack
First Name	Jack
Last Name	Smith
Address	New York
Phone	235645965
Dealer Discount	5

Iain Mercer |(10002)|

User Name	Iain
First Name	Iain
Last Name	Mercer
Address	New Jersey
Phone	123654789
Dealer Discount	4.55

Paul Wilson |(10003)|

User Name	Paul
First Name	Paul
Last Name	Wilson
Address	California
Phone	563214569
Dealer Discount	0

Stuart Caruthers |(10004)|

How to test for this business logic flaw:

- During the profiling phase or through a proxy observe the HTTP traffic, both request and response blocks.
- POST/GET requests would have typical parameters either in name-value pair, JSON, XML or Cookies. Both the name of the parameter and the value need to be analyzed.
- Analyze parameters and their values, look for ON,CN,DN etc. Usually these parameters are linked with LDAP. Also look for the parameter taking email or usernames, these parameters can be prospective targets.
- These target parameters can be manipulated and injected with "*" or any other LDAP specific filters like OR, AND etc. It can lead to logical bypass over LDAP and end up escalating access rights.

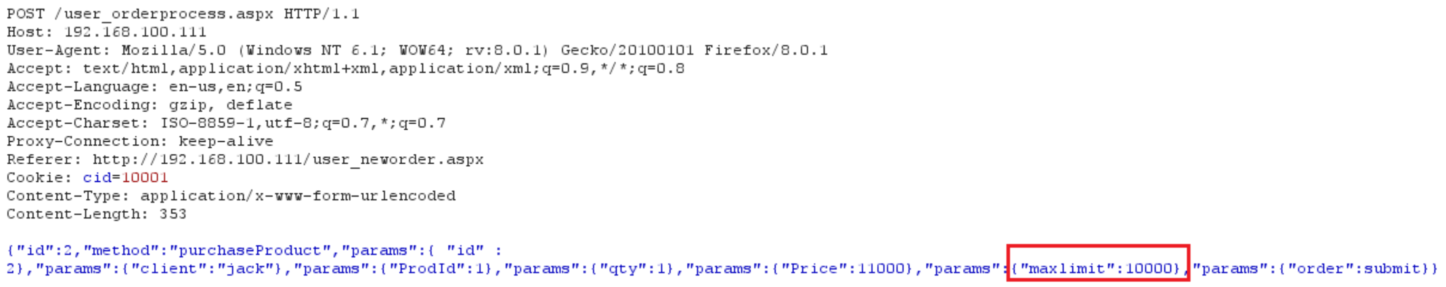
AV5 - Business Constraint Exploitation

The application's business logic should have defined rules and constraints that are very critical for an application. If these constraints are bypassed by an attacker, then it can be exploited. User fields that have poor design or implementation are often controlled by these business constraints. If business logic is processing variables controlled as hidden values then it leads to easy discovery and exploitation. While crawling and profiling the application, one can list all these possible different values and their injection places. It is easy to browse through these hidden fields and understand their context; if context is leveraged to control the business rules then manipulation of this information can lead to critical business logic vulnerabilities.

Example:

A trading application is controlling a monetary upper limit for each of their users. Users are limited to this amount and cannot create transactions above this limit. If the application controlling the limit amount as part of a hidden value and is passing this information to JSON then it is controlled by the user and an attacker can manipulate this amount and bypass this constraint to manipulate the limit to be higher than what is allocated. This leads to an exploitable vulnerability. In some cases, it is possible to do an integer overflow and reverse transactions as well. If the application allows the user to place an order for a negative quantity then we test to see what will happen. Will the attacker get paid by the shop and will our credit card be credited instead of charged?

