



## Traditional SQL Injection Protection: The Wrong Solution for the Right Problem

Amichai Shulman, CTO, Imperva Inc.

# Introduction

- “Traditional SQL Injection Protection: The Wrong Solution for the Right Problem”, looks at several common protection mechanisms against SQL Injection and where they fail
- After a short overview of SQL Injection, 3 different attack classes will demonstrate where these mechanisms fail
- All the attack techniques presented today are the result of research done at Imperva™'s Application Defense Center over the past years
- The presentation will include a live demonstration of the attack techniques against a demo e-commerce application

# The Problem – SQL Injection

- SQL Injection is arguably today's biggest threat to web applications
- Statistics gathered throughout Penetration Testing, show that nearly two thirds of all web applications are vulnerable to SQL Injection attacks
- Such applications are susceptible to various threats, including data theft, confidentiality and integrity breach, access policy violation, remote command execution and denial of service. These threats are usually classified as High or Critical
- The majority of actual hacks reported by the press, of hackers stealing credit card numbers/private Information/etc, are in fact successful SQL Injection attacks

# The Problem – SQL Injection

- An application is vulnerable to SQL Injection as a result of faulty *programming* of the application itself
- It is not the result of a faulty architecture or bad deployment

## Claim

The Wrong Solution - Applying security measures taken from the infrastructure security domain

The Right Solution – Applying security to the application and/or relevant code

# A Quick Overview of SQL Injection

- So what exactly is SQL Injection?
  - SQL Injection is an attack which allows the attacker to alter the syntax of SQL statements generated by the application
  - It is a result of a simplistic database access mechanism, which is based on single string queries
  - The programmer chains together syntax and parameters into a single string which is then sent to the database. Usually, no input validation is performed beforehand
  - In an SQL Injection attack, the attacker embeds SQL syntax into a parameter's value, causing the query's syntax to change

# A Quick Overview of SQL Injection

## Example I: Authentication Circumvention

The Code:

```
...  
Sql Qry = "SELECT * FROM Users WHERE Username = '" &  
Request.QueryString("User") & "' AND Password = '" &  
Request.QueryString("Pass") & "' "  
Logi nRS.Open Sql Qry, MyConn  
If Logi nRS.EOF Then Response.Wri te("Inval id Logi n")  
...
```

When a normal user logs in, the following query is created:

```
SELECT * FROM Users WHERE Username = ' John'  
AND Password = ' Smi th'
```

The attacker, however, inserts `X' OR '1'='1` as the password, altering the query into the following (non empty) one:

```
SELECT * FROM Users WHERE Username = ' John'  
AND Password = ' X' OR '1'='1'
```

# A Quick Overview of SQL Injection

The image consists of two side-by-side screenshots of the Super VEDA website in Microsoft Internet Explorer. The left screenshot shows the login page. The 'Username' field contains 'nosuchuser' and the 'Password' field contains the SQL injection payload 'xxx' or '1'='1'. Both fields are circled in red. The right screenshot shows the main page of the website. The text 'Hello, Mickey' is displayed in the top right corner, circled in red, indicating a successful session hijack. The website header includes 'Super VEDA' and 'One cart. One bill. One shipping.' The main content area features a 'Hot Sales' section with several product listings, including 'The Matrix Revisited (2001)', 'But Beautiful, Standards: Volume 1 - Boz Scaggs', 'Space, Site, Intervention By Erika Suderburg (Editor), Editor, Erika Suderburg', and 'Metropolis (Restored Authorized Edition)'. The footer includes the IMPERVA logo and copyright information for 2003.

# A Quick Overview of SQL Injection

## Example II: Data Retrieval

The Code:

```
Sql Qry = "SELECT * FROM Products WHERE ProdDesc LIKE "  
& "'%' Request.QueryString("SearchTerm") & "%' "  
ProdsRS.Open Sql Qry, MyConn
```

The query that is normally created when using the form is:

```
SELECT * FROM Products WHERE ProdDesc LIKE '%matrix%
```

Showing all  
matching results:



The screenshot shows a web browser window titled "Super VEDA - Microsoft Internet Explorer". The page features the SuperVeda logo and navigation links. A search bar contains the text "matrix". Below the search bar, two search results are displayed:

Product Name	Price	Availability	Action
The Matrix Revisited (2001)	\$14.95	In stock	<a href="#">Add to cart</a>
The Animatrix (2003)	\$19.98	Not in stock	<a href="#">Add to cart</a>

# A Quick Overview of SQL Injection

## Example II: Data Retrieval (Continued)

The attacker now uses the following string as the search term:

```
XXX' UNION SELECT 1, 1, Username + ' : ' +  
Password,  
1, CCNumber, 1 FROM Users --
```

Causing the original query to be altered into the following one:

```
SELECT * FROM Products WHERE ProdName LIKE  
'%XXX' UNION SELECT 1, 1, Username + ' : ' +  
Password, 1, CCNumber, 1 FROM Users --%'
```

As a result, the query now returns all products whose name terminates with 'XXX' (probably none), as well as the list of the users, their passwords, and their credit card numbers

# A Quick Overview of SQL Injection

The screenshot shows a web browser window titled "Super VEDA - Microsoft Internet Explorer". The page displays the SuperVeda logo and navigation links. A search bar is present with the text "SELECT 1,1,Username + ':' + Password" entered. Below the search bar, a list of items is displayed, each with an "Add to cart" button. A red circle highlights this list of items.

Item Name	Price	Stock Status	Action
bugsb : carrots	\$1.23400045000121E+15	In stock	Add to cart
donaldd : scrooge	\$1.23400002000051E+15	In stock	Add to cart
elmerf : rabbits	\$1.23400456789111E+15	In stock	Add to cart
mickeym : minnie	\$1.23400060000101E+15	In stock	Add to cart
tazd : tazdevil	\$1.2340000005011E+15	In stock	Add to cart

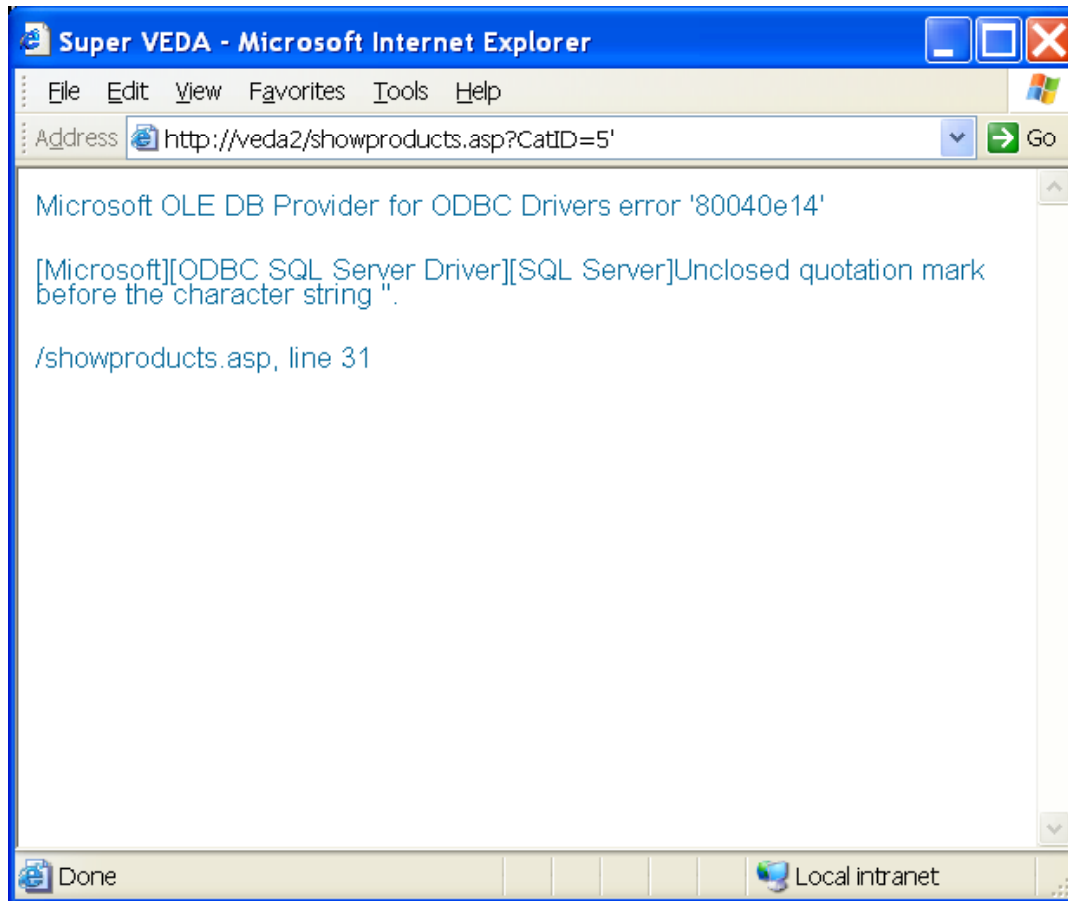
# The Wrong Solution – Infrastructure Countermeasures

- SQL Injection is a ***security*** problem caused by naive ***programming***
- Programming is the responsibility of the R&D division
- Security is the responsibility of the IT division
- Consequence – network or system level countermeasures are taken against application level problems:
  - Server Configuration (e.g. Hiding Error Messages)
  - Network Level Monitoring (e.g. Expanding IDS/IPS systems)
  - Infrastructure ACLs (e.g. Revoking Database Privileges)
- While these countermeasures should always be implemented, as they provide additional security, they are not sufficient

