LEGAL NOTICE

COPYRIGHTS © 2004, Imperva Ltd.

This document contains proprietary and confidential material of Imperva. Any unauthorized reproduction, use or disclosure of this material, or any part thereof, is strictly prohibited. This document is solely for the use by Imperva employees and authorized Imperva customers.

This is an unpublished work protected under the copyright laws. All rights reserved.

DOCUMENT DETAILS

<table>
<thead>
<tr>
<th>Document Type</th>
<th>Application Penetration Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name</td>
<td>Super Veda</td>
</tr>
<tr>
<td>Document Version</td>
<td>1.00</td>
</tr>
<tr>
<td>Created by</td>
<td>Application Defense Center</td>
</tr>
<tr>
<td>Creation Date</td>
<td>17/03/04</td>
</tr>
</tbody>
</table>

REVISION HISTORY

<table>
<thead>
<tr>
<th>VERSION</th>
<th>DATE</th>
<th>AUTHOR</th>
<th>CHANGE DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>17/03/2004</td>
<td>ADC</td>
<td>Document Created</td>
</tr>
</tbody>
</table>
REFERENCES

<table>
<thead>
<tr>
<th>REF</th>
<th>DOCUMENT</th>
<th>VERSION</th>
<th>DATE</th>
</tr>
</thead>
</table>

Note: To simplify cross-referencing a raised numeric in this document will refer to the above reference document.

ACKNOWLEDGMENTS

This specification was developed with input from:

<table>
<thead>
<tr>
<th>NAME</th>
<th>COMPANY</th>
<th>FUNCTION</th>
<th>LOCATION</th>
<th>PHONE</th>
</tr>
</thead>
</table>

CONTACTS

For more information about this document or its contents, please contact:

Imperva Professional Services:

ADC@IMPerva.COM
# Table of Contents

References ......................................................................................................................................... 3  
Acknowledgments ............................................................................................................................. 3  
Contacts ........................................................................................................................................... 3

1 Introduction....................................................................................................................................... 6

2 Scope & Limitations ........................................................................................................................ 7  
   2.1 Scope ...................................................................................................................................... 7  
   2.2 Limitations ............................................................................................................................. 7  
   2.3 Method ................................................................................................................................... 7  
   2.4 Prior Knowledge .................................................................................................................... 7

3 Application Description.................................................................................................................. 8  
   3.1 Functionality .......................................................................................................................... 8  
   3.2 Technology ............................................................................................................................. 9

4 Summary of Results...................................................................................................................... 10  
   4.1 Reading the Entire Database Contents ............................................................................... 10  
   4.2 Unauthorized Access to Accounts ....................................................................................... 10  
   4.3 Obtaining a Discount for Purchases .................................................................................... 10  
   4.4 Parameters Tampering ........................................................................................................ 11  
   4.5 Script Injection into Administrator’s Browser ....................................................................... 11  
   4.6 Script Injection into USER’S BROWSER ............................................................................. 11  
   4.7 Cross-site Scripting ............................................................................................................. 12  
   4.8 Permissions Misuse ............................................................................................................. 12  
   4.9 Forceful Browsing ............................................................................................................... 12  
   4.10 Information Disclosure .................................................................................................... 12

5 Detailed results................................................................................................................................. 14  
   5.1 Reading the Entire Database Contents ............................................................................... 14  
      5.1.1 showproducts.asp .............................................................................................................. 14  
      5.1.2 proddetails.asp ............................................................................................................... 17  
      5.1.3 addcomment.asp ............................................................................................................. 18  
      5.1.4 dosearch.asp .................................................................................................................. 19  
      5.1.5 getstates.asp ................................................................................................................... 20  
   5.2 Unauthorized Access to Accounts ....................................................................................... 20  
   5.3 Obtaining a Discount for Purchases .................................................................................... 22  
   5.4 Parameters Tampering ........................................................................................................ 24  
   5.5 Script Injection into Administrator’s Browser ....................................................................... 27
1 INTRODUCTION

The document hereby describes the proceedings and results of an application penetration test conducted against Imperva's demonstration application located on veda1.imperva.com. The penetration test took place during the month of November as part of an internal research. The penetration test was performed by application security experts from Imperva.
2 SCOPE & LIMITATIONS

2.1 SCOPE

The test was aimed at the Super Veda online market website, accessed through veda1.imperva.com.

The test was aimed at the application only, and did not include any attempts to exploit network or Operating System level vulnerabilities.

2.2 LIMITATIONS

There were no limitations during the test.

2.3 METHOD

The testing was done in a ‘Black-Box’ method, in which the testers had no information or prior knowledge regarding the application’s architecture or the technology used to implement it. This type of test gives an accurate simulation of an actual hacker attacking the system.

The tools used for the penetration test are a mixture of publicly available tools downloaded from the Internet along with special purpose homegrown tools.

2.4 PRIOR KNOWLEDGE

The test team received a single test account's user and password.
3 APPLICATION DESCRIPTION

3.1 FUNCTIONALITY

Super-Veda is a demo application written by Imperva in order to enable a thorough checking of various hacking techniques and common security vulnerabilities exploitation.