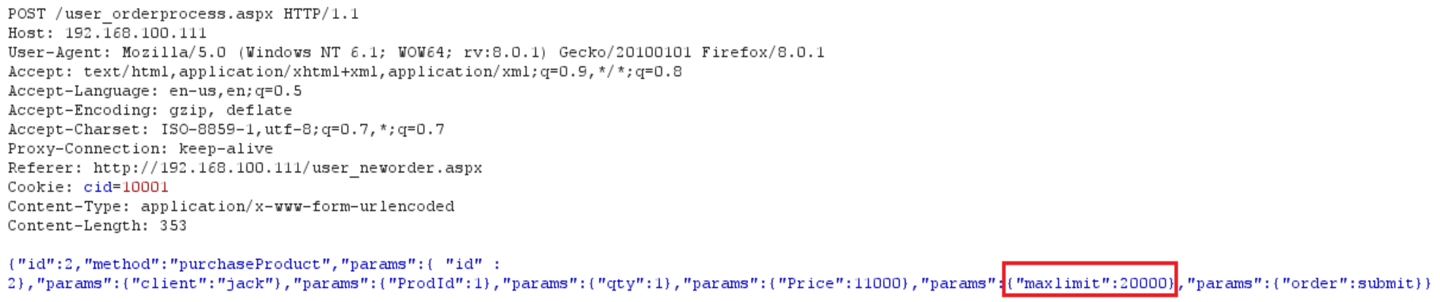
Here is an example where a JSON layer call is injected with tampered limit.



```
POST /user_orderprocess.aspx HTTP/1.1
Host: 192.168.100.111
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0.1) Gecko/20100101 Firefox/8.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Proxy-Connection: keep-alive
Referer: http://192.168.100.111/user_neworder.aspx
Cookie: cid=10001
Content-Type: application/x-www-form-urlencoded
Content-Length: 353

{"id":2,"method":"purchaseProduct","params":{ "id" :
2},"params":{"client":"jack"},"params":{"ProdId":1},"params":{"qty":1},"params":{"Price":11000},"params":{"maxlimit":10000},"params":{"order":submit}}
```

Limit is controlled by variable which is hidden in JSON.



```
POST /user_orderprocess.aspx HTTP/1.1
Host: 192.168.100.111
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0.1) Gecko/20100101 Firefox/8.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Proxy-Connection: keep-alive
Referer: http://192.168.100.111/user_neworder.aspx
Cookie: cid=10001
Content-Type: application/x-www-form-urlencoded
Content-Length: 353

{"id":2,"method":"purchaseProduct","params":{ "id" :
2},"params":{"client":"jack"},"params":{"ProdId":1},"params":{"qty":1},"params":{"Price":11000},"params":{"maxlimit":20000},"params":{"order":submit}}
```

How to test for this business logic flaw:

- During the profiling phase or through a proxy observe the HTTP traffic, both the request and response blocks.
- POST/GET requests would have typical parameters either in name-value pair, JSON, XML or Cookies. Both the name of the parameter and the value need to be analyzed.
- Analyze hidden parameters and their values, look for business specific calls like transfer money, max limit etc. All these parameters which are dictating a business constraint can become a target.
- These target parameters can be manipulated and values can be changed. It is possible to avoid the business constraint and inject an unauthorized transaction.

AV6 – Business Flow Bypass

Applications include flows that are controlled by redirects and page transfers. After a successful login, for example, the application will transfer the user to the money transfer page. During these transfers, the user's session is maintained by a session cookie or other mechanism. In many cases, this flow can be bypassed which can lead to an error condition or information leakage. This leakage can help an attacker identify critical back-end information. If this flow is controlling and giving critical information out then it can be exploited in various use cases and scenarios.

