Designing a Framework Method for Secure Business Application Logic Integrity in e-Commerce Systems

Faisal Nabi
E-Commerce Security Research Group, Hazraat Baba Bullah Shah Research Center, Shaihar Qasoor, Pakistan (Email: nabifaisal@yahoo.com)

(Received Nov. 13, 2009; revised and accepted Feb. 17, 2010)

Abstract

Currently e-commerce system security focuses on mechanisms such as secure transactional protocols, cryptographic schemes, parameter sanitization and it is assumed that putting these in place will guarantee a secure e-Commerce application. However, often vulnerabilities in the business application logic itself are often ignored that can make the effect of these security mechanisms null and void. Essentially, the weakest link can be at the server rather the client and ignoring this is done at a developer's peril. This paper focuses on this weakest link in e-commerce system. In particular, it considers component-based middleware platforms where vulnerabilities may exist in the middleware itself or the components used to construct the e-Commerce application. We outline a logic attacks that would not be prevented by the deployment of the mechanisms commonly used in e-Commerce systems. To counter this problem, we present a secure framework method based on existing techniques that treats security as a first-class concept and considers its interaction with business logic.

Keywords: CBS, design flaws, e-commerce system, integrity, logical attacks, logical flaws, software flaws

1 Introduction

The advent of e-Commerce ushered in a new period pervaded by sense of boundless excitement and opportunities. However, there is always an inverse relationship between return and risk implying that e-Commerce introduces more and newer risks than traditional bricks and mortar commerce. In particular, e-Commerce provides a storefront on the Internet and this makes such businesses vulnerable to hackers. Developers often rely upon vendors of e-Commerce systems providing security mechanisms such as SSL or TLS to protect against attacks. However, reliance on such mechanisms is often insufficient [4, 5, 18].

The real security risks of e-commerce security is more than secure transactional protocols, cryptographic schemes/techniques, parameter santisation, intrusion detection systems etc. [21]. These attributes make up only some part of security and, privacy of e-commerce [12]. The software that executes on the either end of the transaction-server-side or client-side software poses real threats to the security and privacy in e-commerce systems. Two familiar adages play an important role in understanding to secure e-commerce systems:

1) A chain is only as strong as its weakest link;

2) In the presence of obstacles, the path of least resistance is always the path of choice [6]. Although, the security issues of the front-end and back-end software systems in e-commerce application warrant equal attention for complete security in e-commerce but we are more concerned with the often neglected back-end side of security. In particular, component-based middleware business applications.

Section 2 provides an overview of the role of application-level business logic in a Web application. Section 3 introduces component-based software for application development. Section 4 discusses risks in developing component-based business applications. Section 5 outlines the notions of software bugs, flaws and their relationship to software vulnerabilities. Section 6 discusses the concept of a logical vulnerability. Section 7 introduces business logic attacks. Section 8 proposes a classification. Section 9 highlights some case studies. Section 10 proposes a secure framework method for mitigating the existence of flaws in the business software that may lead to expression of these attacks. Finally, Section 11 wraps up the paper with a discussion of the approach.

2 Application Business Logic

The business logic describes the steps required to complete or perform a particular action as defined by the application developer. This is also called business logic because it defines the business rules in e-commerce system at middle tier. The execution of business logic causes the state of the business to evolve over time [9, 10]. Business logic is the implementation of business rules and is defined by a developer and implemented as code [9, 10].

Figure 1 shows the role of business logic in a traditional Web 1.0 style Web application. The user uses a browser to interact with the Web application. The browser forwards requests and receives pages to render in response. The requests are handled by Web servers that delegate requests for the application to the middle-tier application servers where the business logic is implemented. The implementation may be developed using various middleware as shown in the diagram. The business logic will use the middleware to communicate with back-end services such as data storage in the form of databases, other systems such as an Enterprise Resource Planning system (ERP system) or perhaps legacy applications.

The business application logic represents the functions or services that a particular e-commerce site provides. As a result, a given site may often employ custom-developed logic. As the demand for e-commerce services grows, the sophistication of the business application logic grows accordingly [6, 12].

3 Component-based Software Role in Business Logic and Concerns

Component-based software is constructed of components. These are objects that encapsulate behavior and state, are accessed through well-defined interfaces, and are composable with other objects and to provide this conform to some standard. Frameworks for component-based software development will provide tools and code that implement the standard. Examples of such frameworks are JavaBeans, COM, DCOM and CORBA [6]. The great advantage of component based development is the opportunity to reuse industrial-strength software in order to rapidly prototype business application logic colorred(Q. Cheng, J.Yao & R.Xing 2006).

One of the more popular component frameworks for e-commerce application is the server-side Enterprise Java Beans architecture for distributed applications. Other component-based technologies include the common object request broker architecture (CORBA) version 3, an open standard developed by the Object Management Group (OMG) and the Microsoft Distributed Common Object Model/.Net environment.

The component frameworks are the glue that enables software components to provide services, business application logic and provides standard infrastructure services such as naming, persistence, introspection and event handling [12]. The business application logic is coded in software "Components" that can be "Custom-Developed or purchased Commercial-off-the-shelf" [12]. Component-based software is expected to enable distributed B2B applications over the internet and as that off-the-shelf standard business application logic components will be available for purchase.

Components developed using component frameworks execute within application servers that form the middle tier of an n-tier architecture. For example, JBoss is a popular application server for the EJB framework and can load EJB standards compliant beans. The application servers also provide an interface for the business application logic to back-end services such as database management, enterprise resource planning (ERP) and legacy software system services [7, 12]. There is no doubt that component-based software provides numerous benefits, but it poses security hazards similar to the Common Gateway Interface scripts that used to dominate e-Commerce. For example, components generally execute with all rights and privileges of server process and a poorly written component may subject to malformed input causing a buffer overflow. This would allow an attacker to exploit the server side and use the rights allocated to the application server for their own purposes.

One reason for the emergence of components-based software on e-commerce sites is the complexity of the software necessary to implement business application logic. This complexity, in turn, introduces more software flaws that can be exploited for malicious gain.