Super Veda is a web-based market application, where a user can perform online purchases of various products. Accessing the application a user can surf through the site’s many products – books, DVDs, CDs, electronic goods, and such. The user can see the item’s prices, their description, other user’s comments, and the current quantity on the shelves.

In order to actually perform a purchase the user has to sign-up in the site. This is done using a registration page where the user enters his details, such as address, full name, credit card number, phone number, and so forth.

After successful sign-in and choosing a password the user can access the site’s restricted pages using the new password. In this area the user can actually add items to a virtual cart. Some of the items have a certain discount on them. After finishing the selection of the offered items the user can head to the checkout page where he can alter his selections, while viewing the entire contents of his shopping cart.

Once he his satisfied with the quantity and the selection of products he can move on to performing the purchase order. After placing the order the user can access a ‘Track Order’ area, where he can monitor his order’s process.

The site also offers the user the option to use a search engine to directly locate a certain item in which he is interested.

The user can also contact the administrators of the site by filling in a contact form that is sent to them.
Alas, Super Veda is your common online market store, only with dozens of hidden trapdoors, eggs, vulnerabilities, and possible security exploit options - your friendly tool for hacking techniques demonstration and practice.

The main SuperVeda Screen is presented below:

3.2 TECHNOLOGY

The site is built as a collection of ASP pages, running on an IIS 5.0 Server. The data is stored on a Microsoft SQL Server 2000 database, residing on a separate machine.
4 SUMMARY OF RESULTS

The penetration test uncovered a number of serious vulnerabilities that jeopardize the security of individual accounts as well as the security of the application's internal network. Following is a short summary of major vulnerabilities:

4.1 READING THE ENTIRE DATABASE CONTENTS

Severity: Critical

An attacker can alter the address of some of the application web pages in such a way that enables him to query the internal database for all its information. As a result, the attacker can steal the entire collection of information within the database, which includes all the registered usernames, passwords, and credit card numbers. The attacker can generally be granted access to all the information in the database using a manipulation on the input of an SQL query.

4.2 UNAUTHORIZED ACCESS TO ACCOUNTS

Severity: Critical

An attacker can access accounts of all individual users without prior knowledge of their password, thus bypassing the application's authentication.

4.3 OBTAINING A DISCOUNT FOR PURCHASES

Severity: High

An attacker can manipulate the values of a cookie stored on his client in order to mislead the application into believing that his privileges are higher than they actually are, thus resulting in his obtaining a discount for his purchases.
4.4 Parameters Tampering

Severity: High

An attacker can manipulate the values of parameters stored on his client during the purchase session to alter the application's common workflow. This can lead to:

- An attacker misleading a client into reaching the order stage and then changing his order details, having the purchase reach the attacker's address instead of the rightful owner's address.
- Changing the purchase quantity, thus resulting in a purchase whose totals sum is negative.
- Changing the 'sale' parameter, thus causing a product to be sold at a price lower than its listed price.
- Changing the name of the active user while contacting the site's administrator, thus being able to impersonate another user, and acting on his behalf.

4.5 Script Injection into Administrator’s Browser

Severity: High

An attacker can take advantage of the “Contact Us” feature to inject malicious code into the browser of an administrator reading user messages. The injected code can be used to extract information from the administrator’s browser or as a tunnel for the attacker to the internal network.

4.6 Script Injection into User’s Browser

Severity: High

An attacker can take advantage of the products comments feature to inject malicious code into the browser of another visiting user who reads his comment. The injected code can be used to extract information from the innocent browser.
4.7 CROSS-SITE SCRIPTING

Severity: Medium

An attacker can take advantage of numerous input fields in the application in order to mislead an innocent customer into giving away information upon entering the site, or as a tunnel for the attacker for future purchases using the initial customer’s identity. Input fields include the comments area, the search page, and the new user signup form.

4.8 PERMISSIONS MISUSE

Severity: Medium

The user accessing the database is the same for all the users logging in to the system, and is the database administrator (user ‘sa’). This leads to an attacker being able to view tables outside of the scope of the application and to query the sysobjects/syscolumns tables.

4.9 FORCEFUL BROWSING

Severity: Low

Forceful browsing denotes the ability of an attacker to access modules not in the order they were meant to, possibly bypassing some application logic. In particular, unauthenticated forceful browsing may allow a user to access privileged information or actions without successfully completing the authentication process.

It turns out that in the Super Veda site the postcomment.asp page, which theoretically can only be accessed after login, can be accessed without prior authentication.

4.10 INFORMATION DISCLOSURE

Severity: Informative

- Detailed error messages within the Web server enable the user to gather reconnaissance and internal information about the structure of the SQL queries to the databases, the infrastructural technology behind the site.
(database, OS, hardware, etc). Moreover, using the SQL injection techniques mentioned in the above sections an attacker can query the database for its entire internal topology, and infrastructure architecture. This can later be used by an attacker as a lead towards better-attacking of the system.

- In some of the site's pages there are internal programmers comments left within the code. Those enable an attacker to gather crucial information as to the exact nature of some of the application's methodologies, and transactions flow.
5 DETAILED RESULTS

Following is detailed explanation of the above results.