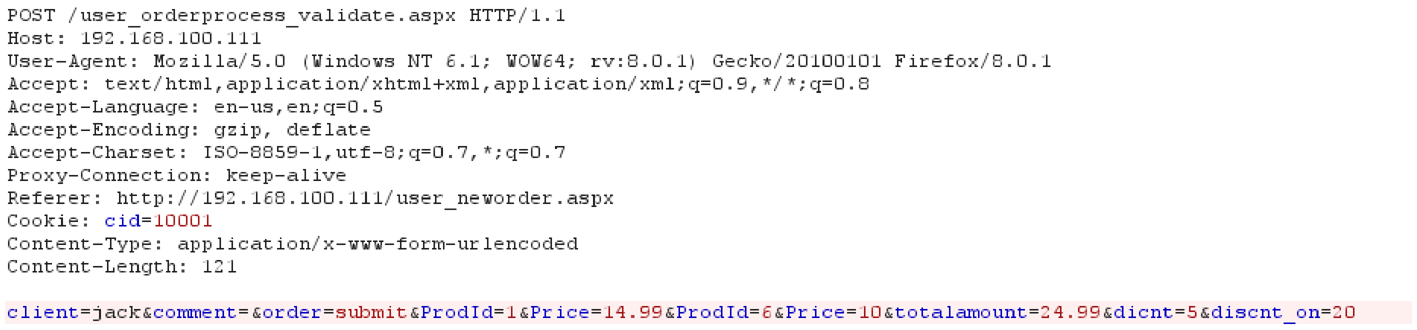
Example:

An airline ticketing service has a seat allocation process that requires three steps. In the last step, the application sends confirmation to the user. After studying the flow, the user figures out a way to bypass step two and go directly to step three. While the application attempts to pass the seat allocation information in step three, the user can inject an upgraded seat assignment in business class. The application does not detect the injection and assumes the user is following the proper steps and allocates a seat in business class.

Similarly, a bank application does a three step process for a wire transfer and stores information about the session. It then passes a valid token as the last step and expects the browser to return that token. Now, the user goes back to a previous step and changes the value that eventually changes the values on the server side session after the post. This enables an injection in the final step which already has a validated token. In this case the transaction will go through successfully.

Here is a simple step process for placing an order.

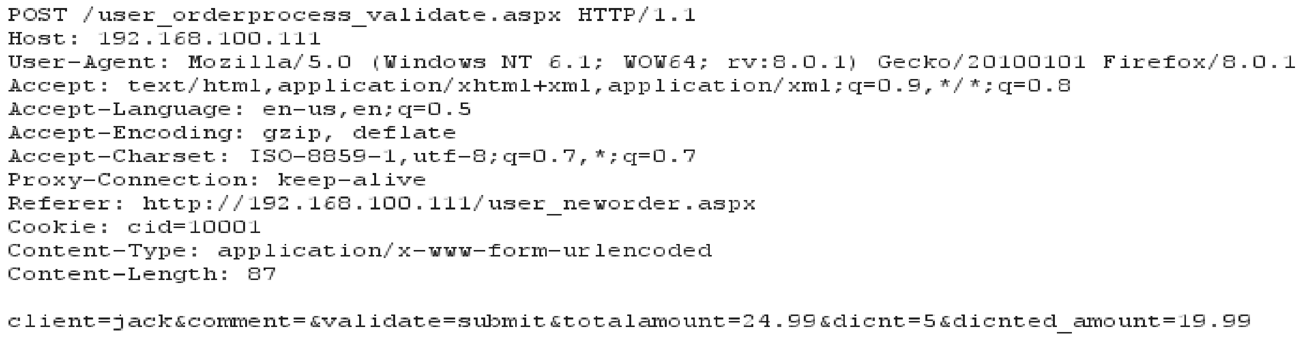
Step 1 – Selecting Items



```
POST /user_orderprocess_validate.aspx HTTP/1.1
Host: 192.168.100.111
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0.1) Gecko/20100101 Firefox/8.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Proxy-Connection: keep-alive
Referer: http://192.168.100.111/user_neworder.aspx
Cookie: cid=10001
Content-Type: application/x-www-form-urlencoded
Content-Length: 121

client=jack&comment=&order=submit&ProdId=1&Price=14.99&ProdId=6&Price=10&totalamount=24.99&dicnt=5&discnt_on=20
```