4 Web Software Application and Component-based Development Risks

Modern Web applications run large scale software applications for e-commerce, information distribution, entertainment, collaborative research work, surveys and numerous other activities. They run on a variety of networked hardware platforms. The software that powers Web applications is distributed, is implemented in multiple languages and styles, incorporates much reuse of custom developed and third-party components and must interface with users, other Web sites and databases. Although. Server-side components are relatively new to the component market. Benefits enable the developer to provide solutions that run on a per server basis. These components serve many clients simultaneously without significant performance loss. Server-side components can also be upgraded efficiently removing the complexities of updating potentially thousands of desktop machines. Component logic is often run on powerful servers as opposed to a desktop machine [14]. This makes the server-side component an excellent candidate for systems that require efficient throughput and performance [15]. The word "heterogeneous" is often used for Web software, it applies in so many ways that the synonymous term "diverse" is more general, familiar and probably more appropriate [13]. The software components are often distributed geographically both during the development and deployment (diverse distribution), communicate in numerous

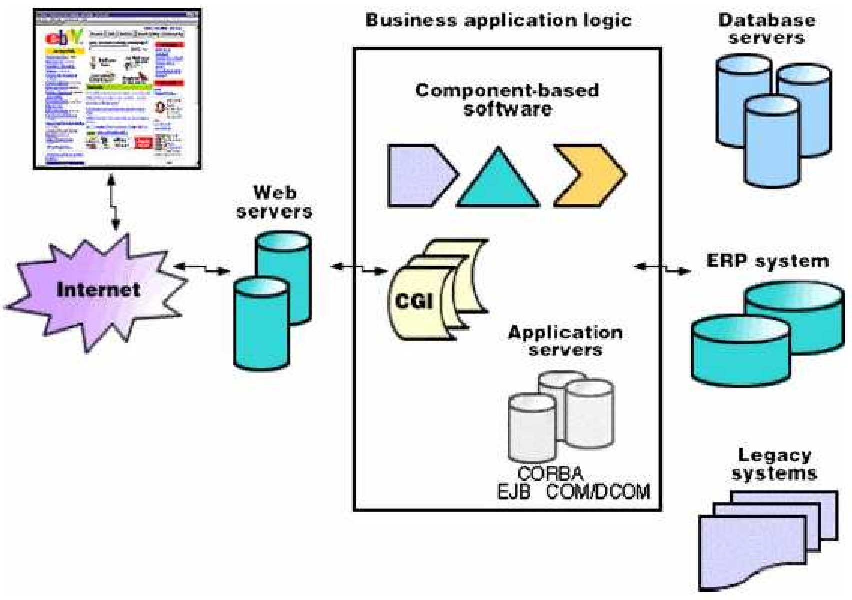


Figure 1: Middle tier of e-commerce servers that implements the business application logic

distinct and sometimes novel ways (diverse communication) [2].

Web-based software systems by integrating numerous diverse components from disparate sources, including custom-built special-purpose applications, customized "Commercial off-the-shelf Software Components & third-party products" [13]. Much of the new complexity found with web-based applications also results from how the different Software components are integrated. Not only is the source unavailable might be hosted on computers at remote, even competing organization. To ensure high quality for the Web systems composed of very loosely coupled components, which seriously required evaluate these Components connections [3].

Web software Components coupled more loosely than any previous software application [13].

The web's function and structure have changed drastically will continue to do so. This was even observed back in 2002 [13]. Examples of such a change in last couple of years idea is the use of Web 2.0 feature Ajax. The Ajax engine is a framework for writing client-side code that handles calls between the client and server. Typically this would implemented as a library of JavaScript functions included on the page [16].

The great advantage of Ajax is that it can be used to provide a more responsive Web application because some tasks can be carried out at the client side without needing to interact with the server. Dangers arise when critical business logic is implemented on the client side where there is little control by the owner of the Web application. For example, examples of exploits have used proxies or the direct invocation of script functions to bypass the intended logic/business logic. Another problem with Ajax is that the source code incorporating business logic checks is now directly available to intruders just be downloading a Web page. Sharing business logic client-side reveals the source information of the complete system which makes the attacker's job easier. For example, an Ajax-enabled application with multiple levels of user account was found to have one JavaScript include file for the entire client-side logic. This meant that an anonymous user with trail account could see the logic behind both unprivileged and administrator-level service calls. This provided a map of the application that an attacker could use to attack business logic in the middle-tier.

Web sites are now fully functional software systems that provide business-to-customer e-commerce, business-to-business e-commerce and many other services to many users. The growing use of third-party software components and middleware represents one of the biggest changes in the e-commerce Web application systems.

The business application logic is a key weak link in security of many online sites. Typically, application subversion attacks as well as data driven attacks exploit weakness in this Web application software.

5 Security Properties Violations in Middle Tier

Traditionally the focus for the security community has been at the hardware, operating system and network levels. In recent years there has been interest in the middleware and application layers. In terms of our Web application, these are the layers where our business rules are implemented. Although their security relies upon lower layers and security mechanisms, vulnerabilities at these layers can represent the weakest link in a Web application. “A software system’s security & its integrity only as secure as its weakest component” [19].

Security problems originate from flaws in software design and implementation, and configuration management. These flaws are leveraged either maliciously or accidentally by the users of the software into providing a level of access and privilege that would not otherwise be granted by the program [5].

- **Vulnerabilities:** It can be the result of design flaw or implementation faults/errors. Design vulnerabilities are very hard to correct, while implementation vulnerabilities can be corrected through patches.a vulnerability is a defect or weakness in system security procedures, design, implementation, or internal controls that can be exercised and result in a security breach or a violation of security policy.

- **Bug:** Bugs are implementation-level problems leading to security risk [20]. Automated source code analysis tools tend to focus on bugs.

- **Flaw:** Defining flaws are design-level problems leading to security risk [20]. Flaws are the software design level problems that exist in the software; Human expertise is required to uncover flaws. Whereas, automated source code analysis tools tend to focus on bugs [20].

A flaw become the cause to of a vulnerability in the underlying software mitigating a flaw typically involves significantly more effort than simply modifying a few lines of code.

Designing software behavior is a process that involves identifying and coding up policy and business logic. The policy is enforced using security mechanisms. There is no sliver bullet for software security. Technology for scanning code is good at finding implementation-level mistakes, but there is no substitute for experience [8].