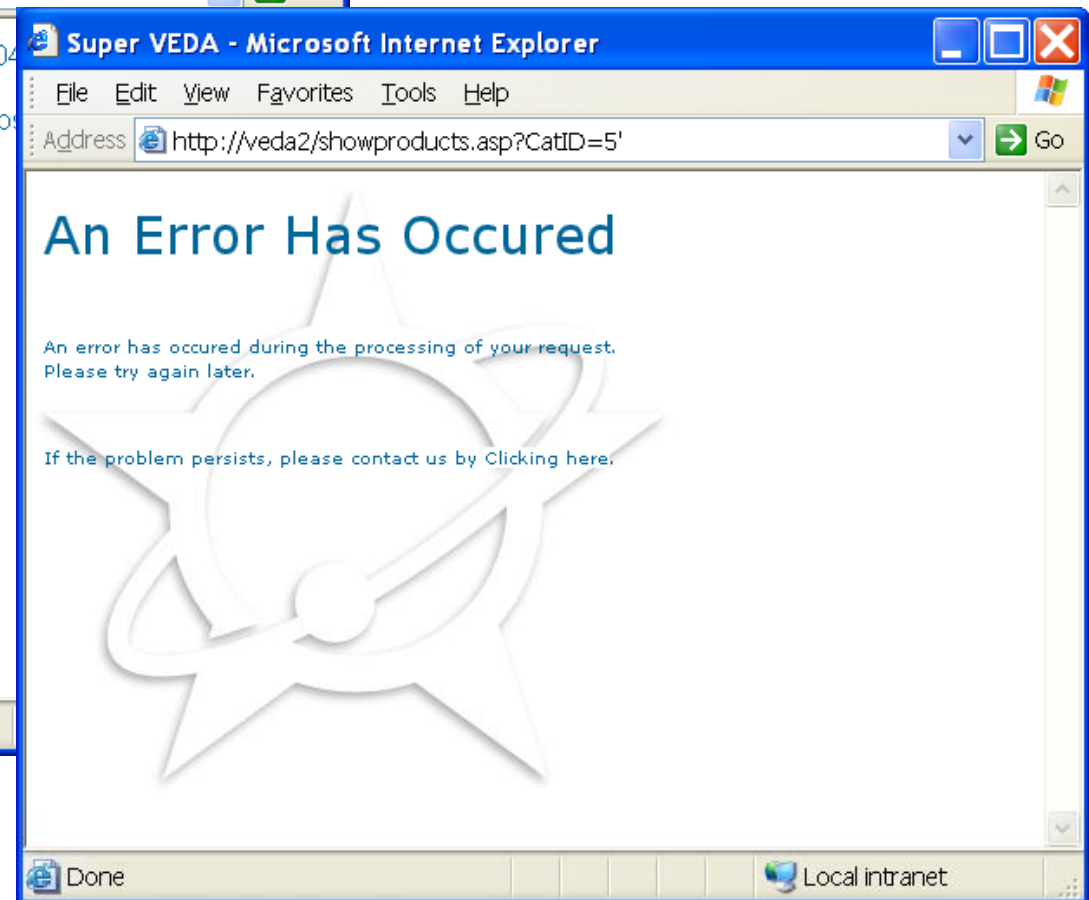
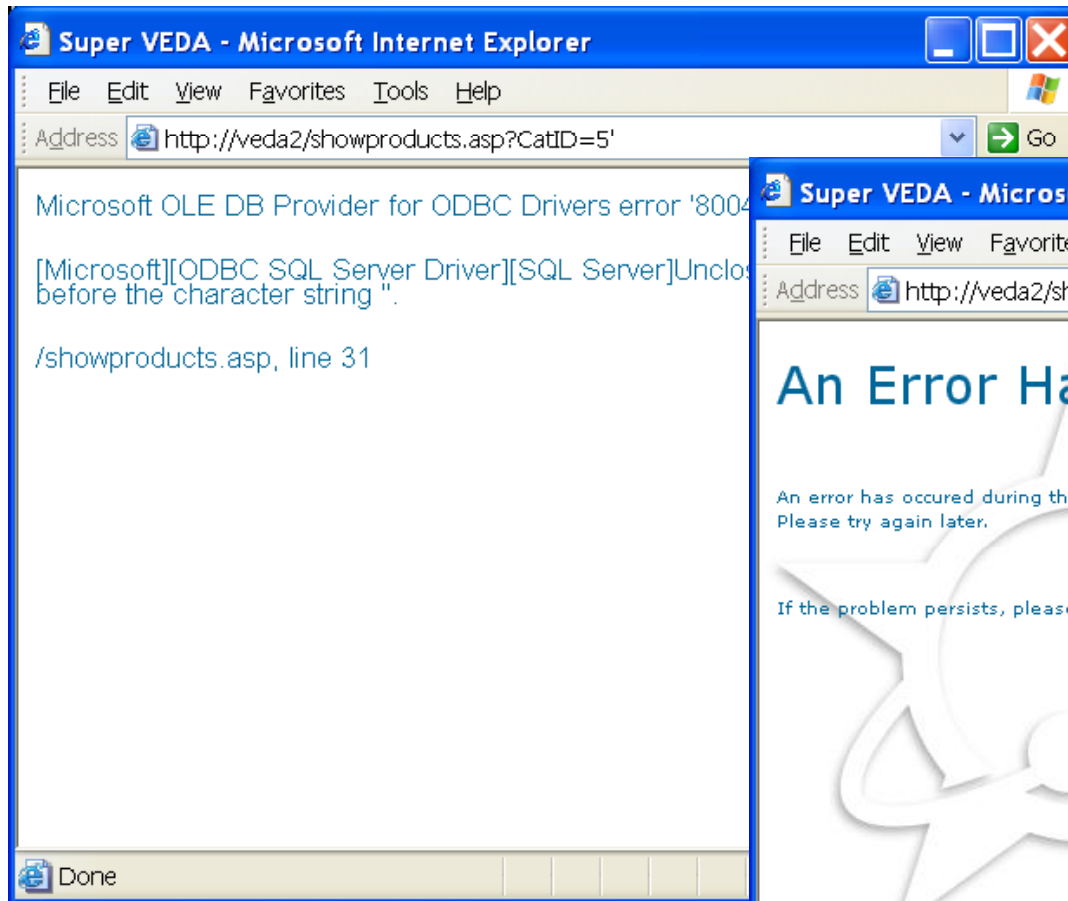
# Solution I: Error Message Hiding

- Simplest and most common countermeasure against SQL Injection
- Based on the notion set by several early SQL Injection papers that relied on detailed error messages for exploitation of SQL Injection vulnerabilities
- Achieved by simple web server configuration options (e.g. suppress error messages or set a custom error message).
- A classic ***Security By Obscurity*** approach. Does not eliminate the vulnerability but rather tries to conceal it

# Solution I: Error Message Hiding



# Solution I: Error Message Hiding



# Blindfolded SQL Injection

- A set of techniques that uses a small number of Boolean tests for detection and exploitation of SQL Injection
- Boolean tests determine whether an error has occurred or not
- Includes techniques for:
  - Identifying the existence of SQL Injection
  - Determining the injection syntax and figuring out the database type
  - Building UNION SELECT exploits
- Other SQL Injection attacks, such as WHERE Statement manipulation and command injection are trivial to construct once the correct syntax is identified

# Blindfolded SQL Injection: Identifying Injections

- Cannot rely on ODBC error messages to identify the injection
- Identifying the occurrence of the error is, however, still possible
- Testing for the existence of SQL Injection can be simply done by replacing a field with equivalent SQL syntax:
  - The number **5** can be represented in SQL as **(6-1)**
  - The string **'test'** can be represented as **'te'+ 'st'** (in MS SQL) or **'te'// 'st'** (in Oracle)
  - A date can be replaced with the database's date function
- Matching results indicate that the system is vulnerable, while an error indicates that the syntax was not parsed by an SQL parser

# Blindfolded SQL Injection: Identifying Injections

The screenshot shows a Microsoft Internet Explorer window titled "Super VEDA - Microsoft Internet Explorer". The address bar contains the URL "http://veda2/showproducts.asp?CatID=5", where the parameter "CatID=5" is circled in red. The page content displays a table of DVD products under the heading "DVD Categorie Products".

Product Name	Price	Availability	
Metropolis (Restored Authorized Edition)	\$29.95	In stock	<a href="#">Add to cart</a>
Harry Potter and The Chamber of Secrets (Widescreen Edition)	\$29.95	In stock	<a href="#">Add to cart</a>
Wild Strawberries - Criterion Collection (1959)	\$39.95	In stock	<a href="#">Add to cart</a>
The Seventh Seal - Criterion Collection (1958)	\$39.95	In stock	<a href="#">Add to cart</a>
The Awful Truth (1937)	\$24.95	In stock	<a href="#">Add to cart</a>

The taskbar at the bottom shows a "Local intranet" icon.