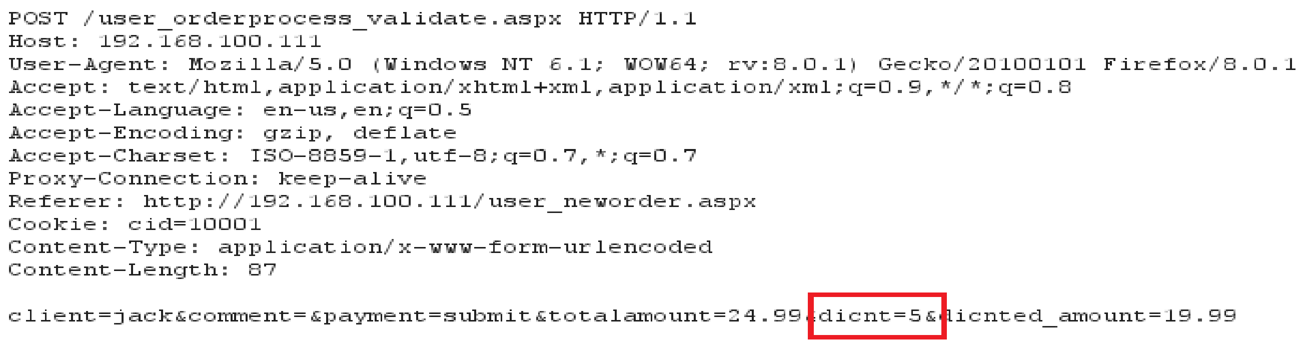
Step 2 - Finalizing Items (Multiple Selections)



```
POST /user_orderprocess_validate.aspx HTTP/1.1
Host: 192.168.100.111
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0.1) Gecko/20100101 Firefox/8.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Proxy-Connection: keep-alive
Referer: http://192.168.100.111/user_neworder.aspx
Cookie: cid=10001
Content-Type: application/x-www-form-urlencoded
Content-Length: 87

client=jack&comment=&validate=submit&totalamount=24.99&dicnt=5&dicnted_amount=19.99
```

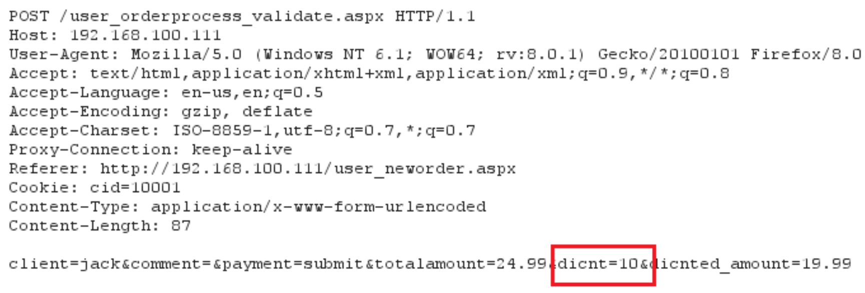
Step 3 – Some Parameters Controlling Discounts Hidden To The User



```
POST /user_orderprocess_validate.aspx HTTP/1.1
Host: 192.168.100.111
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0.1) Gecko/20100101 Firefox/8.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Proxy-Connection: keep-alive
Referer: http://192.168.100.111/user_neworder.aspx
Cookie: cid=10001
Content-Type: application/x-www-form-urlencoded
Content-Length: 87

client=jack&comment=&payment=submit&totalamount=24.99&dicnt=5&dicnted_amount=19.99
```

Here is a bypass, one can change this parameter and obtain a greater discount.



```
POST /user_orderprocess_validate.aspx HTTP/1.1
Host: 192.168.100.111
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0.1) Gecko/20100101 Firefox/8.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Proxy-Connection: keep-alive
Referer: http://192.168.100.111/user_neworder.aspx
Cookie: cid=10001
Content-Type: application/x-www-form-urlencoded
Content-Length: 87

client=jack&comment=&payment=submit&totalamount=24.99&dicnt=10&dicnted_amount=19.99
```

How to test for this business logic flaw:

- During the profiling phase or through a proxy observe the HTTP traffic, both request and response blocks.
- POST/GET requests would have typical parameters either in name-value pair, JSON, XML or Cookies. Both the name of the parameter and the value need to be analyzed.
- Identify business functionalities which are in specific steps (e.g. a shopping cart or wire transfer).
- Analyze all steps carefully and look for possible parameters which are added by the application either using hidden values or through JavaScript.
- These parameters can be tampered through a proxy while making the transaction. This disrupts the flow and can end up bypassing some business constraints.