5.1 READING THE ENTIRE DATABASE CONTENTS

Severity: Critical

An attacker can alter the address of some of the application Web pages in such a way that will enable him to query the internal database for all its information. As a result the attacker can steal the entire collection of information within the database, including all the registered usernames, passwords, and credit card numbers. The attacker can generally be granted access to all the information in the database using a manipulation on the input of an SQL query.

This attack can be performed on several of the site's pages:

5.1.1 showproducts.asp

In the showproducts.asp page the CatID parameter's value is taken and used in a SQL query for the list of products that match this category. For CatID=1 for example a list of all the Books in the database will be formatted and shown to the client. Since the value of the CatID is not checked during the SQL query formatting, an attacker can inject his query string into the CatID value causing the SQL query to return from the database any list that matches the required format.

The attacker uses the UNION SELECT directive in order to add another SELECT statement to the existing one. This second query can be done on any of the entire list of tables within the database.

The first step in the attack is a trial done by the attacker to see if the application checks for SQL injection. This is done by adding the string UNION SELECT * FROM users WHERE 1=1 to the CatID value, giving this URL:

http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT * FROM users WHERE 1=1
This will result in an error message indicating that "All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists", meaning that the '*' in the query that the attacker injected has a number of fields that don't match those of the first query. The attacker would now enumerate on the number of fields in the query one by one. First he would try to form a query of this format:

```
http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT 1 FROM users
WHERE 1=1
```

getting an error similar to the one he got beforehand. Thereafter he will try to add another field to his query:

```
http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT 1,1 FROM users
WHERE 1=1
```

and get an error again.

Doing this again and again will eventually end up in his getting a different error. This is done while the query he injected is:

```
http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT 1,1,1,1,1,1 FROM users WHERE 1=1
```

He learns from this that the first query has 6 fields in it. But still he gets an error. This time the error tells him that there's an problem "converting the varchar value 'Space, Site, Intervention By Erika Suderburg (Editor), Editor, Erika Suderburg's to a column of data type int.

This tells him that there's a field in his query that doesn't match the type of field in the original query – the first being a string (varchar), and the second being a number (int). He would therefore start enumerating on the types of the six fields by changing the number 1 to the string '1'. The query would turn to be of that kind:

```
http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT '1', 1,1,1,1,1 FROM users WHERE 1=1
```

This not solving the problem would tell him that the first field is indeed an integer, and would lead him to trying to turn the second field into a string (1 to '1'). Trying this would end in his getting a different result, telling him that another field has a conversion problem. He would then continue doing the same conversion operation until he would end up with the following query:

```
http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT '1', '1',1,1,1,1 FROM users WHERE 1=1
```

which will result in an actual results page.

Now he would use his knowledge on SQL Servers (he knows that this is the one the site is working with from the detailed error message he is
getting) and query a different table – the sysobjects. This table holds the entire collection of tables within the database, and has a known column names. The query would look like this:

http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT 1,name,1,1,'1','1 FROM sysobjects

Giving him the entire list of tables – as requested.

Last, he would form a query on the syscolumns table which holds, for each of the table IDs gotten from sysobjects, the name of all the columns within that table:

http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT 1,name,1,1,'1','1 FROM syscolumns WHERE id = (SELECT id FROM sysobjects WHERE name='Users')

Getting a list of all the column names, including: CCName – the name of the column holding the credit card numbers, Password – the name of the column holding the passwords and so forth.
With this information he can easily form his finishing query:

http://veda2.imperva.com/showproducts.asp?CatID=1 UNION SELECT 1,CCName,1,1,'1','1' FROM Users

Giving the list of all the credit card numbers for each and every user.

A demonstration of this can be seen in Appendix A.

5.1.2 proddetails.asp

Similar to the showproducts.asp the proddetails.asp gets an id with which he queries the products table and returns a formatted description of its details.
Opening this page - the attacker learns that there are two parameters sent with it, from which the first – ProdID is the querying the database for the product ID. Altering the query in a way similar to that done beforehand in the showproducts.asp page yields another SQL injection query, demonstrated in the following query:

http://veda1.imperva.com/proddetails.asp?ProdID=16 UNION SELECT 1,2,3,4,5,6,7 FROM Users

See a detailed demonstration of the attack in Appendix A.

5.1.3 addcomment.asp

In the addcomment.asp, as in the proddetails.asp a SELECT statement is built according to the given ProdID parameter. Adding a UNION SELECT statement to the invalidated parameter will yield in a record set queried from the database that applies to our injection. The URL with the injected SQL will look like this:

http://veda1.imperva.com/addcomment.asp?ProdID=31 UNION SELECT password FROM users
A demonstration of this can be seen in Appendix A.