# Blindfolded SQL Injection: Identifying Injections

Super VEDA - Microsoft Internet Explorer

Address <http://veda2/showproducts.asp?CatID=5> Go

### DVD Categorie Products

Product Name	Price
Metropolis (Restored Authorized Edition)	\$29.95
Harry Potter and The Chamber of Secrets (Widescreen Edition)	\$29.95
Wild Strawberries - Criterion Collection (1959)	\$39.95
The Seventh Seal - Criterion Collection (1958)	\$39.95
The Awful Truth (1937)	\$24.95

Super VEDA - Microsoft Internet Explorer

Address [http://veda2/showproducts.asp?CatID=\(6-1\)](http://veda2/showproducts.asp?CatID=(6-1)) Go

### DVD Categorie Products

Product Name	Price	Availability	
Metropolis (Restored Authorized Edition)	\$29.95	In stock	<a href="#">Add to cart</a>
Harry Potter and The Chamber of Secrets (Widescreen Edition)	\$29.95	In stock	<a href="#">Add to cart</a>
Wild Strawberries - Criterion Collection (1959)	\$39.95	In stock	<a href="#">Add to cart</a>
The Seventh Seal - Criterion Collection (1958)	\$39.95	In stock	<a href="#">Add to cart</a>
The Awful Truth (1937)	\$24.95	In stock	<a href="#">Add to cart</a>

Done Local intranet

# Blindfolded SQL Injection: Getting the Syntax Right

- Once an injection is identified, the correct injection syntax is built
- SQL Injection usually occurs inside a WHERE statement, hence the following tests take place
  - Changing the WHERE statement without changing its result, using ***OR 1=2*** or ***AND 1=1***
  - Testing whether the query can be terminated using a comment ( -- )
  - Terminating, if necessary, any open parentheses
- Using some trial and error, the correct syntax can be almost always identified
- Manipulating the WHERE statement is now trivial

# Blindfolded SQL Injection: UNION SELECT Exploits

- A UNION SELECT statement must match the number of fields and type of fields of the original query
- All early SQL Injection papers claimed that UNION SELECT exploits can only be crafted based on detailed error messages
- Detailed errors allowed the attacker to differentiate between different errors:
  - Bad syntax
  - Wrong number of fields
  - Wrong type of fields

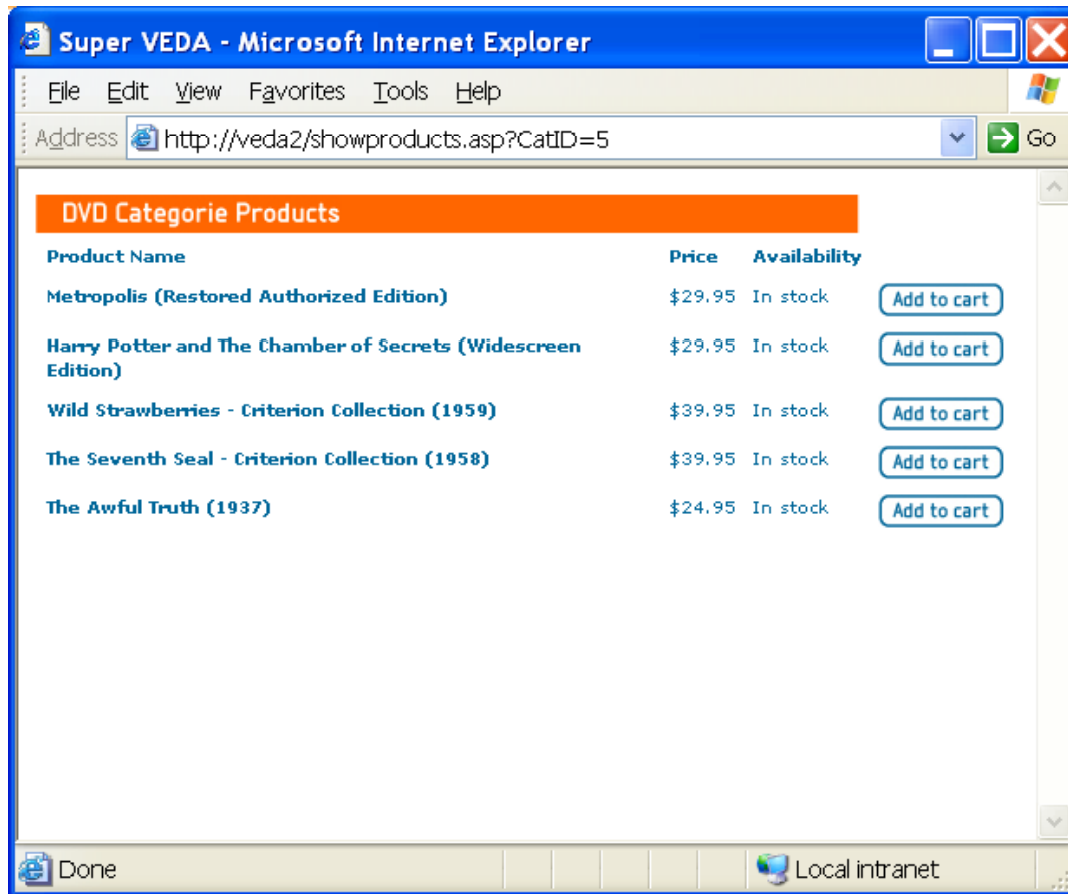
# Blindfolded SQL Injection: UNION SELECT Exploits

- With Blindfolded SQL Injection all errors are identical, hence some form of differentiation must be found
- Step #1 – Enumerating the number of columns
- Step #2 – Enumerating the type of fields
- Step #3 – Do whatever you want!

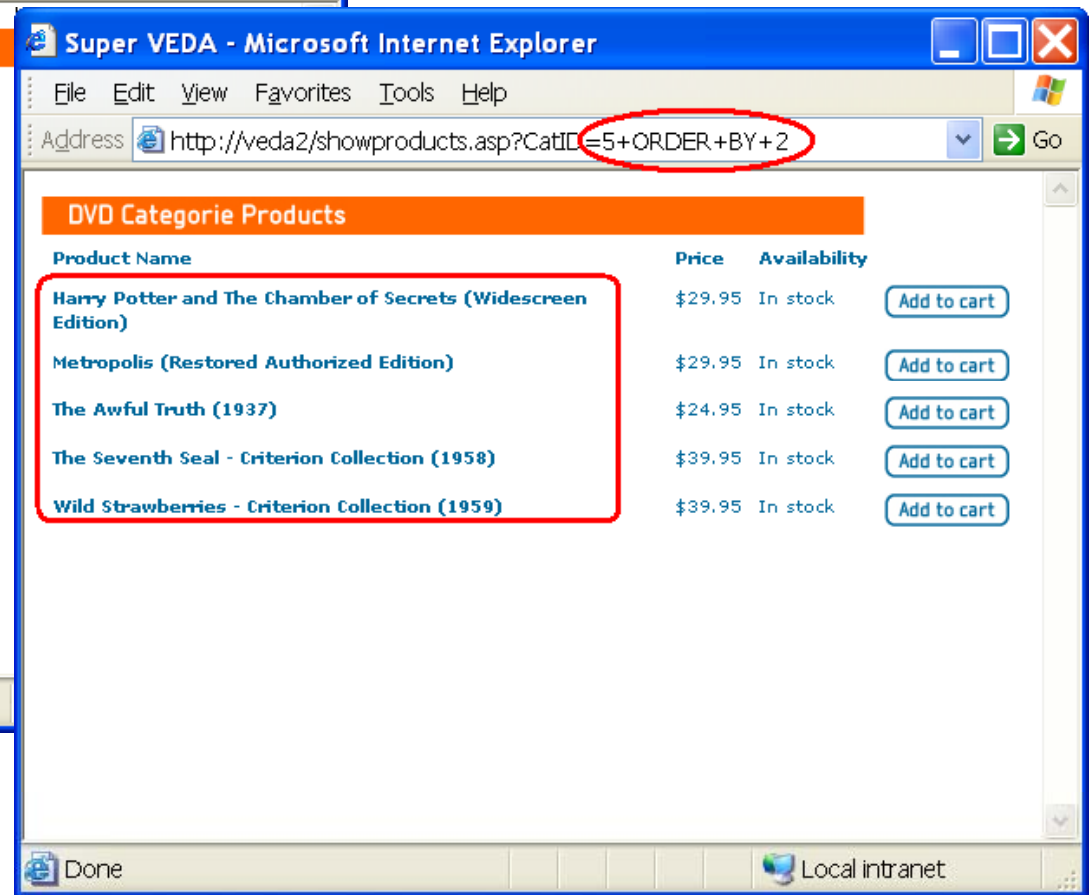
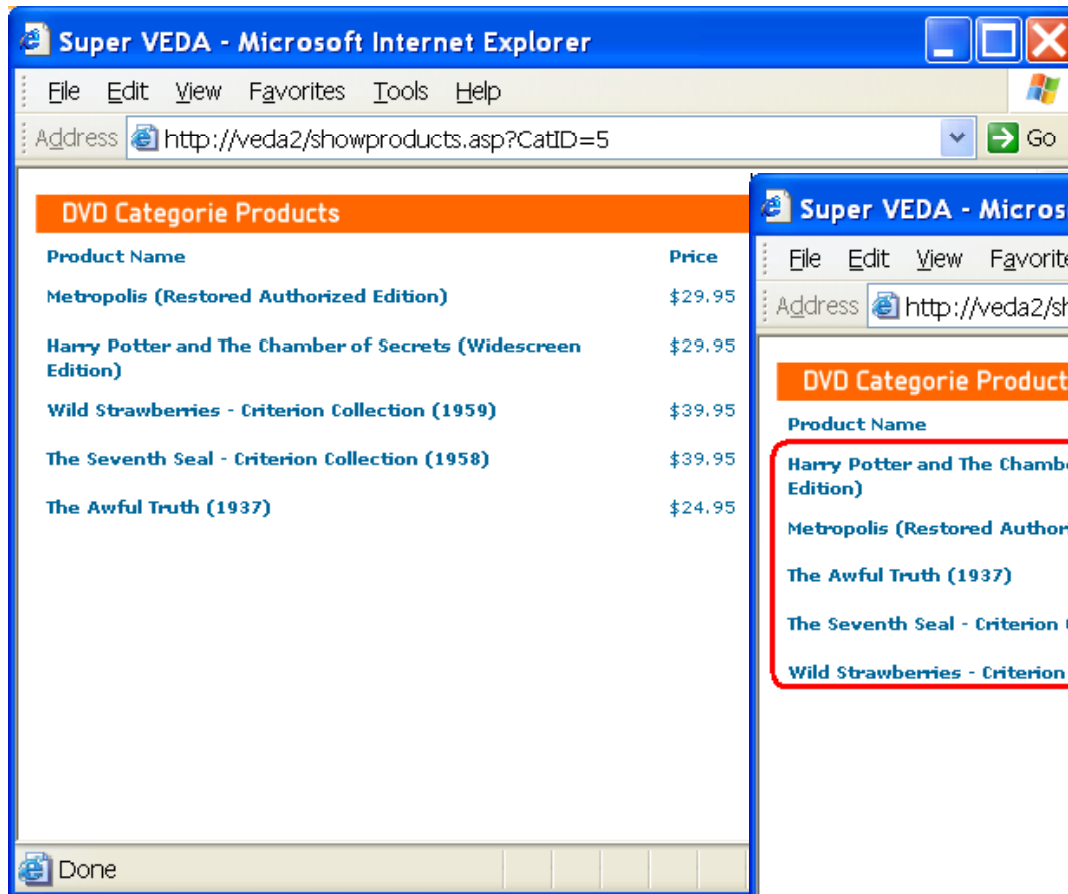
# Blindfolded SQL Injection: UNION SELECT Exploits