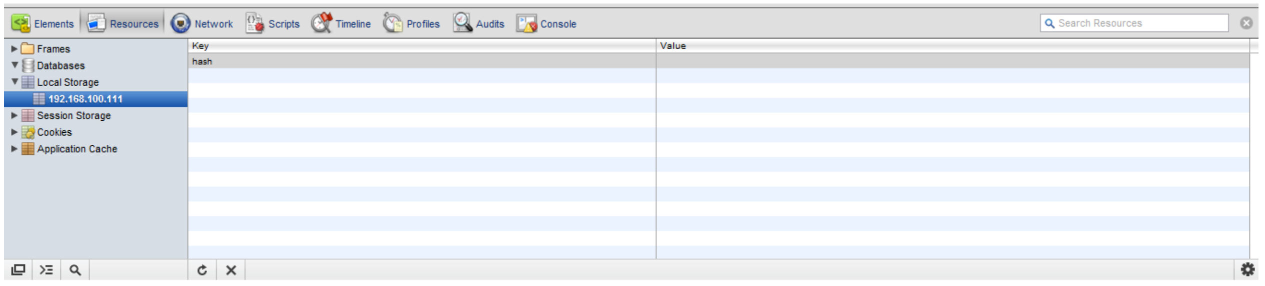
AV7 - Exploiting Clients Side Business Routines Embedded in JavaScript, Flash or Silverlight

Many business applications are now running on rich internet application (RIA) frameworks leveraging JavaScripts, Flash, and Silverlight. In many cases, the logic is embedded in the client side component. These components can be reverse engineered. If it is in Flash or Sliverlight, both of these files can be decompiled and the actual logic used by the application can be discovered. The application running on the client side using JavaScript can be debugged line by line to identify embedded logic. This includes any client side logic to implement algorithms for cryptography, credential storage, privilege management etc. Once the client side code has been reverse engineered, it can be exposed attacks that lead to further exploitation.

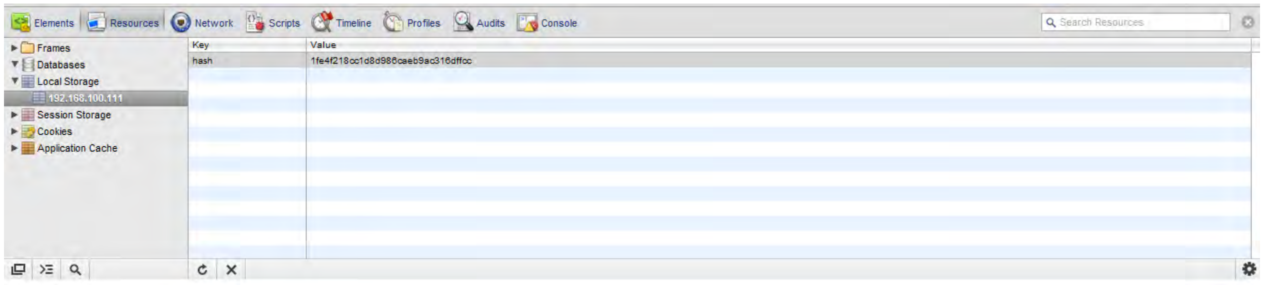
Example:

A banking application maintains critical information on the client side in the browser and sends information at the time of requirement. This information is being used at the time the transaction is performed. The browser makes a call to the application and sends encrypted code back. This code is created based on his/her account number. The logic of encrypting this code is created in JavaScript and one can actually reverse engineer the call. This allows an attacker to look at another user's information by simply guessing their account numbers. The same logic is possible with Silverlight and Flash files as well.