6 What is Logical Vulnerability in Terms of E-Commerce Web-Application?

All Web applications employ logic in order to deliver their functionality. Writing code in a programming language involves at its root nothing more than breaking down a complex process into very simple and discrete logical steps. Translating a piece of functionality that is meaningful to human beings into a sequence of small operations that can be executed by computer involves a great deal of skill. Doing it in an elegant and secure fashion is even harder still. In the very simplest of Web applications, a vast amount of logic is performed at every stage. This represents an intricate attack surface that is always present but often overlooked. Many code reviewers and penetration tests focus exclusively on the common “headline” vulnerabilities like SQL injection & Cross-Site Scripting, because these have an easily recognizable signature and well-researched exploitation vector. By contrast, flaws in an application’s logic are harder to characterize; each instance may appear to be a unique one-off occurrence, and they are not usually identified by any automated vulnerability scanners. As a result, they are not generally as well appreciated or understood & they are therefore of great interest to an attacker.

For example, an application might direct the user from point A to point B to point C, with the point B being a security validation check A manual review of the application might show that it is possible to go directly from point A to point C bypassing the security validation at pint B entirely.

7 Application Logic Attacks Operation

Unlike, common application technical attacks, such as SQL injection or Buffer Overflow. Each application logic attack is usually unique primarily because it has to exploit a function or feature that is specific to the application. This makes it more difficult for automated vulnerability testing tools to identify or detect such vulnerability class of attacks because these look for non-application specific flaws.

Such types of problem may sound unlikely or rate but a review of attacks on websites between 2006 and 2007 as identified in a 2008 FBI report (Published CNBC TV dated: 13 Feb. 2008) shows that server side application software flaws caused many of breaches. This was despite many of these sites using modern technologies such as the Java-based Web development model (JSP + Servlets) or the Microsoft .Net ASP Web development.

Such types of problem may sound unlikely or rate but a review of attacks on websites between 2006 and 2007 as identified in a 2008 FBI report (Published CNBC TV dated: 13 Feb. 2008) shows that server side application software flaws caused many of breaches. This was despite many of these sites using modern technologies such as the Java-based Web development model (JSP + Servlets) or the Microsoft .Net ASP Web development.

Figure 2 shows the poor design of an e-commerce Web software application. All the application logic is merged in a “Single Class” in the “Servlets”. Please note that Servlets are primarily intended to accept HTTP requests for delegation to other classes and to return formatted HTTP pages or replies to the client. Essentially this means that Servlets play a presentation role. The figure shows in the application server two different address-isolated containers performing different jobs. The first container “Web Container” only generates “Representation +Rendering Logic” whereas the EJB container “Application’s Business Logic” as per defined business rule/policy. Each container executes code in a different address space thereby minimising the opportunity for flaws in presentation code to affect business logic code.

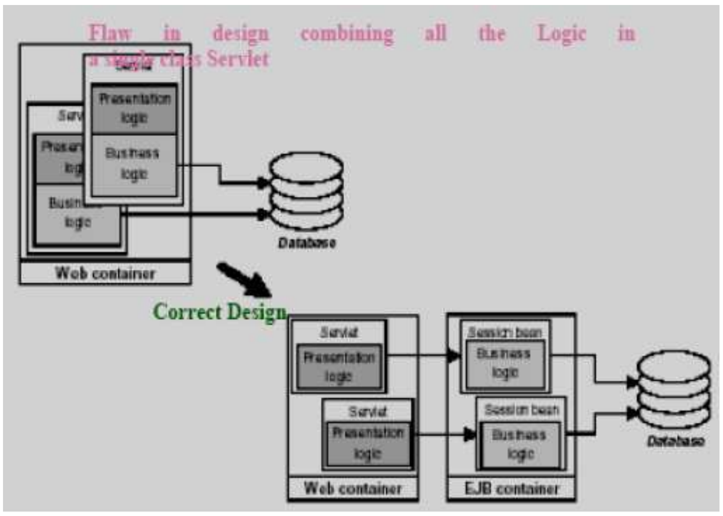


Figure 2: Flaw in design. Combining all the logic in a single servlet

Merging all together, leads to this possibility.

When application logic attacks are successful, it is often because developers do not build sufficient process validation and controls into application logic. This lack of functional flow control of logic allows attacker to perform certain steps incorrectly or out of order of the defined Logic.

From another angle to attempt an experiment attacking on application's business logic in the scenario of the (SOAP) by injecting code in the SOAP message.

In this case, as we know all that (SOAP) is a message-based communications technology that uses the XML format to encapsulate data. It can be used to share information and transmit messages between systems, even if these run on different operating systems and architectures. Its primary use is in Web services, and in the context of a browser-accessed Web application, you are most likely to experience SOAP in the communications that occur between "Application Components" [1].

SOAP is often used in large-scale enterprise applications where individual tasks are performed by different computers to improve performance (W3C.org). It is also often found where a Web application has been deployed as a front end to an existing application. In this situation, communications between different components may be implemented using SOAP to ensure modularity and interoperability. Because XML is an interpreted language, SOAP is potentially vulnerable to code injection in a similar way as the other examples already are [11]. XML elements are represented syntactically, using the "Metacharacters" < > and/. If user-supplied data containing these characters is inserted directly into a SOAP message, an attacker may be able to interfere with the structure of the message and so interfere with the application's logic or cause other undesirable effects [15].