- With Blindfolded SQL Injection all errors are identical, hence some form of differentiation must be found
- Step #1 – Enumerating the number of columns
  - Done using an ORDER BY statement
  - ORDER BY sorts the result by a specific field. The field can be specified through its name or through its index (an integer)
  - When an existing field is chosen, the result is sorted according to it. However, when a non-existent field is chosen, an error occurs
  - Using some trial and error, the exact number of fields can be enumerated. Under the assumption that no more than 100 fields are used, no more than  $\log_2(100)$  requests are needed (~7 requests)

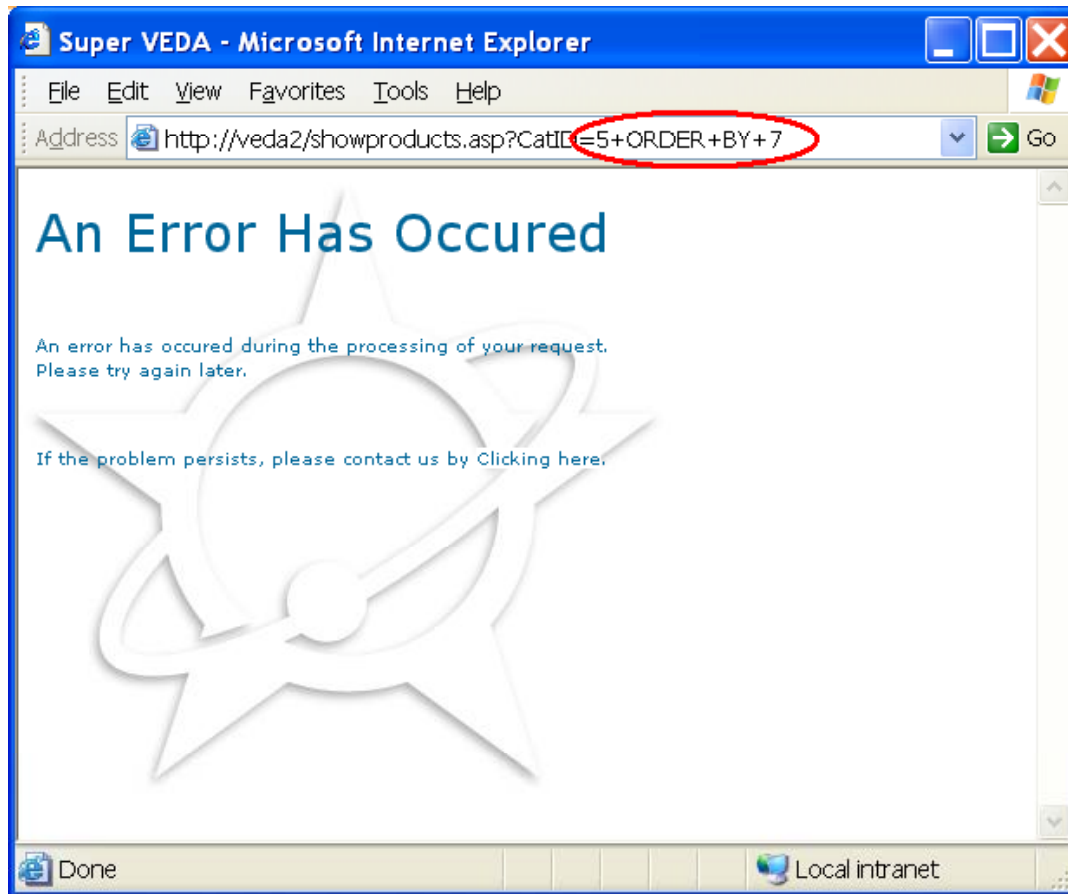
# Blindfolded SQL Injection: UNION SELECT Exploits



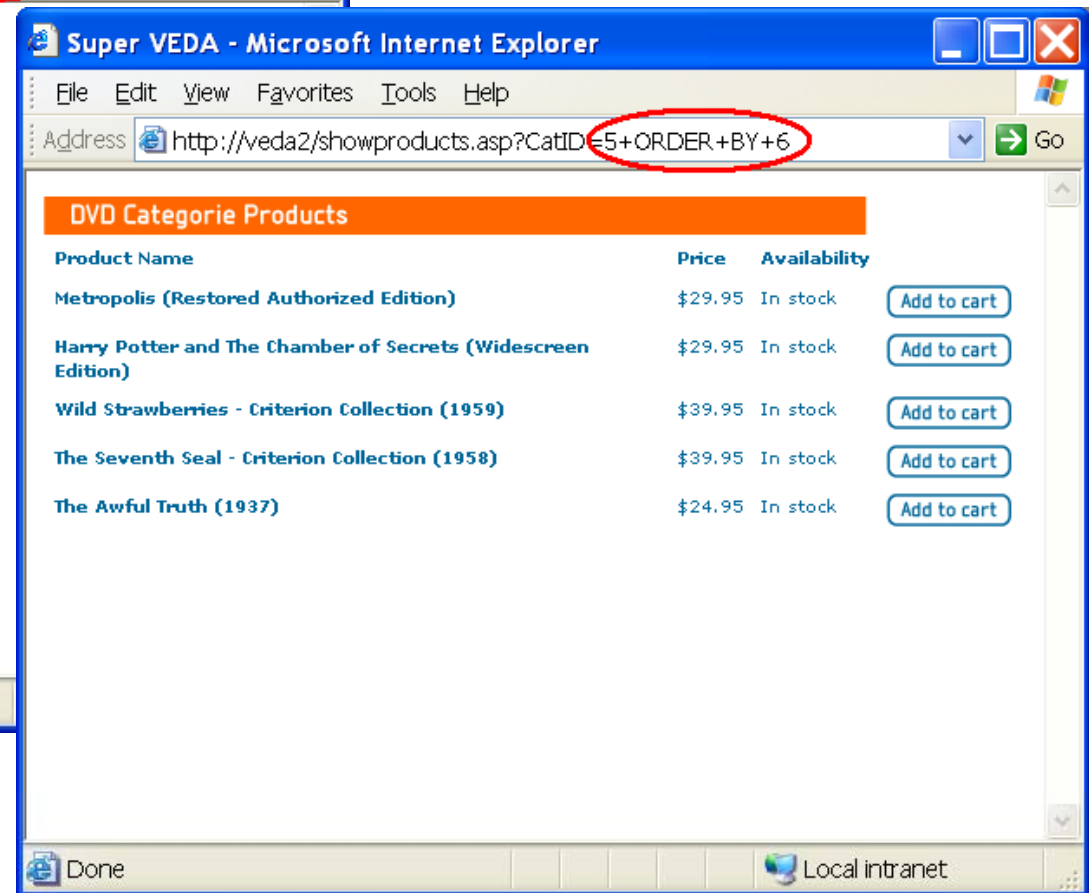
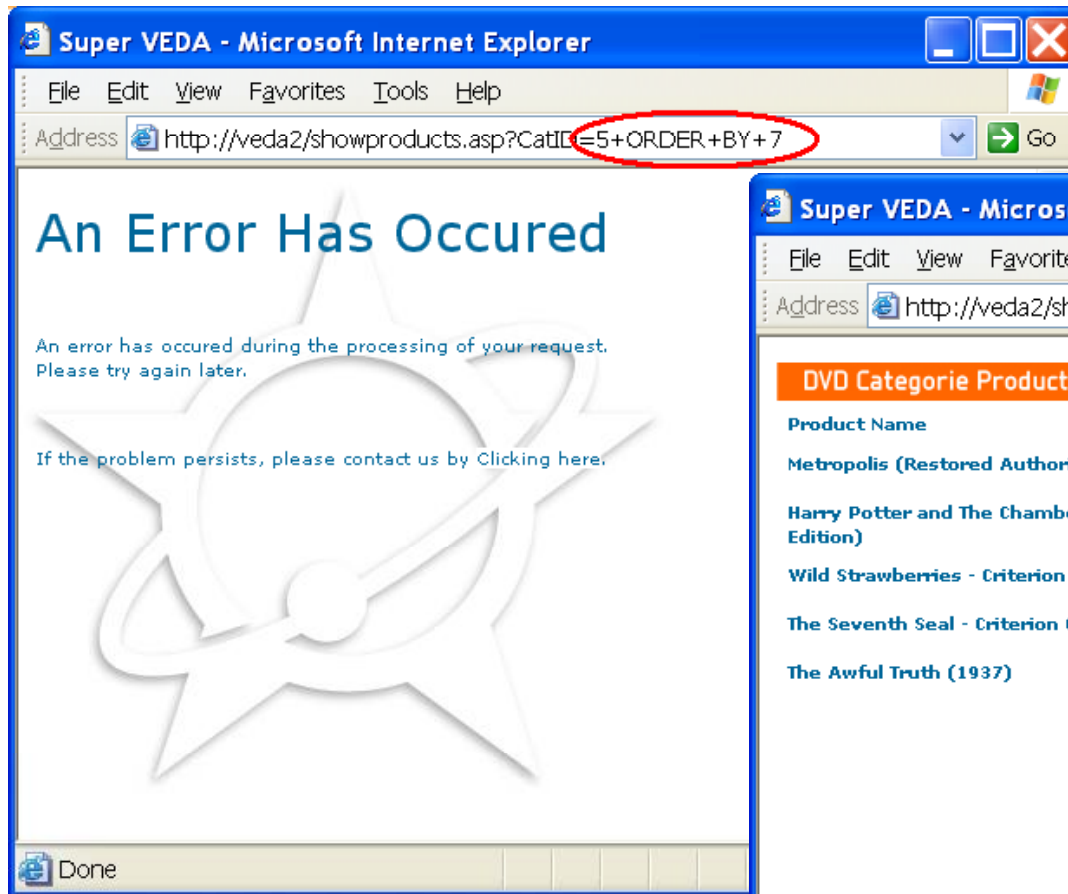
# Blindfolded SQL Injection: UNION SELECT Exploits