Here is an example where a user's hash being maintained on client side on localStorage. This hash is a simple MD5 hash of user's name.



Key	Value
hash	



Key	Value
hash	1fe4f218cc1d8d986caeb9ac316dffcc

Now that we know the hash type and value used by the application, the above value can be changed and used to hijack another user's account.

How to test for this business logic flaw:

- Once page is loaded in the browser one needs to analyze DOM using firebug or any other similar tool.
- Identify all variables residing on browser stack.
- Look for suspicious values and parameters.
- One can manipulate these values inside DOM and replay the HTTP request. If business logic is trusting only on client side then it may end up giving unauthorized access.

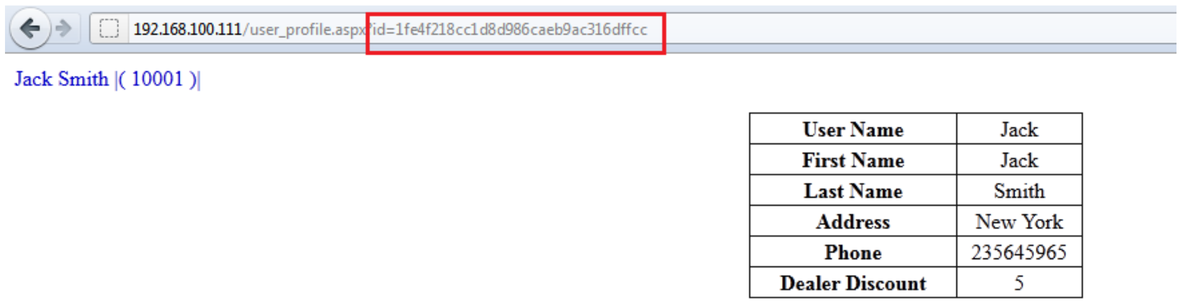
AV8 - Identity or Profile Extraction

A user's identity is one of the most critical parameters in authenticated applications. The identities of users are maintained using session or other forms of tokens. Poorly designed and developed applications allow an attacker to identify these token parameters from the client side and in some cases they are not closely maintained on the server side of the session as well. This scenario opens up a potential opportunity for abuse and system wide exploitation. The token is either using only a sequential number or a guessable username.

Example:

Let's assumes a social networking site running on a multi-domain and platform is doing "single sign on" and maintaining a unique session token for authentication and authorization. There is a server called Login where the user information is stored. When the user clicks on their profile, all requests are redirected to the Login server and by design, it passes an account number that is buried deep down in an XML document. It is easy to tamper and guess another user's number. If an HTTP request is made with this number, it fetches other user's profile. An attacker can open a dummy account and exploit this hole to harvest several users' critical information from their profiles.

Jack's profile accessed using following URL.

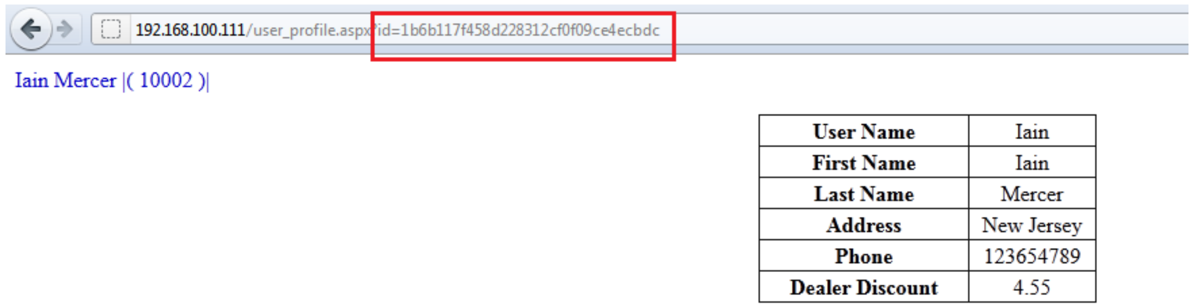


192.168.100.111/user_profile.aspx?id=1fe4f218cc1d8d986caeb9ac316dffcc

Jack Smith |(10001)|

User Name	Jack
First Name	Jack
Last Name	Smith
Address	New York
Phone	235645965
Dealer Discount	5

Now an attacker can access other user's profile by guessing encryption/hashing logic.