5.1.4 dosearch.asp

While checking the search.htm file an attacker can notice that the search itself is done by a different page – which is not directly accessed – called: dosearch.asp. This page is given a parameter called string to which the content of the search is transferred. According to this string a query is formed looking for the content of the string in the entire list of products. The attacker can learn that he is able to change the query by deleting the entire parameter, and getting the entire list of products. This also indicates that the query is of the form:

```
SELECT <fields> FROM products WHERE <product name> LIKE %<the STRING parameter>%
```

Which – without any other parameter (empty string) - results in the entire list of products – as the attacker has seen. Noticing this, the attacker can easily move on to altering the SELECT query to include a UNION SELECT statement which will result in him getting the entire contents of a database as long as his query matches the types/number of parameters as the first one.

After enumerating on the number of columns in the query, and their types, the attacker can build a query similar to the following one.

```
http://veda1.imperva.com/dosearch.asp?string=%’ UNION SELECT 1,2,’3’,’4’,5,6,’7’ FROM users WHERE ‘=’
```

Notice that since the returned page is built to return results only for products fields, the query will not result in any response returned to the user. However, since the SQL is processed by the server, it can result in using the SQL for an UPDATE of the database or for INSERTing values into the database. Such values can be new products with different
pricings, or overwriting of the pricing of already existing products. Later on the attacker can purchase those very same products for a lower price.

A detailed explanation of the attack is given in Appendix A.

5.1.5 getstates.asp

The getstates.asp page is accessed from the registerx.asp page. When a user signing up for the first time chooses a country that has states (e.g. United States), those are loaded from the database, and added to the registration form. The getstates.asp page gets a countryid parameter which indicates the country to which states are to be loaded. With this countryid an SQL query is formed that loads the list of states. Changing the id or adding a "'" sign to it results in an SQL error which indicates the possibility of SQL injection to this page.

In a way similar to that of the above SQL injections, an attacker can use the UNION SELECT in order to change the original SQL, which has the following form:

\[
\text{SELECT <state fields> FROM Countries WHERE countryID=223}
\]

which – again – would give the passwords list from the users database.

Notice that in this query the SELECT probably has a GROUP BY statement too, which can be learned from the fact that the returned list is ordered.

See the step by step reconstruction of the attack in Appendix A.

5.2 Unauthorized Access to Accounts

Severity: Critical

An attacker can access accounts of all individual users without prior knowledge of their password, thus bypassing the application's authentication.

In the login.asp page there's a query for the user's username and password. Getting those two parameters the application builds an SQL query similar to this one:
SELECT userID FROM <users table> WHERE username = '<parameter received from the username field in the login.asp page>' AND password = '<parameter received from the password field in the login.asp page>'

By putting the string ' OR ''=' in the username field, and the string ' OR ''=' in the password field, the query would become:

SELECT userID FROM <users table> WHERE username = '' OR ''='' AND password = '' OR ''=''  

Resulting in a return of the entire list of records within the table. Since the application checks only whether the query returned a value or not, it will assume - since all the entries are returned - that the user successfully entered a correct login, and will use the first returned record as the correct login. This enables an attacker to access the site using an impersonated identity.

The attacker can change the query in a similar way, but with a slight change, where he asks for the entire record set without the name of the first person:

SELECT userID FROM <users table> WHERE username = '' OR ''='' AND password = '' OR ''='' AND name <> 'Mickey'

Given that 'Mickey' is the name of the first returned record in the record set.

This will enable the attacker to be given the name of the second person on the table's list. By adding more and more conditions like this the attacker can enumerate on the entire collection of users within the table – getting the entire list of names.

See a demonstration of this vulnerability in Appendix B.
5.3 OBTAINING A DISCOUNT FOR PURCHASES

Severity: High

An attacker can manipulate the values of a cookie stored on his client in order to mislead the application into believing that his privileges are higher than they are, thus resulting in him getting a discount for his purchases.

In what is known as ‘cookie poisoning’ attack - an attacker can monitor cookies sent from the server to the client, and easily notice a cookie being sent to the client while logging in to the , by which he’ll learn about a parameter called ‘Privilege’ that usually gets the value ‘None’.

Given the tempting name the attacker might try to interfere with the content of this cookie parameter by either changing it or erasing it. Either action taken by him will eventually result in him finding out that while continuing his navigations through the site he suffers from a problem while trying to end his purchase. Going to the checkout screen, and clicking the Buy button will yield the following error message:

“The server was unable to process your request”

Willing to find out exactly what was the cause of this error, the attacker might decide to take a look at the source of the received HTML page, where he will find the ‘commented-out’ programmer comment:

<font class="black_body_title">The server was unable to process your request</font><br /> <!-- Unknown Privileges Value. Privileges must be: None, FreeShip or Discount.-->

"The server was unable to process your request"
With this information the attacker can easily learn the other optional values for the Privilege parameter: Discount, or FreeShip.

Returning to the login page again he can now capture the cookie being received and change its value from None to Discount, for instance.

This leads to easily getting a 10% discount at the time of the purchase.
See a step by step reconstruction of the attack in Appendix A.

5.4 **PARAMETERS TAMPERING**

**Severity: High**

An attacker can manipulate the values of parameters stored on his client during the purchase session to alter the application's common workflow. This can lead to:

- An attacker misleading a client into reaching the order stage and then changing his order details, having the purchase reach the attacker's address instead of the rightful owner's address.