# Blindfolded SQL Injection: UNION SELECT Exploits



# Blindfolded SQL Injection: UNION SELECT Exploits



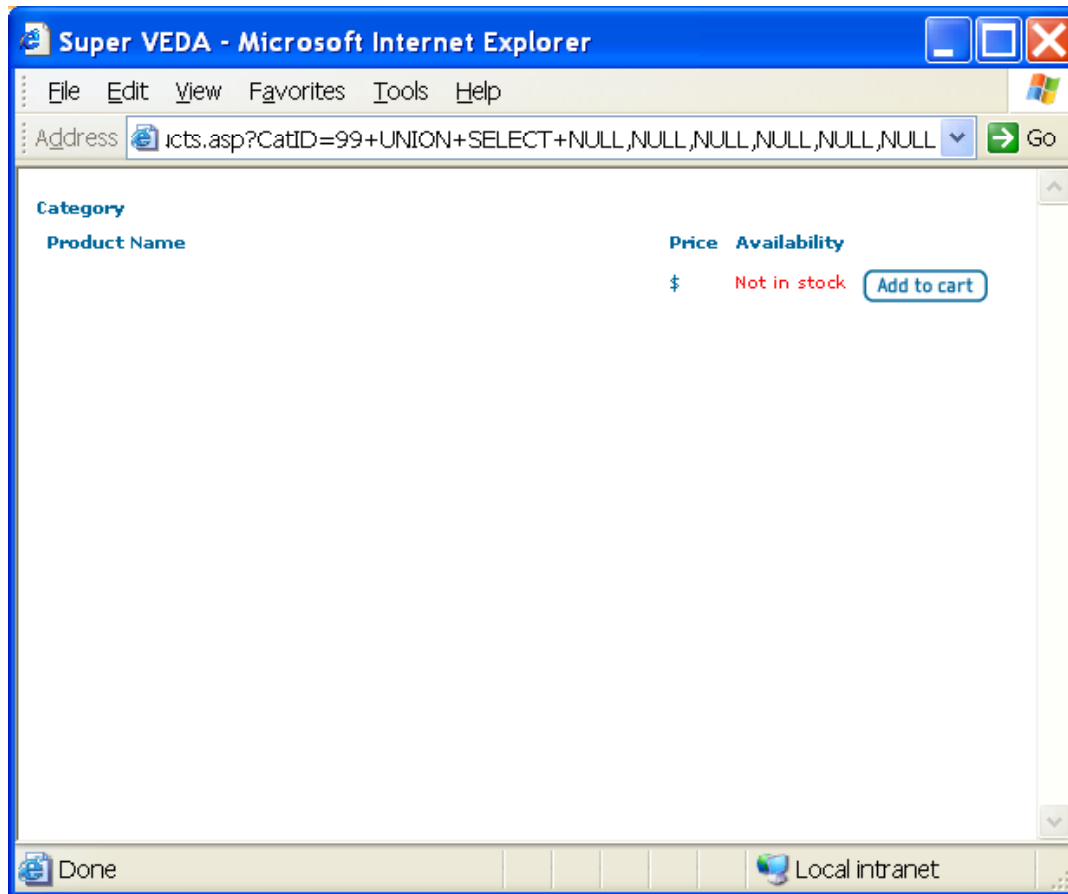
# Blindfolded SQL Injection: UNION SELECT Exploits

- Step #2 – Enumerating the type of fields
  - Create an initial request with all fields set to NULL
  - Type detection is done by guessing one field at a time
  - Allows quick detection of types of all fields.

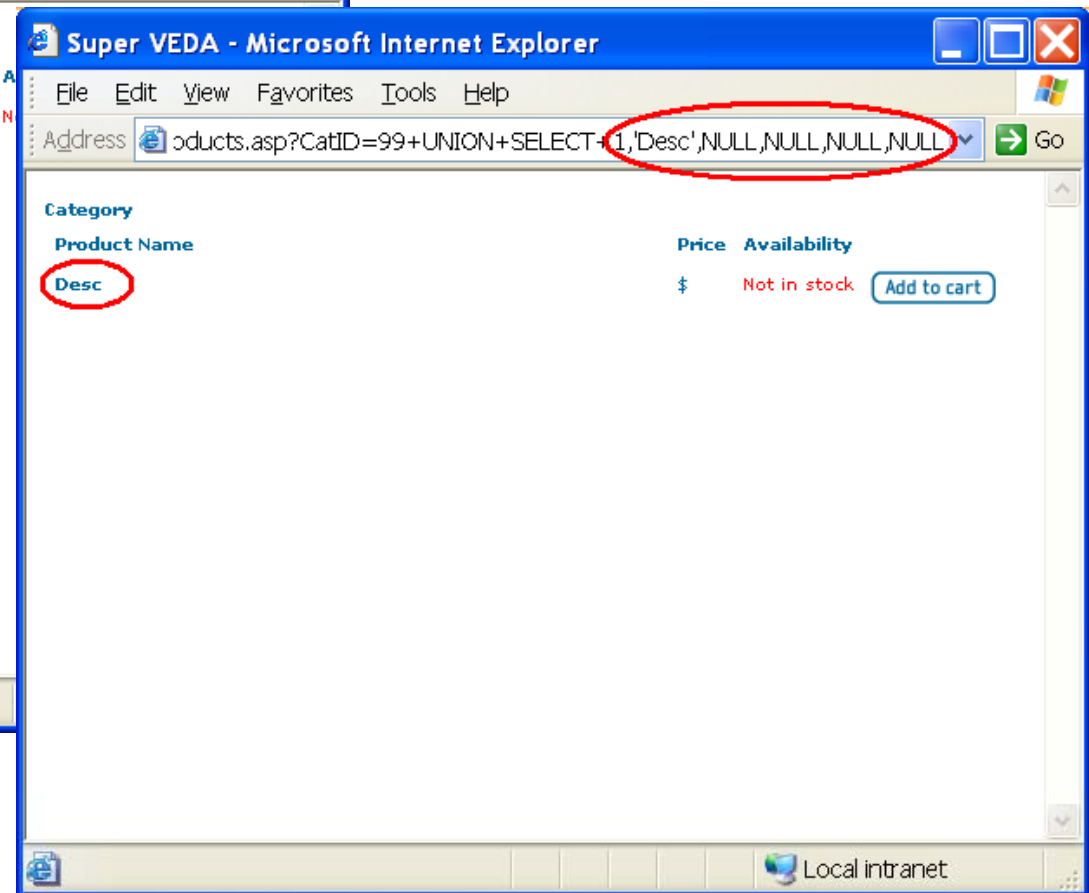
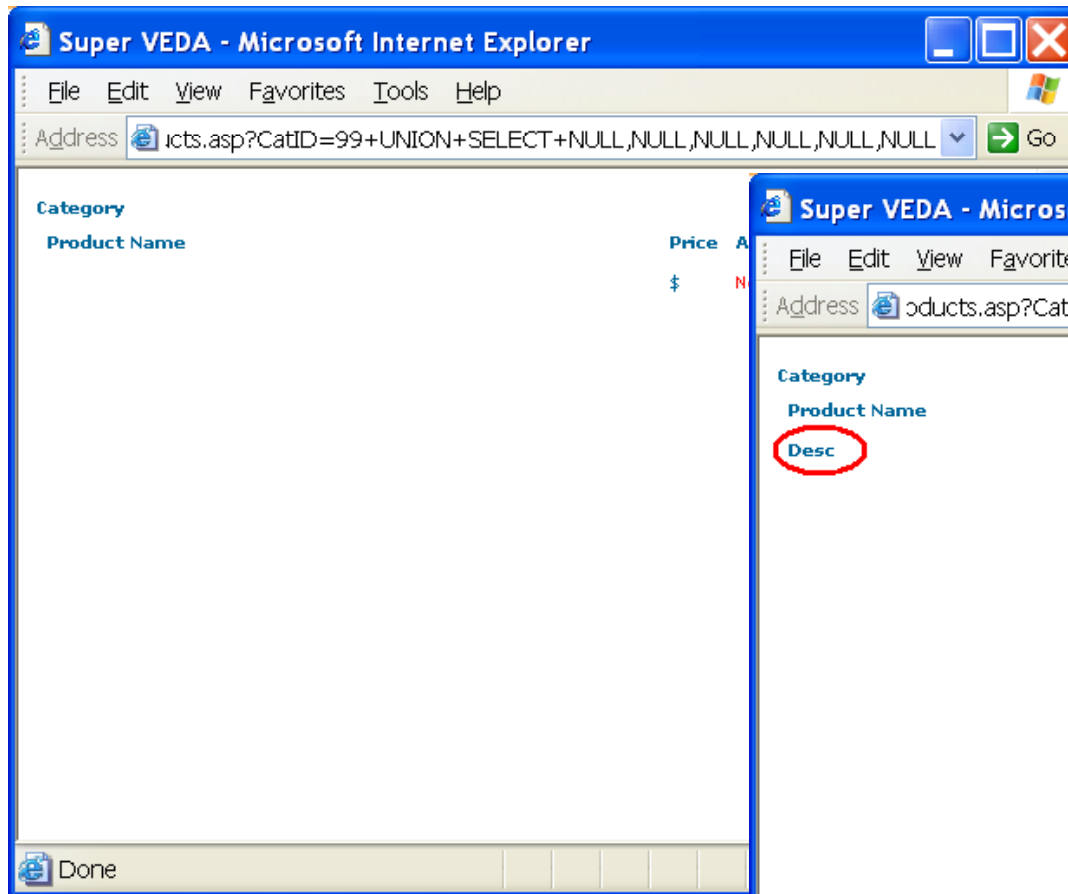
For a given query of  $n$  columns:

  - Using this technique, maximum required attempts =  $n * 3$
  - Using brute force, maximum required attempts =  $3^n$
  - For 10 fields, the difference is between 30 and ~60,000
- Once field types are known, exploitation is trivial

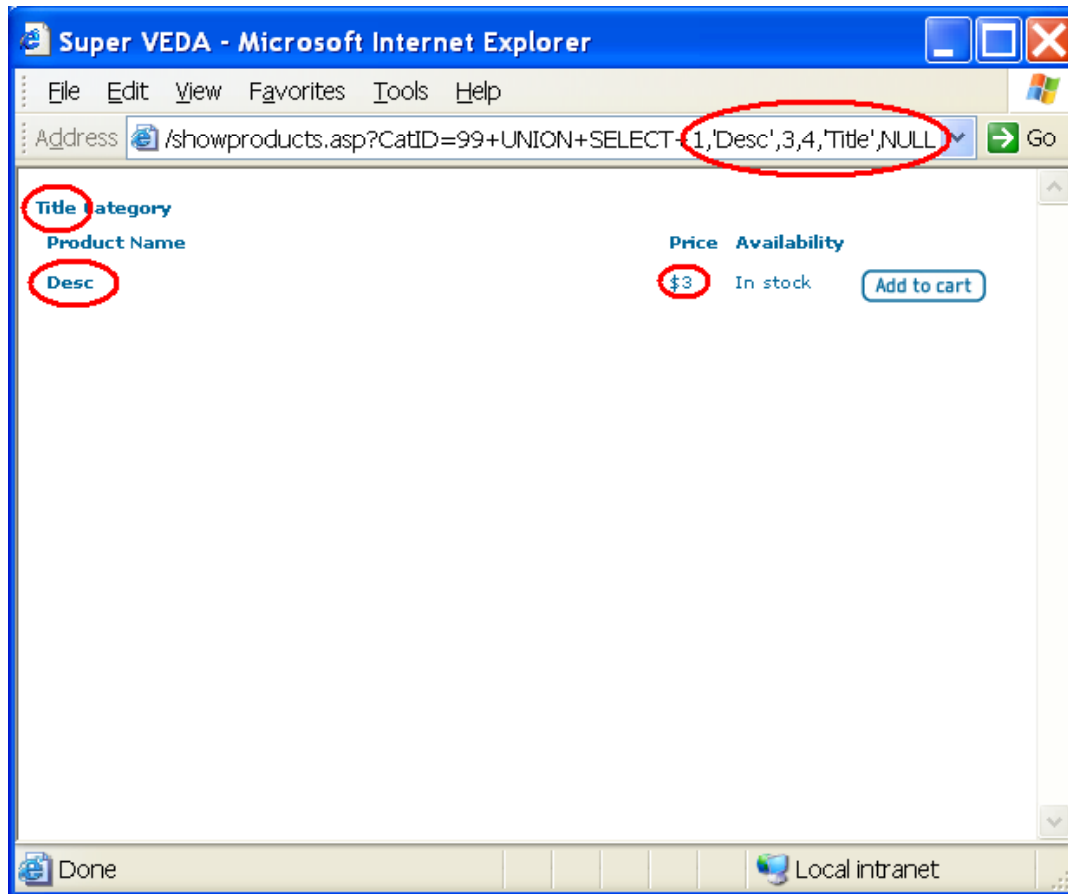
# Blindfolded SQL Injection: UNION SELECT Exploits



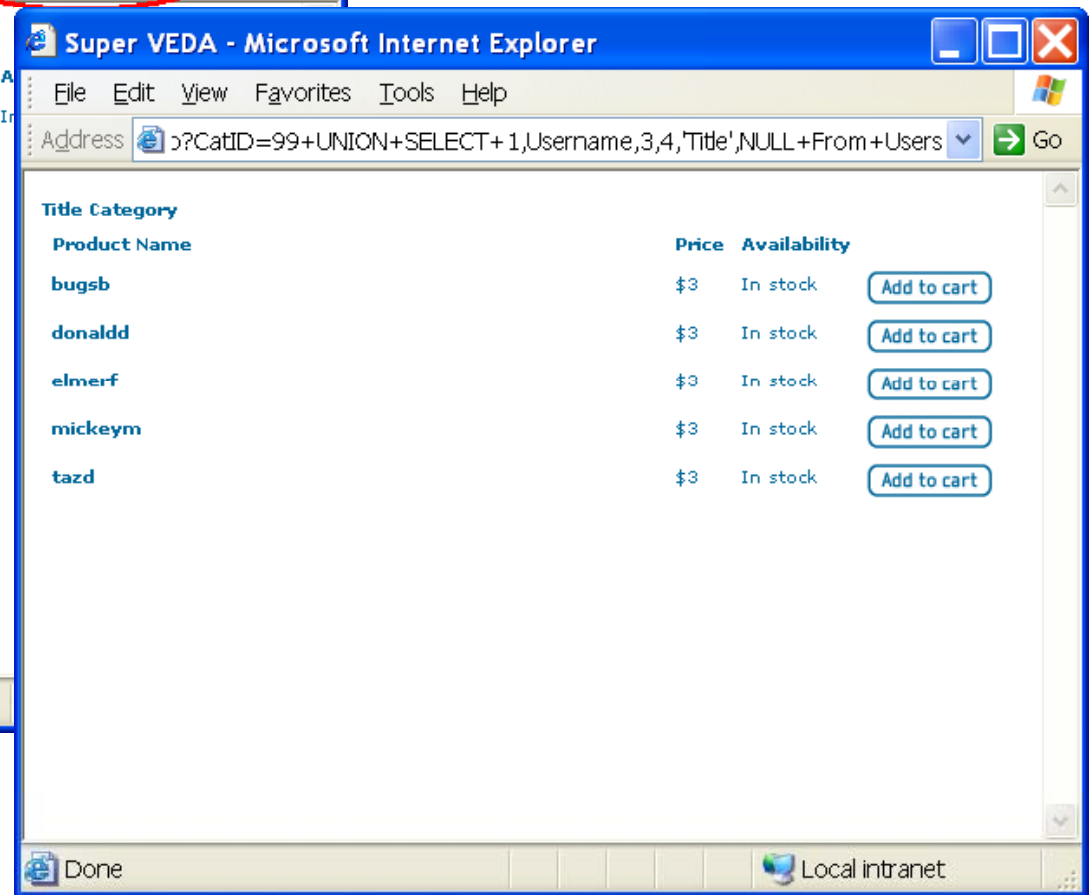
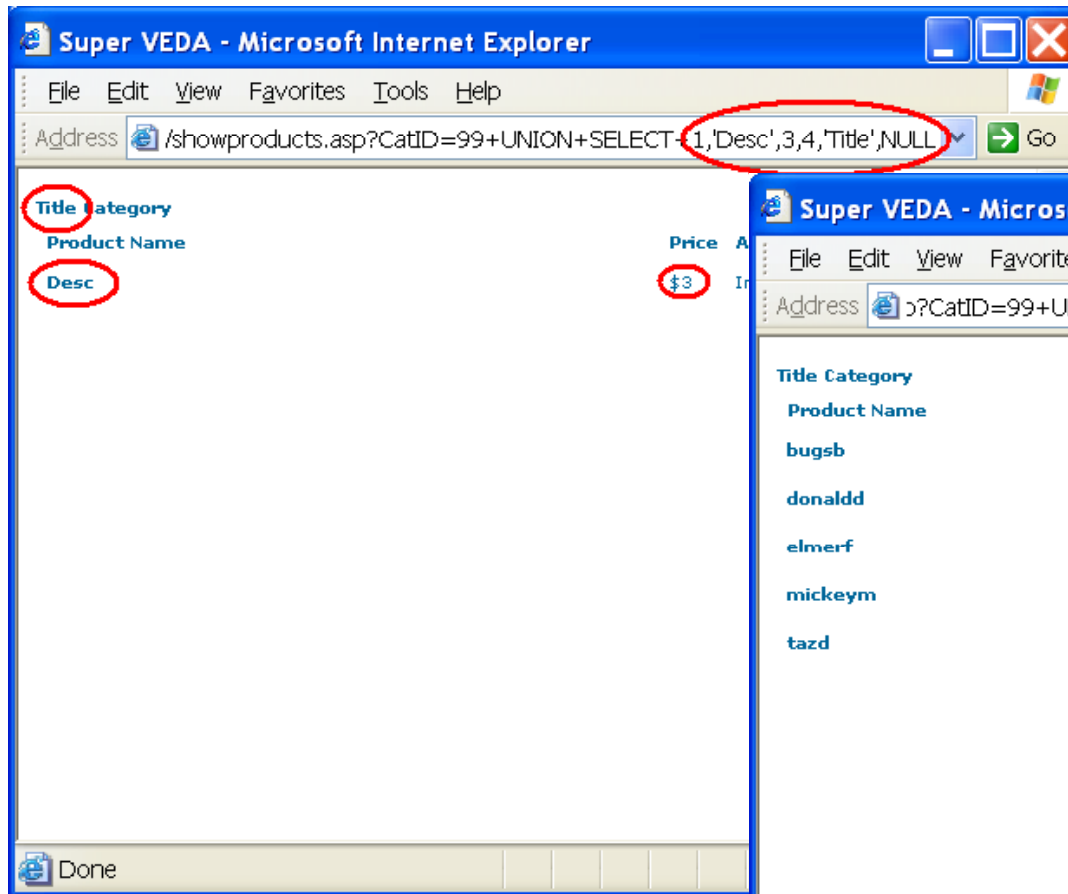
# Blindfolded SQL Injection: UNION SELECT Exploits