A "Banking Application" in which a user initiates a funds transfer using an HTTP request like the following:

```
POST/transfer.asp HTTP/1.0
Host: ask-bank.com
Content-Length: 65
FromAccount=18281008&Amount
    =1430&ToAccount=08447656&Submit
    =Submit
```

In the course of processing this request, the following SOAP message is sent between two of the application's back-end components:

```
<soap: Envelope xmlns:soap="http://www.w3.org/
2008/2/soap-envelope">
<soap: Body>
<pre: Add xmlns:pre=http://target/lists
    soap: encodingStyle="http://www.w3.org/
    2008/2/soap-encoding">
<Account>
<FromAccount>18281008</FromAccount>
<Amount>1430</Amount>
<ClearedFunds>False</ClearedFunds>
<ToAccount>08447656</ToAccount>
</Account>
</pre: Add>
</soap: Body>
</soap: Envelope>
```

Look how the XML elements in the message correspond to the parameters in the HTTP request, and also the addition of the ClearedFunds (Component). At this point in the application's logic, it has determined that there are insufficient funds available to perform the requested transfer, and has set the value of this Component to False, with the result that the component which receives the SOAP message does not act upon it. In this situation, there are various ways in which you could seek to inject into the SOAP message, and so interfere with the application's logic. For example, submitting the following request will cause an additional ClearedFunds (Component) to be inserted into the message before the original element (while preserving the SQL's syntactic validity). If the application processes the first ClearedFunds (Component) that it encounters, then you may succeed in performing a transfer when no funds are available:

```
POST/transfer.asp HTTP/1.0
Host: ask-bank.com
Content-Length: 119
FromAccount=18281008&Amount=1430
</Amount>
<ClearedFunds>True</ClearedFunds>
<Amount>1430&ToAccount=08447656
        &Submit=Submit
```

If, on the other hand, the application processes the last ClearedFunds (Component) that it encounters, you could inject a similar attack into the ToAccount parameter.

A different type of attack would be to use XML comments to remove part of the original SOAP message altogether, and replace the removed elements with your own. For example, the following request injects a ClearedFunds (Component) via the Amount parameter, provides the opening tag for the ToAccount (Component), opens a comment, and closes the comment in the ToAccount parameter, thus preserving the syntactic validity of the XML:

```
POST/transfer.asp HTTP/1.0
Host: ask-bank.com
Content-Length: 125
FromAccount=18281008&Amount=1430
</Amount>
<ClearedFunds>True</ClearedFunds>
<ToAccount><!–&ToAccount=–>
08447656&Submit=Submit
```

A further type of attack would be to attempt to complete the entire SOAP message from within an injected parameter and comment out the remainder of the message. However, because the opening comment will not be matched by a closing comment, this attack produces strictly invalid XML, which will be rejected by many XML parsers:

```
POST/transfer.asp HTTP/1.0
Host: ask-bank.com Content-Length: 176
FromAccount=18281008&Amount=1430<Amount>
<ClearedFunds>True</ClearedFunds>
<ToAccount>08447656</ToAccount>
</Account>
</pre: Add>
</soap: Body>
</soap: Envelope>
    <!–&Submit=Submit.
```

SOAP injection can be difficult to detect, because supplying XML metacharacters in a noncrafted way will break the format of the SOAP message, and this will often simply result in an uninformative error message. Nevertheless, the following steps can be used to detect SOAP injection vulnerabilities with a degree of reliability. In most situations, you will need to know the structure of the XML that surrounds your data, in order to supply crafted input which modifies the message without invalidating it. In all of the preceding tests, look for any error messages that reveal any details about the SOAP message being processed. If you are lucky, a verbose message will disclose the entire message, enabling you to construct crafted values to exploit the vulnerability. SOAP injection can be prevented by employing boundary validation filters at any point where user-supplied data is inserted into a SOAP message. This should be performed both on data that has been immediately received from the user in the current request and on any data which has been persisted from earlier requests or generated from other processing that takes user data as input [15].

We have seen that above explained these cases & experience "No automation based Detection Tool" can Check Vulnerabilities performed by the intruders. It makes all of your efforts "Zero" which an organization invests on security for their organizational assert.

8 Application Logic Attacks Types

The different types of Logic Attack occur each time since it has to exploit a function or a feature that is specific to the application, for example when an attacker repeatedly uses an application's functionality such as the ability to create several thousand new accounts or posting repeated messages on discussion board. This type of attack abuses a useful application with little or no modification to the original function. The Logical Attacks focus on the exploitation of a Web application's logic flow. Application logic is the expected procedural flow used in order to perform a certain action. Password recovery, account registration, auction bidding, and e-Commerce purchases are all examples of application logic. A Web site may require a user to correctly perform a specific multi-step process to complete a particular action. An attacker may be able to circumvent or misuse these features to harm a Web site and its users.

9 Case Studies

The logic flaws differ hugely, each application logic attack is usually unique but they share many common themes and they demonstrate the kinds of mistake that human designer and developers will always be prone to making.

Therefore, insights gathered from case studies about logic flaws should help to uncover new flaws in entirely different situations.

9.1 Case Study: Insufficient Process Validation

This case study focus on the application logic flaw in the Web application an online retailer. This is case of Insufficient-Process-Validation.

The Application Functionality: The process of placing an order involved the following stages:

1) Browse the product catalog and add items to the shopping basket.
2) Return to the shopping basket and finalize the order.
3) Enter payment information.
4) Enter delivery information.

The Design Logic of Application: The developers assumed that users would always access the stages in the intended sequence, because this was the order in which the stages are delivered to the user by the navigational links and forms presented to their browser. Therefore, any user who completed the order process must have submitted satisfactory payment details along the way.

Attack Possibilities: The Designer/Developers' assumption was flawed for fairly obvious reasons. Users control every request that they make to the application and so can access any stage of the ordering process in any sequence. By proceeding directly from stage 2 to stage 4, an attacker could generate an order that was finalized for delivery but that had not actually been paid for.

Attacking Technique in this Scenario: The technique for finding and exploiting flaws of this kind is known as "Forced Browsing". Note that this should be distinguished from the Open Web Application Security Project (OWAP http://www.owasp.org/index.php/Forced_browsing). Unlike that attack, this does not involve resource discovery at the level of the Web browser. Our definition applies to the application-level. Forced browsing involves circumventing any controls imposed by in-browser navigation on the sequence in which application functions may be accessed:

- When a multistage process involves a defined sequence of requests, attempt to submit these requests out of the expected sequence. Try skipping certain stages altogether, accessing a single stage more than once, and accessing earlier stages after later ones.
- The sequence of stages may be accessed via a series of GET or POST requests for distinct URLs, or they may involve submitting different sets of parameters to the same URL. The stage being requested may be specified by submitting a function name or index within a request parameter. Be sure to understand fully the mechanisms that the application is employing to deliver access to distinct stages.
- From the context of the functionality that is implemented, try to understand what assumptions may have been made by developers and where the key attack surface lies. Try to identify ways of violating those assumptions to cause undesirable behavior within the application.
- When multistage functions are accessed out of sequence, it is common to encounter a variety of anomalous conditions within the application, such as variables with null or uninitialized values, a partially defined or inconsistent state, and other unpredictable behavior. In this situation, the application may return an error message and debug output, which can be used to better understand its internal workings and thereby fine-tune the current or a different attack. Sometimes, the application may get into a state entirely unanticipated by developers, which may lead to serious security flaws.