- Changing the purchase quantity, thus resulting in a purchase whose totals sum is negative.

- Changing the 'sale' parameter, thus causing a product to be sold at a lower price than its actual price.

- Changing the name of the active user while contacting the site's administrator, thus being able to impersonate another user, and acting in his behalf.

Following is a discussion of each of those tampering possibilities:
1. In the proddetails.asp page two parameters are passed to the product details query. Those are the ProdID that indicates the product number according to which the product is identified during all the queries, and the sale parameter which gets either the value ‘yes’ or ‘no’ which obviously tells the application whether this product is on sale or not. For each of the products, if the sale parameter indicates that the product is indeed on sale, then a query to a ‘Sales’ table is done, and according to the dates of the sale, a sale price is received. A user can easily change the parameter that is being sent from his client to yes for products that aren’t on sale, resulting in him getting a sale discount for those products while adding them to his cart. Notice that for products that do not have any sale option in the Sales table’s database, this attack wouldn’t work as no sale price would appear there. An attacker has to perform an enumeration on the products in the site, and try to gain a sale price for each of them.

2. In the checkout.asp page there’s the ability to change the quantity of each of the products purchased. This is done by calling the updatebasket.asp page with several parameters among which is the Quantity parameter with an integer value indicating the amount.
An attacker can directly call the updatebasket.asp page with different quantities. The major danger of this is the ability of the attacker to inject negative quantities to the updatebasket.asp page, which will result in a negative price for the products! Since the total amount of products is summed later on, the attacker can manipulate the total of all his purchase to be very low or zero, instead of negative, and thus avoid being noticed by any later inspection of the price order.
3. While in the contactus.asp page a user can direct a comment to the site’s administrator. The message is sent in his behalf and a hidden parameter with the user’s username is sent to the administrator. The user can alter the value of this parameter in his client-side html page, thus causing the comment to be sent using another user’s name.

To see a step by step reconstruction of those attacks, see Appendix D - Parameters Tampering.

5.5 SCRIPT INJECTION INTO ADMINISTRATOR’S BROWSER

Severity: High

An attacker can take advantage of the “Contact Us” feature to inject malicious code into the browser of an administrator reading user messages. The injected code can be used to extract information from the administrator’s browser or as a tunnel for the attacker to the internal network.
An attacker can embed HTML tags within the subject and body field of messages sent to the bank through the `contactus.asp` module. Tags may include invocation of scripts written in JavaScript or VBScript. When the message is reviewed by an administrator, embedded tags are parsed and executed by the administrator’s browser. The attacker may embed simple code that sends back a reply with the administrator’s credentials.

The attacker may even embed entire viruses. This is a particularly nasty possibility since the administrator is operating from within the bank’s network, behind its perimeter defense, which makes virus proliferation easier.

See a demonstration of this vulnerability in Appendix E - Script Injection into Administrator’s Browser.

### 5.6 Script Injection into User’s Browser

**Severity: High**

Script injection into the user’s browser – as opposed to cross site scripting – is a script written by an attacker that will be viewed by any viewer of a certain page in the site. This can be done by injecting script code into pages saved on the server, and later visited by innocent clients.

Such injection can be done in two places in the site, the first being the registration page. Since any comment sent by a user is later seen in the product’s comments, with his name, a user can change the name field in the registration to become a script tag that would make any comment-reading user run his script.
Needless to say that the name and the comments are saved on the Web server, thus requiring no cross-site attack on the innocent users.

This results in any user visiting any of the registered user’s comments on items, running the script.

See a demonstration of this vulnerability in Appendix F.
5.7 CROSS-SITE SCRIPTING

Severity: Medium

An attacker can take advantage of numerous input fields in the application in order to mislead an innocent customer entering the site into giving away information, or as a tunnel for the attacker for future purchases on behalf of the first. Input fields include the comments area, the search page, and the new user signup form.

The first cross site scripting attack is based on a malicious user embedding malicious code (in the form of Javascript or VBScript) in the search field of the search.asp page. This allows an attacker to send a mail to any user asking him to view a list of search results. If the innocent user would surf to this linked page, where the malicious code is injected by the attacker he would have a response script sent to him. This can result in the user’s session cookie sent to the attacker for instance, which will enable the attacker to act on the user’s behalf without his knowledge.

Appendix G - Cross-site Scripting demonstrates this problem.
5.8 **PERMISSIONS MISUSE**

**Severity: Medium**

The user accessing the database is the same for all the users logging in to the system, and is the database administrator (user ‘sa’). This leads to an attacker being able to view tables outside of the scope of the application and to query the sysobjects/syscolumns tables.

5.9 **FORCEFUL BROWSING**

**Severity: Low**

Forceful browsing denotes the ability of an attacker to access modules not in the order they were meant to, possibly bypassing some application logic. In particular, unauthenticated forceful browsing may allow a user to access privileged information or actions without successfully completing the authentication process.

It turns out that in the Super Veda site, the `postcomment.asp` page, which originally can be accessed only after logging-in, can be accessed without prior authentication.