# Blindfolded SQL Injection: UNION SELECT Exploits



# Blindfolded SQL Injection: UNION SELECT Exploits



## Solution II: Signature Protection

- Relies on the existing IDS/IPS infrastructure or on an easily installed signature protection component
- Attempts to detect common SQL Injection strings such as UNION SELECT, OR 1=1, ' --, EXEC XP\_CMDSHELL, etc.
- Signatures can only be practically applied to HTTP traffic, as SQL Injection strings are not different than valid SQL statements.
- Placing strict signatures on keywords such as INSERT, SELECT and DELETE, and characters such as ', = and -- will cause the security mechanism to block valid requests
- SQL Language, however, is extremely rich, and it is not practical to cover all possible different exploitations

# SQL Injection Signature Evasion

- A set of techniques which allow an attacker to evade signature protection mechanisms
- Method includes:
  - Detecting signature protection
  - Generic evasion techniques
  - SQL language specific evasion techniques
    - Value Equivalence
    - White Space Equivalence
    - String Equivalence

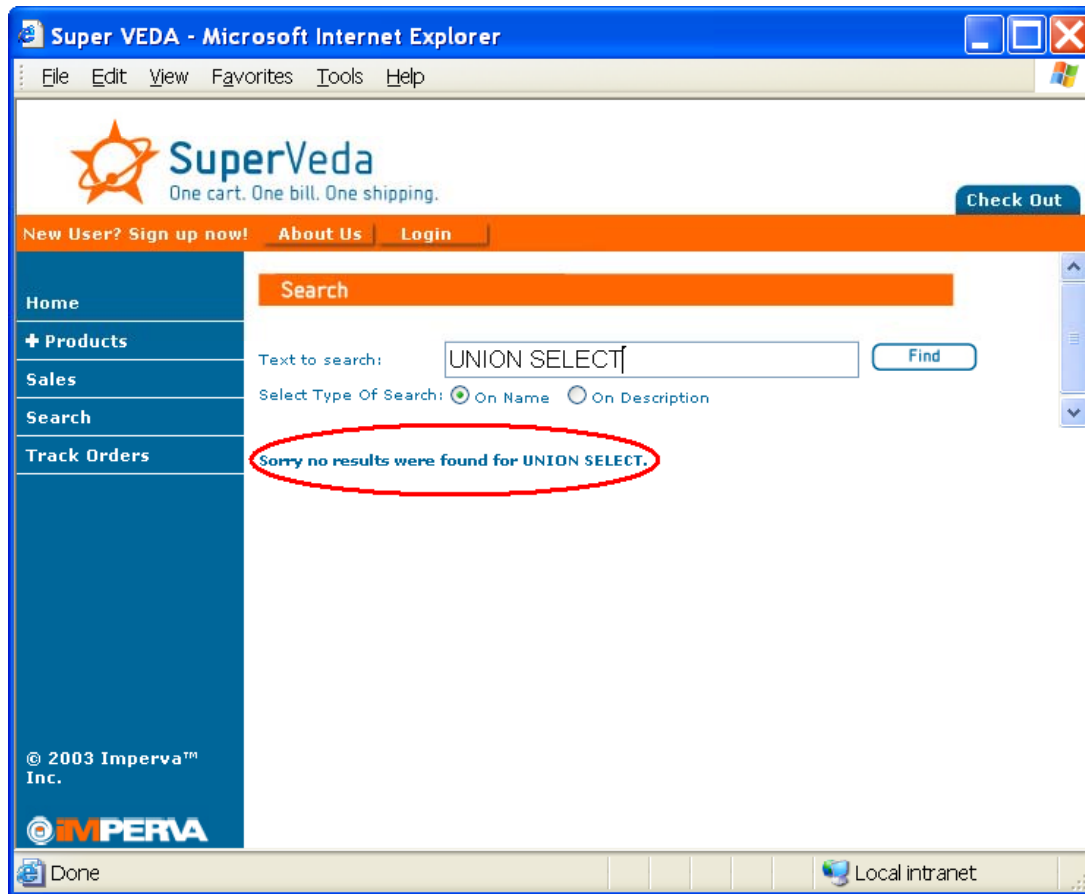
# SQL Signature Evasion: Detecting Signature Protection

- Signature blocking is performed in one of two mechanisms
  - TCP Reset/Packet Drop – Performed by network devices
  - Custom Error Page – Similar to the one displayed upon an unsuccessful injection
- Recognizing network blocks is easy, as it is different than any other error and is a strong indication of a signature detection mechanism (“The page cannot be displayed”)
- When a custom error page is used, it is harder to identify that the error was the result of a signature mechanism rather than a specific context mechanism, and extra validation is required

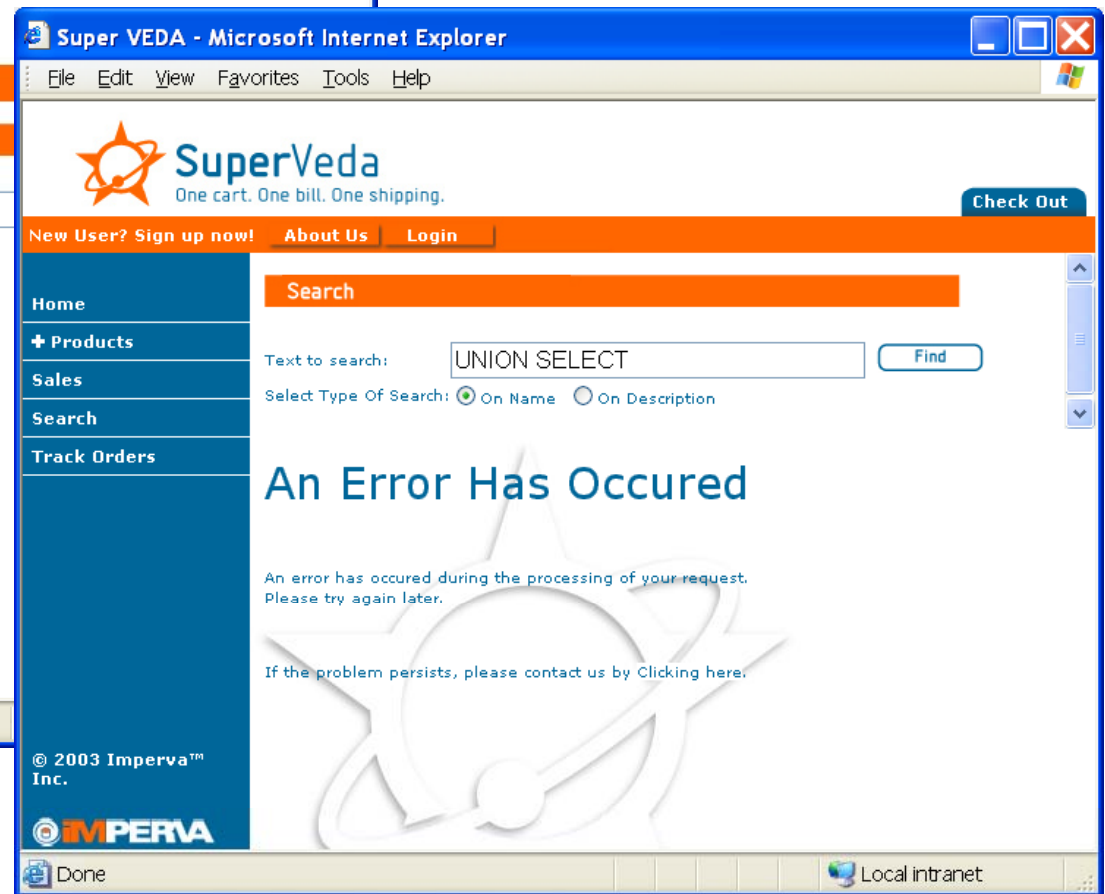
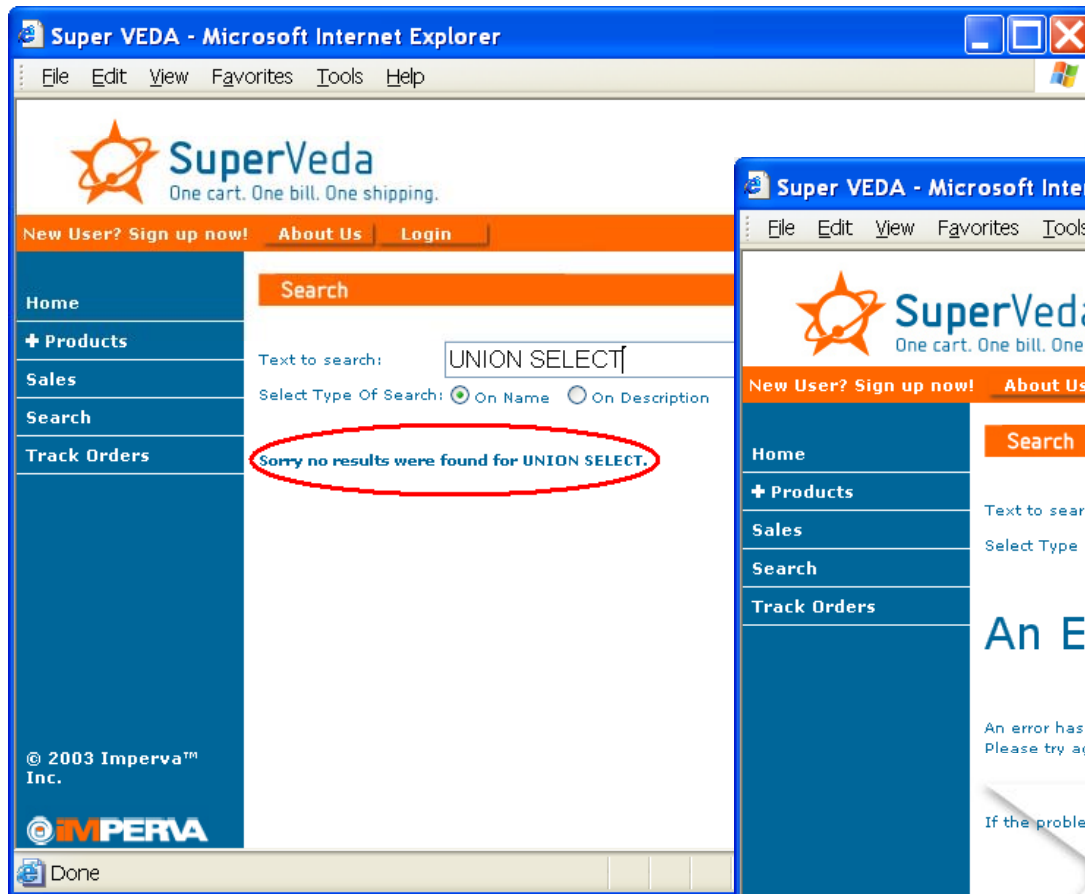
# SQL Signature Evasion: Detecting Signature Protection