9.2 Case Study: Component-based Software Causes Design Flaw

EFU Insurance financial services company started an online insurance scheme "filing your own insurance". Unfortunately, there was a logic flaw in the Web application deployed by financial services company. This is a case of component-based Web application software flaw which cause of the application logic flaw.

The Application Functionality: The application enabled users to obtain quotations for insurance, and if desired, complete and submit an insurance application online. The process was spread across a dozen stages, as follows:

- At the first stage, the applicant submits some basic information, and specifies either a preferred monthly premium or the value the applicant wishes insurance for. The application offers a quotation, computing whichever value the applicant did not specify.
- Across several stages, the applicant supplies various other personal details, including health, occupation, and pastimes.
- Finally, the application is transmitted to an underwriter working for the insurance company. Using the same Web application, the underwriter reviews the details and decides whether to accept the application as is, or modify the initial quotation to reflect any additional risks.

Through each of the stages described, the application employed a shared component to process each parameter of user data submitted to it. This component parsed out all of the data in each POST request into name/value pairs, and updated its state information with each item of data received.

The Design Logic of Application: The component which processed user-supplied data assumed that each request would contain only the parameters that had been requested from the user in the relevant HTML form. Developers did not consider what would happen if a user submitted parameters that they had not been asked to supply.

Attack Possibilities: The Logic/assumption was flawed, because users can submit arbitrary parameter names and values with every request. As a result, the core functionality of the application was broken in various ways:

- An attacker could exploit the shared component to bypass all server-side input validation. At each stage of the quotation process, the application performed strict validation of the data expected at that stage, and rejected any data that failed this validation. But the shared component updated the application's state with every parameter supplied by the user. Hence, if an attacker submitted data out of sequence, by supplying a name/value pair which the application expected at an earlier stage, then that data would be accepted and processed, with no validation having been performed. As it happened, this possibility paved the way for a stored cross-site scripting attack targeting the underwriter, which allowed a malicious user to access the personal information belonging to other applicants.

- An attacker could buy insurance at an arbitrary price. At the first stage of the quotation process, the applicant specified either their preferred monthly premium or the value they wished to insure, and the application computed the other item accordingly. However, if a user supplied new values for either or both of these items at a later stage, then the application's state was updated with these values. By submitting these parameters out of sequence, an attacker could obtain a quotation for insurance at an arbitrary value and arbitrary monthly premium.

- There were no access controls regarding which parameters a given type of user could supply. When an underwriter reviewed a completed application, they updated various items of data, including the acceptance decision. This data was processed by the shared component in the same way as for data supplied by an ordinary user. If an attacker knew or guessed the parameter names used when the underwriter reviewed an application, then the attacker could simply submit these, thereby accepting their own application without any actual underwriting.

Attacking Technique in this Scenario: The flaws in this application were absolutely fundamental to its security, but none of them would have been identified by an attacker who simply intercepted browser requests and modified the parameter values being submitted.

- Whenever an application implements a key action across multiple stages, you should take parameters those are submitted at one stage of the process, and try submitting these to a different stage. If the relevant items of data are updated within the application's state, you should explore the ramifications of this behavior, to determine whether you can leverage it to carry out any malicious action, as in the preceding three examples.

- If the application implements functionality whereby different categories of user can update or perform other actions on a common collection of data, you should walk through the process using each type of user and observe the parameters submitted. Where different parameters are ordinarily submitted by the different users, take each parameter submitted by one user and try to submit this as the other user. If the parameter is accepted and processed as that user, explore the implications of this behavior as previously described.

9.3 Case Study: Barclay Bank Web Copycat Attack

This case study highlights the effect of a poorly designed application logic of a reused server-side component combined with the mis-configuration of the server-side component. The same flawed server-side component was incorporated into registration functionality and elsewhere within the application, including within the core functionality.

The Application Functionality: The application enabled existing customers who did not already use the online application to register to do so. New users were required to supply some basic personal information, to provide a degree of assurance of their identity. This information included name, address, and date of birth, but did not include anything secret such as an existing password or PIN number. When this information had been correctly entered, the application forwarded the registration request to back-end systems for processing. An information pack was mailed to the user's registered home address. This pack included instructions for activating their online access via a telephone call to the company's call center and also a one-time password to use when first logging in to the application.

The Design Logic of Application: The application's designers believed that this mechanism provided a very robust defense against unauthorized access to the application. The mechanism implemented three layers of protection:

- A modest amount of personal data was required up front, to deter a malicious attacker or mischievous user from attempting to initiate the registration process on other users' behalf.
- The process involved transmitting a key secret out-of-band to the customer's registered home address. Any attacker would need to have access to the victim's personal mail.
- The customer was required to telephone the call center and authenticate himself there in the usual way, based on personal information and selected digits from a PIN number.

This design was indeed robust. The logic flaw lay in the actual implementation of the mechanism. The developers implementing the registration mechanism needed a way to store the personal data submitted by the user and correlate this with a unique customer identity within the company's database. Keen to reuse existing Component code, they came across the following class, which appeared to serve their purposes:

```
class CCustomer {
String firstName;
String lastName; CDoB dob; CAddress homeAddress;
long custNumber;
... }
```

After the user's information was captured, this object was instantiated, populated with the supplied information, and stored in the user's session. The application then verified the user's details, and if they were valid, retrieved that user's unique customer number, which was used in all of the company's systems. This number was added to the object, together with some other useful information about the user. The object was then transmitted to the relevant back-end system for the registration request to be processed. The developers assumed that making use of this code component was harmless and would not lead to any security problem. However, the assumption was flawed, with serious consequences.

Attack Possibilities: The same component (code) that was incorporated into the registration functionality was also used elsewhere within the application, including within the core functionality, which gave authenticated users access to account details, statements, funds transfers, and other information. When a registered user successfully authenticated itself to the application, this same object was instantiated and saved in her session to store key information about her identity.