5.10 **INFORMATION DISCLOSURE**

**Severity: Informative**

- Detailed error messages within the Web server enable the user to gather reconnaissance and internal information about the structure of the SQL queries to the databases, and the infrastructural technology behind the site (database, OS, hardware, etc). Moreover, using SQL injection techniques mentioned in the above sections an attacker can query the database for their entire internal topology, and infrastructure architecture. This can later be used by an attacker as a lead towards better-attacking of the system.

- In some of the site's pages there are internal programmers comments left within the code. Those enable an attacker to gather crucial information as to the exact nature of some of the application's methodologies, and transactions flow.
Using common HTTP knowledge a user can use the TRACE directive in order to query the server for its type and the OS type. Such a query results in the Server parameter returned, indicating the fact that the site is operating on Microsoft-IIS/5.0 server.
6 RECOMMENDATIONS

Implementing the following recommendations will eliminate the above mentioned problems:

6.1 AVOIDING SQL INJECTION

The following recommendations refer to all pages that are susceptible to SQL Injection. This includes:

- showproducts.asp
- proddetails.asp
- addcomments.asp
- dosearch.asp
- getstates.asp
- login.asp

The best way to avoid SQL Injection is by using prepared statements, rather than using string queries. With a prepared statement, the syntax of the statement is first set, and only later the parameters are transferred, ensuring that there is no possible mix between the SQL syntax and the parameter (unlike string queries, that mix, during the creation of the string, the parameters and the syntax).

In addition, it is recommended that sanity checks will be performed on each received parameter before processing it. All ID parameters (such as CatID, ProdID, etc.), can be easily validated by verifying they contain only digits. Other parameters may have only specific values, or should contain only alphanumeric characters.
6.2 OBTAINING A DISCOUNT FOR PURCHASES

In order to avoid the possibility of an attacker taking advantage of the information stored in his machine it is advised to perform the check of the users eligibility for a discount/free purchase at the time of the purchase itself, and not saving this data at the client side in a cookie.

6.3 PARAMETERS TAMPERING

- For the ‘sale’ parameter it is advised to take those steps into account:
  1. It is advisable to completely delete sales that are over from the table, so no attack on those can be performed.
  2. It is recommended to add the on-sale flag to the product itself, and to check for it being on sale on the server side, and not according to a parameter received from the client.

- For the ‘quantity’ parameter being sent from the ‘checkout.asp’ page:
  It is advisable to have the quantity checked on the server, upon submitting each of the shopping cart’s items quantities for negative values, so that it won’t be possible to inject those.

- For the username parameter being sent from the ‘contactus.asp’ page:
  As a rule it is advisable not to refer to values received from the client as is, even if they are ‘hidden’ input fields. An attacker can easily locate such, and manipulate their contents. With regards to the Contact Us page, it is advisable to either have the user type his name upon writing his comments to the site’s administrator, if authentication is essential for the flow of the sent comment, or have the username be kept in a session object on the server – which is more recommended.

6.4 SCRIPTS HANDLING

The following recommendations refer both to script injection and cross-site scripting attacks.

Avoiding script injection requires changes to each module that displays alphanumeric data that is controlled by users (e.g. text entered by users in input fields). Each time such data is displayed it should be first encoded using a schema called HTML encoding. This encoding schema transforms
special HTML character to its mnemonic representation. For example, the character ‘<’ is encoded as ‘&lt;’. Such an encoding ensures that benign text is displayed as is, while text with injected code in it is displayed as pure text without execution of the code (actually the code is displayed as part of the text). Built-in functions for HTML encoding are available in both JavaScript and VBScript.

Changes should be made to modules that display data for users as well as modules that display the data for the administrators.

6.5 **PERMISSIONS MISUSE**

It is recommended to create a specific Web server – db connection user with limited privileges, under which most of the read-only queries will be operated. Since most of the queries in the site don’t require update of the db, there’s no need for an administrator user managing the connection.

6.6 **FORCEFUL BROWSING**

Make sure that the *postcomment.asp* page, and all other future added pages, are beginning with an authentication-checking page, that redirects any unauthenticated access to the page to the *login.asp* page.

6.7 **INFORMATION DISCLOSURE**

It is recommended to have minimal informative data sent to the user. For example, a default error page is returned to the user instead of specific error pages reported to the user.

It is advised to suppress all abilities of the Web server to disclosure information regarding its type to HTTP requests, and preferably to configure it to answer only GET and POST requests by clients.
APPENDIX A - READING THE ENTIRE DATABASE CONTENTS

**showproducts.asp**

The attacker uses the 'Open in new window' option to open the showproducts.asp page solely. Thus, having the ability to inject URL strings to it.

**Learning that the Page is Vulnerable to SQL Injection**

The attacker tries to add the ` ` sign to the URL learning that the CatID is part of an SQL statement.

![Image of showproducts.asp](../images/showproducts.asp.png)

**Adding the UNION SELECT Query**

The attacker changes the select query to include the UNION SELECT statement.

![Image of showproducts.asp with error](../images/showproducts.asp_error.png)
Getting an error message stating that the number of columns doesn't match that of the beginning of the query makes the attacker start enumerating on the number of fields.