- The best method of validation is to rely on the basic characteristics of signature protection:
  - It searches the entire request (or at least the entire Query String)
  - It does simple string (or regexp) matching
  - It works in a similar manner throughout the entire application
  - It does *not* understand the context of the application
- The solution is therefore replacing a benign string, such as a search string, with a suspected SQL Injection string, without actually escaping to the SQL syntax (for instance, UNION SELECT, with no preceding quote)

# SQL Signature Evasion: Detecting Signature Protection



# SQL Signature Evasion: Detecting Signature Protection



# SQL Signature Evasion: Generic Evasion Techniques

- Non-SQL Specific
- Takes advantage of common IDS evasion techniques, such as:
  - IP Fragmentation
  - TCP Segmentation
  - White Space Diversification
  - Various Encodings (HTTP/UTF8/Unicode/etc)
- Vulnerability to these techniques is a result of poor implementation rather than an inherent problem
- Most commercial IDS/IPS offerings are resilient to such techniques

# SQL Signature Evasion: SQL Based Techniques

- Advanced SQL Signature evasion techniques take advantage of the inherent problem of SQL Injection signature detection - the diversity of the SQL Language
- Technique #1: Value Equivalence
  - Used to evade signatures that detect WHERE statement manipulation attacks such as OR 1=1
  - Evasion is done by replacing the 1=1 (or '1'='1') with any other SQL expression returning a TRUE value
  - While the OR keyword remains a part of the attack, it is practically impossible to place a signature on that string

# SQL Signature Evasion: SQL Based Techniques

- Value Equivalence Examples:
  - ... OR 'Simple' = N'Simple'
  - ... OR 'Simple' = 'Sim'+ 'ple'
  - ... OR 'Simple' LIKE 'Sim%'
  - ... OR 2 > 1
  - ... OR 'Simple' > 'S'
  - ... OR 2 BETWEEN 1 AND 3
  - ... OR 'Simple' BETWEEN 'R' AND 'T'
  - ... OR 1 IN (1)

# SQL Signature Evasion: SQL Based Techniques

- Technique #2 – White Space Equivalence
  - Used to evade signatures that contain white spaces, such as
    - OR 1=1
    - UNI ON SELECT
    - EXEC SP\_
  - While white spaces are normally used as delimiters between keywords and values in SQL syntax, many parsers allow them to be omitted or replaced with non parsed strings. This allows reproducing attacks without the expected white spaces usage
  - Implementation may vary between different database vendors

# SQL Signature Evasion: SQL Based Techniques

- White Space Equivalence Examples:
  - Omitting all white spaces (Works for MS SQL)
    - ... OR' Si mpl e' =' Si mpl e'
    - ... OR' S' BETWEEN' R' AND' T'
  - Using Comments
    - ... UNI ON /\*Nothi ng\*/ SELECT
    - ... OR/\*\*/1/\*\*/=/\*\*/1
  - Using more than one parameter
    - A Login URL: ...l ogi n. asp?User=X' OR' 1' /\*&Pass=Y\*/=' 1
    - Result Query: ...WHERE User=' X' OR' 1' /\* AND Pass=' \*/=' 1'

# SQL Signature Evasion: SQL Based Techniques

- Technique #3 – String Equivalence

- Allows evasion of any string signature by replacing it with an equivalent, yet different, string
- The basic string equivalence is done by executing a concatenated string (Most DB's have more than one way of doing so), such as:  
... ; EXEC(' INS' + 'ERT INTO...' )
- Another possible string equivalence is through its hexadecimal representation, allowing the keyword *SELECT* to be represented as `0x73656c656374`
- Additionally, in some databases, comments can be placed in the middle of keywords, breaking the string in the middle, as follows:

... UN/\*\*/ION SEL/\*\*/ECT ...

## Solution III: Database ACLs

- Keeping database access privileges to a minimum for the application account according to the least privileges principle
- Should significantly reduce the risk for non application-specific components in the database, and to some extent reduce the risk to application data integrity. Does not affect the risk for application data confidentiality at all.
- Protects the database against system level attacks that require special system privileges, such as the following:

```
; EXEC MASTER.XP_CMDSHELL(' cmd. exe /e di r' ) --  
; SHUTDOWN --  
; DROP DATABASE MyApp --
```

# SQL Injection Denial of Service

- A set of techniques which allow an attacker to launch Denial of Service attacks against databases through SQL Injection
- Basic DoS techniques require the application to be running with a privileged user, which can perform tasks such as *SHUTDOWN*, *DROP TABLE*, *DROP DATABASE*, etc.
- Advanced techniques, however, allow the attacker to perform various destructive activities (either making the server unavailable or corrupting application data), through a user with limited privileges

# SQL Injection DoS: Data Corruption/Destruction

- While not a classic DoS attack, Data destruction/corruption may often render the application useless
- Recovery time may be significantly longer – Instead of a reboot, data restoration is required
- Can often be done on pages which perform DELETE or UPDATE statements, based on a parameter provided by the user
- Injecting an *OR 1=1* (or equivalent) string will cause the query to delete or alter the entire contents of the table. For instance, injecting into a password change form:

```
UPDATE Users SET Password='BOGUS' WHERE User='User' OR '1'='1'
```



# SQL Injection DoS: Resource Consumption

- Resource consumption attacks can be done with a read-only user and can prevent others from using the server
- Can be performed through several techniques, such as:
  - Creating an large record set created from a correlated query:

```
SELECT A1.* , B1.* FROM A AS A1, B AS B1
WHERE EXISTS (SELECT A2.* , B3.* FROM A AS A2, B AS B3
              WHERE A1.AID = A2.AID)
AND EXISTS (SELECT B2.* , A3.* FROM B AS B2, A AS A3
            WHERE B1.BID = B2.BID)
```

- Executing endless loops:

```
BEGIN DECLARE @A INT;
          WHILE (1=1) BEGIN
            IF (1=2) BEGIN
              SET @A = 1;
            END
          END
END
```

# SQL Injection: The Right Solution

- **The Right Solution** – Applying security to the application and/or relevant code
- Should take place in 3 different layers
  - The Application Itself – Writing secure code, which is not vulnerable to SQL Injection
  - The Database – Applying various security mechanisms that are built in to the database
  - External Mechanism – Relying on external software or hardware, that is aware of the application context
- Infrastructure solutions can provide additional security

# The Right Solution – Secure Coding

- Three major considerations when writing database access code:
  - Use Prepared Statements/Parametric Queries – Unlike string queries, these are more complex database access methods, which are invulnerable to SQL Injection (and in most situations yield better database performance)
  - Use Stored Procedures – Perform all database access through stored procedures, and provide the application user with permissions limited to these stored procedures only
  - Input Sanitation – Always verify that all input is within expected range, in terms of length, type and character set

# The Right Solution – Secure Coding

- Take this back to your programmers (Example in C#):

```
...
// Defining the Query with @PrID as its parameter
String StrQry = "SELECT * FROM Products Where ProdID = @PrID";

// Creating the connection and the SQL Command
SqlConnection MyConn = new SqlConnection(ConnectionString);
SqlCommand MyCmd = new SqlCommand(StrQry, MyConn);

// Creating and setting the parameter
MyCmd.Parameters.Add(new SqlParameter("@PrID", SqlDbType.Int));
MyCmd.Parameters["@PrID"].Value = Request.QueryString["ProdID"];

// And Execute
MyConn.Open();
SqlDataReader SqlDR = MyCmd.ExecuteReader();
...
```

# The Right Solution – Database Security Mechanisms

- Three database mechanisms to be used:
  - Strict database permissions, while using different users for different types of tasks (for instance, only the login page needs actual access to the Users table). If stored procedures are used, permissions can be limited to the usage of the stored procedures
  - Resource Quotas – Limiting the resource consumption permitted for a single user and/or a single connection reduces the risk of Denial of Service abuse
  - Audit Log – Logging every query sent by the application allows identifying and analyzing attacks after they occur, and are important for compliance with most security/privacy regulations

# The Right Solution – External Mechanism

- An external security mechanism that is designed to be aware of the application context (implemented either as a separate device or externally installed software), provides another layer of security on top of the ones performed by the programmers
- Can replace or revalidate some of the security tasks already described, such as input validation and logging
- It can also perform many tests on incoming requests and outgoing responses based on expected behavior
- Being aware of the application context and the protocol specifics such a device can be used to protect database servers from a multitude of clients and not only web servers.



---

# Thank You

Imperva, Inc.

3400 Bridge Parkway, Suite 101, Redwood Shores CA 94065

Sales: +1-866-926-4678 [www.imperva.com](http://www.imperva.com)