The majority of the functionality within the application referenced the information within this object in order to carry out its actions-for example, the account details presented to the user on his/her main page were generated on the basis of the unique customer number contained within this object. The way in the component code was already being employed within the application meant that the developers' assumption was flawed, and the manner in which they reused it did indeed open up a significant vulnerability. Although the vulnerability was serious, it was in fact relatively subtle to detect and exploit. Access to the main application functionality was protected by access controls at several layers, and a user needed to have a fully authenticated session to pass these controls. To exploit the logic flaw, therefore, an attacker needed to perform the following steps:

- Log in to the application using his own valid account credentials.
- Using the resulting authenticated session, access the registration functionality and submit a different customer's personal information. This causes the application to overwrite the original "CCustomer" object in the attacker's session with a new object relating to the targeted customer.
- Return to the main application functionality and access the other customer's Account. A vulnerability of this kind is not straightforward to detect when probing the application from a black-box perspective. However, it is also hard to identify when reviewing or writing the actual source code. Without a clear understanding of the application as a whole and the use made of different components in different areas, the flawed assumption made by developers may not be evident. Of course, clearly commented source code and design documentation would reduce the likelihood of such a defect being introduced or remaining undetected.

Attacking Technique in this Scenario: In a complex application involving either horizontal or vertical privilege segregation, try to locate any instances where an individual user can accumulate an amount of state within their session which relates in some way to their identity. Try to step through one area of functionality, and then switch altogether to an unrelated area, to determine whether any accumulated state information has an effect on the application's behavior.

10 The Proposed Framework Method for Secure Design of Business Application Logic

With this term pointing to the business logic layer in n-tier applications. The need for such a framework method is motivated by the fact that logical flaws do not show patterns or signatures and, thus, their discovery cannot be automated. Therefore, in order to counter this problem, we present a secure framework method based on existing techniques that treats security as a first-class concept and considers its interaction with business logic.

Effect of Attacks on System Design: One of the first steps in system design should be the analysis of the possible attacks on specific system and their consequences when successful. This analysis can be used to define the countermeasures that need and will also be useful later to evaluate the system security.

Layers Pattern: Security encompasses all the architectural levels of a system. The layers architectural pattern [17] is therefore the starting point of the design of secure systems. This pattern provides a structure where we can define patterns at all levels that together implement a secure system. Its main idea is the decomposition of a system into hierarchical layers of abstraction, where the higher levels use the services of the lower Levels. Here it provides a way to put things in perspective and to describe the mechanisms needed at each layer. Figure 3 shows the specific set of layers we consider. This figure shows some of the participants at each layer and their correspondence across layers.

10.1 Strategy: Checking for the Existence and Analyzing Weaknesses in Application Logic

In order to check for the existence of weaknesses in the application and analyze them, the strategy must focus on the configuration of the server-side middleware and the components implementing the business logic. For example, in the EJB environment, the configuration of Java connectors would need to be checked. Also components implementing each function in which the application can check a user's credentials should be analyzed for logic flow problems. Every request parameter submitted to the application should be varied to see what effect it has on the logic flow. This process would be repeated many times, modifying each parameter in turn in various unexpected ways designed to interfere with the application logic. Each stage of the mechanism tries to modifying the sequence and accessing different stages that the developer may not have anticipated. Determine whether any single piece of information is submitted at more than one stage, either because it is captured more than once from the user or because it is transmitted via the client in a hidden form field, cookie, or preset query string parameter. If so, try submitting different values at different stages (both valid and invalid), and observing the effect. Try to determine whether the submitted item is sometimes superfluous, or is validated at one stage and then trusted subsequently, or is validated at different stages against different checks. Try to exploit the application's behavior to gain unauthorized access or reduce the effectiveness of the controls imposed by the mechanism. Look for any data that is transmitted via the client that has not been captured from the user at any point. If hidden parameters are used to track the state of the process across successive stages, then it may be possible to interfere with the application's logic by modifying these parameters in crafted ways.

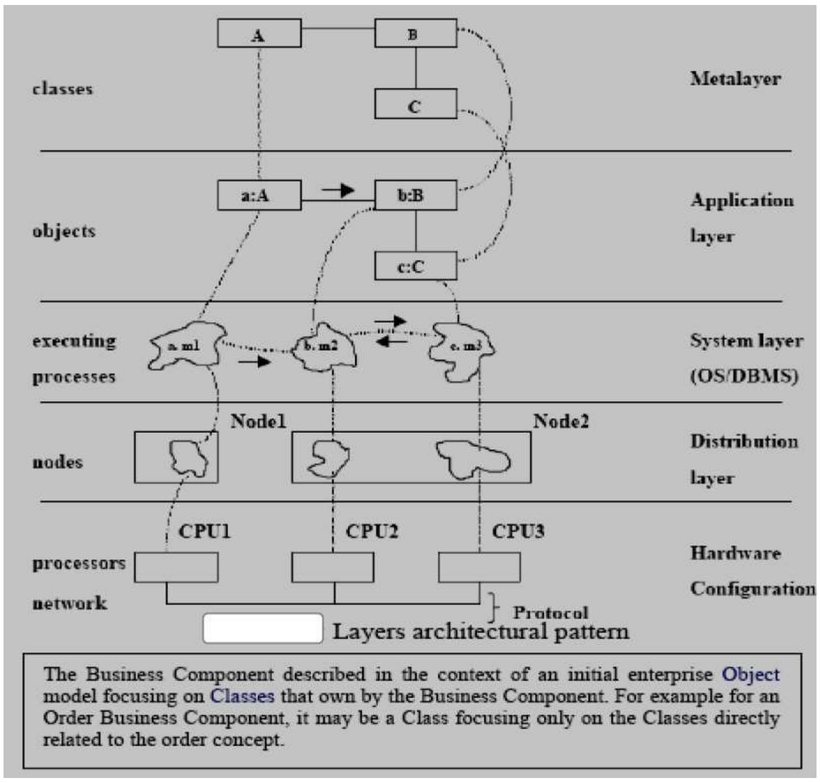
The strategy should be based on try each Parameter would consist the following changes:

- Submit an empty String as the Value.
- Remove the name/value pair altogether.
- Submit very long and very short Values.
- Submit the String instead of numbers.
- Submit the same named parameter multiple times, with the same and different values. For example apply this into above mentioned case of SOAP such as a ClearedFunds (Component) via the Amount parameter to check its integrity & functionality from all expected ways.