After a few cycles he reaches the correct number, getting the following error:
Identifying the Field's Types

After understanding that there are 6 fields in the original query, the attacker enumerates on their types by changing the type from int to char until the error messages change.

First he changes the first 1 to ‘1’ – not getting any result.

Then he changes the second field to a varchar field – getting this new error message:

He understands that there is yet another field who is to be turned from int to varchar, so he keeps his enumeration, soon getting another new error message.
Which – again – tells him that he is correct in his previous placements of varchar fields, but there are still more changes to perform. He continues his path of changing ints to varschars, until he gets a full compliant query result page.
Locating the Users Table

This one yields no information as it gives 1-s as its output. He changes the varchar fields into the known 'name' column which exists in the sysobjects table, getting a list of all the tables in the database.
Among this long list he will eventually find the Users table.
Locating the Users Table's Columns

After having the table's name, the attacker heads to getting its column names. This is done by querying the syscolumns table.

Getting the Information

After learning that the Users table has fields such as CCNumber and Password, he heads towards querying those – practically being able to get whatever information he wants from the DB…

Password
Credit Card Numbers
**add2cart.asp**

SQL Injection without seeing the result.

```
https://veda.webcohort.com/add2cart.asp?prodID=3&sql=--%20UNION%20SELECT%20*%20FROM%20users
```

Thank you for adding this item to the cart.

Click here to close the window.
proddetails.asp

The attacker injects the following SQL statement into the URL, getting this result.
addcomment.asp

The regular addcomments.asp page queries the database for the name of the product to which to add comment. This query is done using the ProdID parameter taken from the URL query string.

Adding our UNION SELECT query to the URL will result in an altered query giving anything we would query for from the database. For instance, in this example we queried for the password of the first user in the record set.

Obviously, using different queries this can be used to get any entry value from the database.
Activating the `dosearch.asp` page can be done either by directly accessing it using the URL, or by injecting data to the search field in the `search.htm` file which is immediately transferred to the `dosearch.asp` page.

### Injecting through the `search.htm` File

Breaking out of the original SQL query is done using a `%` sign which enables us to add the UNION SELECT query of our own. Putting the entire query:

```
* UNION SELECT 1,1,password,'1','1','1' from users where '%1'='1'
```
Results in a return of the entire passwords collection from the database:
Access to the database can also be done directly by altering the URL of the *dosearch.asp* page:
getstates.asp

Injecting through the search.htm File

After getting to the registration page, an attacker might take a look at the HTML code of the latest, noticing it's using a javascript to call getstates.asp in case he chooses a country that has states.
Querying the States from `getstates.asp`

Querying directly from `getstates.asp` leads to us getting the states.

```
http://veda1.webroot.com/getstates.asp?countryid=223 - Microsoft Internet Explorer
```


Modifying the SQL Query

Changing the query to enable breaking out of the original one and adding a UNION SELECT with a new statement results in getting a modified list that includes the data from the database.
APPENDIX B - UNAUTHORIZED ACCESS TO ACCOUNTS

The attacker reaches the login.asp page while trying to access any of the restricted pages. Since the application checks, on any of the restricted pages, whether the client is authenticated, and finds out that he isn't, it sends him to the login page that looks like this:

```
Login

Username: __________________________
Password: __________________________

Login
```

Click here to send the password to you.

Access to the Site

The attacker then inserts the following SQL bits into the username/password fields:

```
Access to the Site

Username: OR 'M'
Password: ************
```

And get in using the first name in the returned record set ('Mickey'):
Application Penetration Test for Super Veda

- Sample Report -

Super Veda

One cart. One bill. One shipping.

New User? Sign up now!  About Us  Logout  Hello, Mickey

Home

SuperVeda.com has teamed up with Borders, Inc., Circuit City, and Office Depot to make thousands of books, DVDs, CDs, office products, electronics, and other items available for pickup at hundreds of stores nationwide. Items eligible for in-store pickup feature a box on their product detail pages in which you can

Products

Imperva™

Page
54 of
73
APPENDIX C - OBTAINING A DISCOUNT FOR PURCHASES

Access to the Site

The attacker first traces the connection between the client and the server, finding out that during login a cookie is being sent to the client with the Privilege=None directive.
Changing the Cookie Parameter

Using common HTTP protocol impersonating tool the attacker then erases the cookie parameter, so that it is not sent to the user. Thereafter, the attacker continues to navigate the site, until he reaches the purchase checkout screen, where – while trying to perform a purchase he gets this notification screen:
Bearing in mind that this doesn't occur during normal use, the attacker uses the 'View Source' option to check the nature of the activity in this page. The source reveals the following information:

Where the last line is of interest to him, stating in the 'commented-out' comment that:

<!-- Unknown Privileges Value. Privileges must be: None, FreeShip or Discount.-->

<!-- Unknown Privileges Value. Privileges must be: None, FreeShip or Discount.-->
Poisoning the Cookie

With this information the attacker heads towards changing the cookie (during another login to the application) to 'Discount'.