192.168.100.111/user_profile.aspx?id=1b6b117f458d228312cf0f09ce4ecbdc

Iain Mercer |(10002)|

User Name	Iain
First Name	Iain
Last Name	Mercer
Address	New Jersey
Phone	123654789
Dealer Discount	4.55

How to test for this business logic flaw:

- During the profiling phase or through particular proxy observe HTTP traffic, both request and response blocks.
- POST/GET requests would have typical parameters either in name-value pair, JSON, XML or Cookies. Both name of parameter and value need to be analyzed.
- Look for parameters which are controlling profiles.
- Once these target parameters are identified, one can decipher, guess or reverse engineer tokens. If tokens are guessed and reproduced – game over!

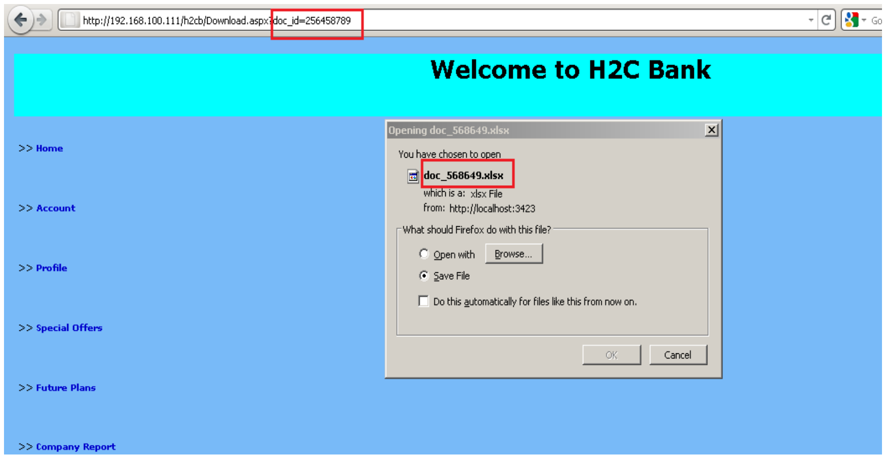
AV9 - File or Unauthorized URL Access and Business Information Extraction Identity

Business applications contain critical information in their features, in the files that are exported and in the export functionality itself. A user can export their data in a selected file format (PDF, XLS or CSV) and download it. If this functionality is not carefully implemented, it can enable asset leakage. An attacker can extract this information from the application layer. This is one of the most common mistakes and easy to exploit as well. These files can be fetched directly from URLs or by using some internal parameters.

Example:

A banking application allows users to fetch their monthly transactions in CSV or XLS format. These files are created when a request for the monthly transaction list is made and a temporary link is created for downloading the information, which is given to the user to download the statement. An attacker can analyze this mechanism and identify the hidden call. If this call is not well guarded with proper authorization, then it leads to possible compromise. An attacker can analyze the file naming convention and start guessing for another users' URLs and extract this information from the application. This flaw can prove very deadly since it leads to privacy and security concerns for the application owner.