Carefully, closely review the application's response to the preceding requests. If it is found any unexpected divergences from the base case occur feed that back into your framing of further test cases. If one modification causes a change in behavior than try to combine this with other changes to push the application's business logic to its limits.

10.2 Strategy: Secure Business Application Logic

Verifying the design of secure business logic for an e-commerce distributed application in the middle tier is also very important. since many attacks are caused by design flaws in the e-commerce systems such logical flaws dose not often refer to component based flaws but also architectural, component modelling to set the logic of application while using business rules related to the particular business or activity. Therefore, it is very important to define clearly architectural design of topology in which system going to design for deploy by separating each tier clearly, second stage focus on the application logic design strategy & policy with that components have to function under given business defined rule/policy, third stage refer to design strategy for components which dynamic Web content is used to tailor an individual's interactions with a Web site & provide users with more interactive information. Dynamic content may be rendered in various form, such as static HTML files, Java Script or JSP file rendered using component supported environment such as Java Servlets in a J2EE .those invoke business -logic application hosted middle tier to access back-end business data. in the above given example of attack as it is stated that always follow right principles of Web application software engineering



The Business Component described in the context of an initial enterprise Object model focusing on Classes that own by the Business Component. For example for an Order Business Component, it may be a Class focusing only on the Classes directly related to the order concept.

Figure 3: Layers architectural pattern

in the right technology environment support for component interoperability & security, it is noticed in the example that threat to the application integrity & application dependability based on the design flaw & engineering of the component while defining logic for e-commerce system Web application software by merging all logic and invoke the direct access using Servlets through JDBC to the back-end having bypass the middleware process or without encapsulating the business logic in an enterprise bean which also define screening & as a shield to the organizational resources by restricting unauthorized people to access valuable data or temper the resources.

10.3 Strategy: Planning a Secure Component System

Identifying the components that need to be secured is a very important factor and first stage in the designing a secure environment for system. Next mechanisms that can be used to secure those components need to be identified. It is then necessary to understand which mechanisms are to be put together to secure the components thus giving rise to a secure development scenario.

In the n-tier distributed-computing environment, front-end presents presentation logic which invoke the business logic for the submitted request then the business logic layer hosted application interacts with the data tier & its logic for requested enquiry and computes the results that will be delivered to the presentation-logic layer Typically, security senility increases its flow from the first layer towards the last such partitioning into zones helps define the security requirement for the environment & the design of the topology to the host the components. It is also need to make sure that every aspect of the application's design is clearly mentioned in the sufficient detail to understand every assumption made by the designer and all such assumption must be explicitly on the record within design plan.

For example, sources code is clearly commented purpose & intended uses of each component and assumption made by each component about any thing that is outside of its direct functional control. It is also important to reference to all client code which makes use of the component and clear to it effect could have prevented the Logic flaw within the online registration functionality as defined in the example in that case "client" here refers not to the user end of the client-server relationship but to other code for which the component being considered is an immediate dependency. When implementing functions that update session data on the basis of input received from the user or actions performed by the user, reflect carefully on any impact that the updated data may have on other functionality within the application unexpected side effects can occur in entirely unrelated functionality defined by a different programmer or even a different development team.

10.4 Strategy: Architectural Risk Analysis for Component-based Business Logic

Design flaws account for 50% of the Security problems in the component-based software system [20]. Architectural risk analysis is, at best, a good general-purpose yardstick by which we can judge our security design's effectiveness [20]. Because roughly 50 percent of security problems are the result of design flaws, performing a risk analysis at the design level is an important part of a solid good secure Component-based software system engineering.

To encompass the design stage, any risk analysis process should be tailored. The object of this tailoring exercise is to determine specific vulnerabilities and risks that exist for the software [20]. Above mentioned design model by Cigital dose not clarify the each layer in the tier and its components, as its very important for a functional decomposition of the application into major components, processes, data stores, and data communication flows, mapped against the environments across which the software will be deployed, allows for a review of threats and potential vulnerabilities, as its defined in the new proposed n-tier e-commerce Web system architectural risk analysis & security management model.

It can contemplate using modelling languages, such as UML, to attempt to model risks; even the most rudimentary analysis approaches can yield meaningful results. Consider above model, which shows an n-tier deployment design model for Web-based application issues. As we applied risk analysis principles to this level of design, we achieved immediately some useful conclusions about the security design of the application.

During the risk analysis process must consider the following:

1) The threats those are likely to want to attack the system.

2) The risks present in each tier's environment.

3) The kinds of vulnerabilities that might exist in each component, as well as the data flow.

4) The business impact of such technical risks, were they to be realized.

5) The probability of such a risk being realized.

Solution Summary: Ensure that every aspect of the application's design must be clearly & sufficiently detailed to understand every assumption and designed function logic within the application by designer.

Mandate that all CBSD should be clearly commented to include the following information throughout.

1) The purpose and intended use of each component (IF Component code available information of code & its functional business logic within the component).

2) The assumptions & logic made by each component about anything that is outside of its direct control.

3) Reference to all client-code which makes use of the component clear documentation to this effect could have prevented the logic flaw within the online registration functionality (Note: Client have dose not refer to the user-end of the client-server relationship but to other code for which the component being considered is an Immediate dependency).

Solution Artifacts: As that there is no unique signature by which logic flaws in component-based rapid developed Web software application can be identified, because there is no silver bullet so far developed which could protect.

Good Practice: Good practice that can be applied to significantly reduce the risk of logical flaws appearing within component-based development and its logic. There are two important artifacts which we consider.

1) **During Security-focused Review of Application Design:** During the security-focused review of design, must reflect upon every assumption made within the design, and try to imagine circumstances in which each assumption and logic might be violated. Focus particularly on any assumed condition that could conceivably be within the control of application user based on business process, rule and policy.