![Image of Poised the Cookie with cookie value set to 'Discount']
Getting a Discount

After having the cookie changed, the attacker can head to the checkout screen again, and notice that he is now getting a discount on his purchase.
APPENDIX D - PARAMETERS TAMPERING

Getting a Sale Price for a Product not on Sale

First the attacker goes to the product details page, seeing the product's initial price.

---

Product Details

Runner's World

Product Description: 12 issues / 12 months
List Price: $19.99
Quantity In Stock: 34 Units
Add to cart

User Comments

Be Very Very Afraid
I'm hunting vulnerabilities!
By Elmer Fudd

Add a comment for this product
Afterwards, he calls the same page again, this time changing the URL's sale parameter to yes, instead of no, getting a notification of this product not being on sale, but being sold for a different price. Doing the same with products that have a sale price would result in the attacker getting the product's lower price.
Changing the ProdID to a product that has a sale price would result in the attacker getting the normal ‘The products was added to the cart’ message, but the price is lower than the normal one.

**Sending a Message on Behalf of Another User**

The attacker – being logged-in – fills in a complaint in the Contact Us page of the site.

Choosing to view the source of the contactus.asp page reveals this:

```html
A hidden input parameter named 'username', holding the value 'bugsb' which is the username of the logged-in user is embedded within the page. He easily changes this value to the name of another user, and presses the submit button – making the message seem as if it came from another user.
```
Getting a Lower Price Purchase

The attacker purchases a set of products in the site and heads to the checkout page where he sees the list of products he purchased, and the total sum for them.

Noticing the while hovering over the +/- sign of the quantity the linked URL includes a parameter called Quantity, that indicates the ability to add another item of the same kind to the purchase. This parameter is determined by the server according to the original number in the page – having the + sign indicate a single increment of the number, and the – sign indicate a single decrease of the number.

The attacker generates a URL of his own, with the quantity being -1 for example, which would look like this one:

This would result in the price being lowered by \(-1\) times the original price, which is obviously lower than the original amount.

He can alter some more of those numbers to create a sum of zero or lower for the Total, which will be either ‘paid to him’ or not charged to his bill!
Deleting a User

An attacker might follow the parameters sent by the pages to one another while surfing through the site, until he notices that the register.asp page alone doesn’t generate any HTML code by the server.
He would later try to locate referring pages to the above, until he notices that for this page to operate normally a parameter is added to the URL, stating that this page is to be opened in a mode that refers to creating a new user.

The attacker could then easily guess that a parameter by the name deluser might exist too, and use this URL to call it.

http://veda1.imperva.com/register.asp?mode=deluser
APPENDIX E - SCRIPT INJECTION INTO ADMINISTRATOR’S BROWSER

An attacker can inject script code to the administrator reading data from the site, by heading to the contactus.asp page in the site, where he will get a text area that should hold the information that is sent to the administrator later on.
Willing to attack the administrator of the site, he can include malicious code in the text area that would later on be operated while the administrator checks the data. This is demonstrated below:

```
<script>alert('script injection on the admin')</script>
```

APPENDIX F - SCRIPT INJECTION INTO USER’S BROWSER

Script Injection using the Registration Page

Script injection into the user’s browser – as opposed to cross site scripting – is a script written by an attacker that will be viewed by any viewer of a certain page in the site. This can be done by injecting script code into pages saved on the server, and later visited by innocent clients.

Such injection can be done in two places in the site, the first being the registration page. Since any comment sent by a user is later seen in the product’s comments, with his name, a user can change the name field in the registration to become a script tag that would make any comment-reading user run his script.

Needless to say that the name and the comments are saved on the Web server, thus requiring no cross-site attack on the innocent users.

The injection is easily done while putting a script code in the First Name field, during registration, as demonstrated below:
Now the attacker heads towards the comment page, where he adds comment by his name.

Later on, when a user logs into a page including the name of the attacker, for instance the product’s comments page, he will receive, from the server, the script code embedded in the commenting user’s first name.
Script Injection using the Comment Page

Another page where an attacker can know that any visiting innocent user will surely view is the comments page. Injecting a script into that page, again, would result in attacking any visiting client, since the comments code is saved on the server-side, and sent to a visiting user.

The attacker can add comment that include a malicious code to any of the products in the site.

A visiting user choosing to view the product's comment would be receiving this script code, and run it on his machine, enabling the attacker to have files sent to the 'attacker's machine, install Trojans on the clients, and other malicious attacks.
Cross-Site Scripting in the Search Page

An attacker can take advantage of numerous input fields in the application in order to mislead an innocent customer entering the site into giving away information, or as a tunnel for the attacker for future purchases on his behalf. As a demonstration of this ability we will show an injected javascript alert tag resulting in a dialog box appearing with the word ‘hacked’.

In the search page search.asp the results of the search are returned as is, in the returned page. An attacker injecting the search string:

```html
<script>alert("hacked")</script>
```

Would result in this dialog box appearing on his screen.
Cross-Site Scripting in the Comments Page

In the subject of a comment in the *addcomment.asp* page, an attacker can inject scripting.