Here we have a URL to download a document. Clearly, by guessing a user ID, an attacker can find other documents residing on the server.



```
http://192.168.100.111/h2cb/Download.aspx?doc_id=256458789
```

Welcome to H2C Bank

Opening doc_568649.xlsx

You have chosen to open

doc_568649.xlsx

which is a: xlsx File

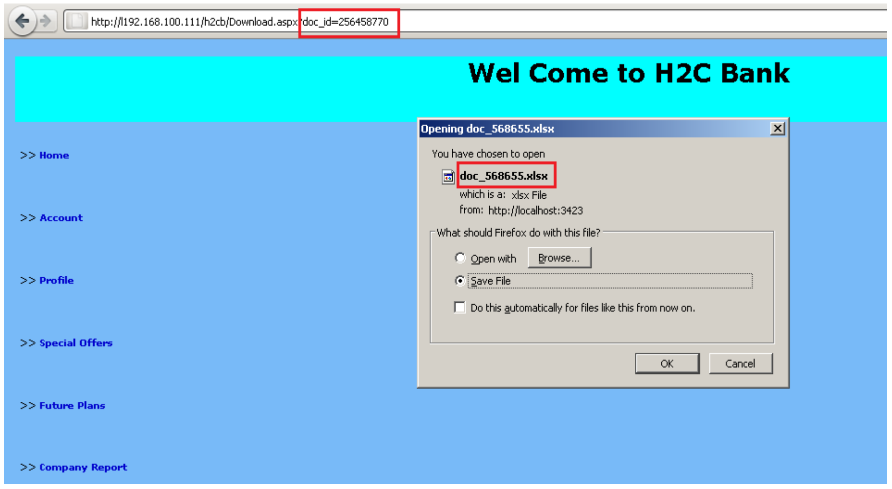
from: http://localhost:3423

What should Firefox do with this file?

Open with [✓ Save File]

Do this automatically for files like this from now on. ☐

Fetching another user's document from application.



```
http://l192.168.100.111/h2cb/Download.aspx?doc_id=256458770
```

Wel Come to H2C Bank

Opening doc_568655.xlsx

You have chosen to open

doc_568655.xlsx

which is a: xlsx File

from: http://localhost:3423

What should Firefox do with this file?

Open with [✓ Save File]

Do this automatically for files like this from now on. ☐

How to test for this business logic flaw:

- During the profiling phase or through a particular proxy, observe the HTTP traffic, both request and response blocks.
- POST/GET requests would have typical parameters either in a name-value pair, JSON, XML or Cookie. Both the name of parameter and value need to be analyzed.
- Identify file call functionalities based on parameter names like file, doc, dir etc. These parameters will point you to possible unauthorized file access vulnerabilities.
- Once a target parameter has been identified start doing basic brute force or guess work to fetch another user's files from server.

AV10 – Denial of Services (DoS) with Business Logic

Denial of Service (DoS) vulnerabilities are very costly for business applications because they bring down the application for a significant amount of time or at a critical juncture. Many features, if not implemented correctly, can lead to a DoS condition where the attacker can identify a loophole and try to exploit it. In some cases, the design itself creates the possibility of a DoS attack, and in other cases, race conditions force a DoS vulnerability. There are no universal DoS attacks like TCP flooding on networking at the application layer, but it depends on the features and the applications. In some cases, infinite loops implemented in the application layer can force a DoS attack.

Example:

- Airline systems provide a window of three minutes before you choose a seat or buy a ticket. During these three minutes, an attacker can put many tickets on hold at the same time without having the intention of buying any tickets. This scenario leads to a DoS for an actual user who wants to buy a ticket since all seats are on hold for those three minutes. The actual buyer gets a message that the flight is full and the attackers causes a DoS.
- An auction site blocks a user for 24 hours after three incorrect attempts. If two users bid on the same product before the last three minutes, a malicious user can complete three attempts on behalf of the opponent user and lock his/her account, The malicious user then logs-in and places the final bid. The competing user will never get a chance to log in since their account is locked. It is clear case of DoS. A lot of business functionalities allows this type of DoS attack and it is important to put them on a threat model and provide defense at application layer.

Conclusion

Business logic flaws are difficult to identify and discover. These flaws are unique to each application and must be discovered by manual testing. This paper is intended as a starting point to assist penetration testers with looking for these flaws as a part of their security reviews. Rapid7 also offers, as a part of its AppSpider OnDemand add on services,, manual testing for business logic flaws in web applications.

About Rapid7