2) **Security-focused Code Review:** Carefully, think laterally about two key areas;

   a. The ways in which unexpected user behavior and Input will be handled by the application.

   b. The potential side effects of any dependencies and interoperation between different code components and different application function.

11 Conclusion

Attacking an application's logic involves a mixture of systematic probing and lateral thinking. As we have identified, there are various key checks that you should always carry out to the application's behavior in response to unexpected input. These include removing parameters from requests, using forced browsing to access functions out of sequence, and submitting parameters to different locations within the application. Often, the way an application responds to these actions will point towards some defective assumption that can violate, to malicious effect.

Much of the security today is addressed as an audit activity that mostly relies on the penetration testing such testing activities often attempt to identify vulnerabilities that belong to certain categories of threats & use tools that are tailored around these threats. They may have security policies that auditors follow which require them to check a specific list of the things, but they often fall short of identifying vulnerabilities that a result of the way the application logic has been custom developed. The fact is that many attacks that are reported today fall under what we define as application logic attacks. Therefore, common sense is a appropriate tool while designing your Web application software and deploying component based business logic into that system, must focus on security beside the functionality because this functionality can be productive only when it work as per and within its functional control defined business policy into the e-commerce systems.

Acknowledgements

This research has been conducted with the bless of all mighty Allah Pak, Reference of wasila Ya Rasool-Allah Muhammad Peace be upon him and Ali Ibn-e-Abi Talib (There is no lad but Ali, there is no blade like Zolfikaar).Bullah what I know; who I am (Bullah ki Jana mein koon).I am also grateful of my Murshid Pak Hazraat Muostar Malang baba Chishtia Nizamia benefited from a great sufi saint Hazraat Khawaja Abdul Shakoor Malang baba Chishtia Nizamia, from the reference of Sultan-al-Hind Hazrat Khawaja Moinuddin Chisti (rahamtullah) famous with the title Khawaja Gharib nawaz (rahamtullah). Special Dedication to Dr Fiaz Hussain, Department of Computer Science & Technology, University of Bedfordshire, he is top class scientist, who gave me a lot of courage, while I was part of university of Luton.

References

[1] C. V. Berghe, J. Riordan, and F. Piessens, *A Vulnerability Taxonomy Methodology applied to Web Services*, IBM Zurich Research Laboratory, 2005.

[2] F. Cao, B. R. Bryant, R. R. Raje, M. Auguston, A. M. Olson, and C. C. Burt, "Component specification and wrapper/glue code generation with two-level grammar using domain specific knowledge," *Formal Methods and Software Engineering*, LNCS 2495, pp. 103-107, Springer-Verlag, Berlin, Heidelberg, 2002.

[3] E. Dustin, J. Rashka, and D. McDiarmid, *Quality Web System; Performance, Security & Usabilty*, Adition-Wesley, Boston, 2001.

[4] R. Ganesan, M. Gobi, and K. Vivekanandan, "A novel digital envelope approach for a secure e-commerce channel," *International Journal of Network Security*, vol. 11, no. 3, pp. 121-127, 2010.

[5] A. K. Ghosh, *E-Commerce Security: Weak Links Best Defence*, John Wiley & Sons, ISBN 0-47119223-6, New York, NY, 1998.

[6] A. K. Ghosh, *Security and Privacy in E-Commerce*, John Wiley & Sons, 2000.

[7] A. K Ghosh, *Security & Privacy for E-Business*, John Wiley & Sons, ISBN 0-471-384211-6, 2001.

[8] G. Hoglund, and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.

[9] M. Hung, and Y. Zou, *Extracting Business Process from the Three-Tier Architecture System*, Queen's University Kingston, ON, K7L 3N6, Canada 2005.

[10] M. Hung, and Y. Zou, "A Framework for Exacting Workflows from E-Commerce Systems," *Proceedings of Software Technology and Engineering Practice*, pp. 43–46, 2005

[11] M. McIntosh, and P. Austel, "XML signature element wrapping attacks and countermeasures," *Workshop on Secure Web Services*, pp. 20–27, 2005.

[12] F. Nabi, "Secure business application logic for e-commerce systems," *Computers & Security*, pp. 208–217, 2005.

[13] J. Offutt, "Qulaity attributes of Web software applications," *IEEE Software*, pp. 25–32, Mar. 2002.

[14] R. R. Raje, B. R. Bryant, M. Auguston, A. M. Olson, and C. C. Burt, "A unified approach for the integration of distributed heterogeneous software components," *Proceedings 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration*, pp. 23–31, 2001.

[15] M. A. Rahaman, A. Schaad, and M. Rits, "Towards secure soap message exchange in a soa," *Workshop on Secure Web Services*, pp. 35–42, 2006.

[16] P. Ritchie, "The security risks of Ajax/Web 2.0 application," *Network Security*, pp. 4–8, 2007.

[17] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

[18] G. Simson, and S. Gene, *Web Security and Commerce*, O'Reilly Publishing, 1997.

[19] J. Viega, and G. McGraw, *Building Secure Software*, John Wiley, ISBN 0-321-42523-5, 2006.

[20] D. Verdon, and G. McGraw, "Risk analysis in software design," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 79-84, 2004.

[21] C. Yang, "Secure Internet applications based on mobile agents," *International Journal of Network Security*, vol. 2, no. 3, pp. 228-237, May 2006.

Faisal Nabi, Junior Scientist in the field of Information & Computer Security was born in Karachi City. He had his initial schooling, college & University degree level education from Karachi. He went UK for higher education Oct 2001 Joined University of Luton, (Great Britain) M.Sc by Research in Electronic Commerce.. He is specialized in e-commerce/information security. Interest Area of Research: Cryptography, Steganography, Virtual Invisible Secure Disk design, Secure Architecture, Web Security. Started research R&D as an assistant Researcher with Professor: Carsten Maple 23 Nov 2003, Institute of Applied Research for Applicable Computing, University of Luton. In April 2004, he was appointed as Researcher at Deveraux & Deloitte Research Centre, UK. Faisal's Research work has been published in International Journals of IEEE/MCB level. Faisal has also received Research Award in the "First Cyber Security Conference 29 Sept 2004" at Sheraton Hotel, held in Karachi in conjunction with NR3C National Response of Cyber Crime.