CS 380S

0x1A Great Papers in Computer Security

Vitaly Shmatikov

http://www.cs.utexas.edu/~shmat/courses/cs380s/

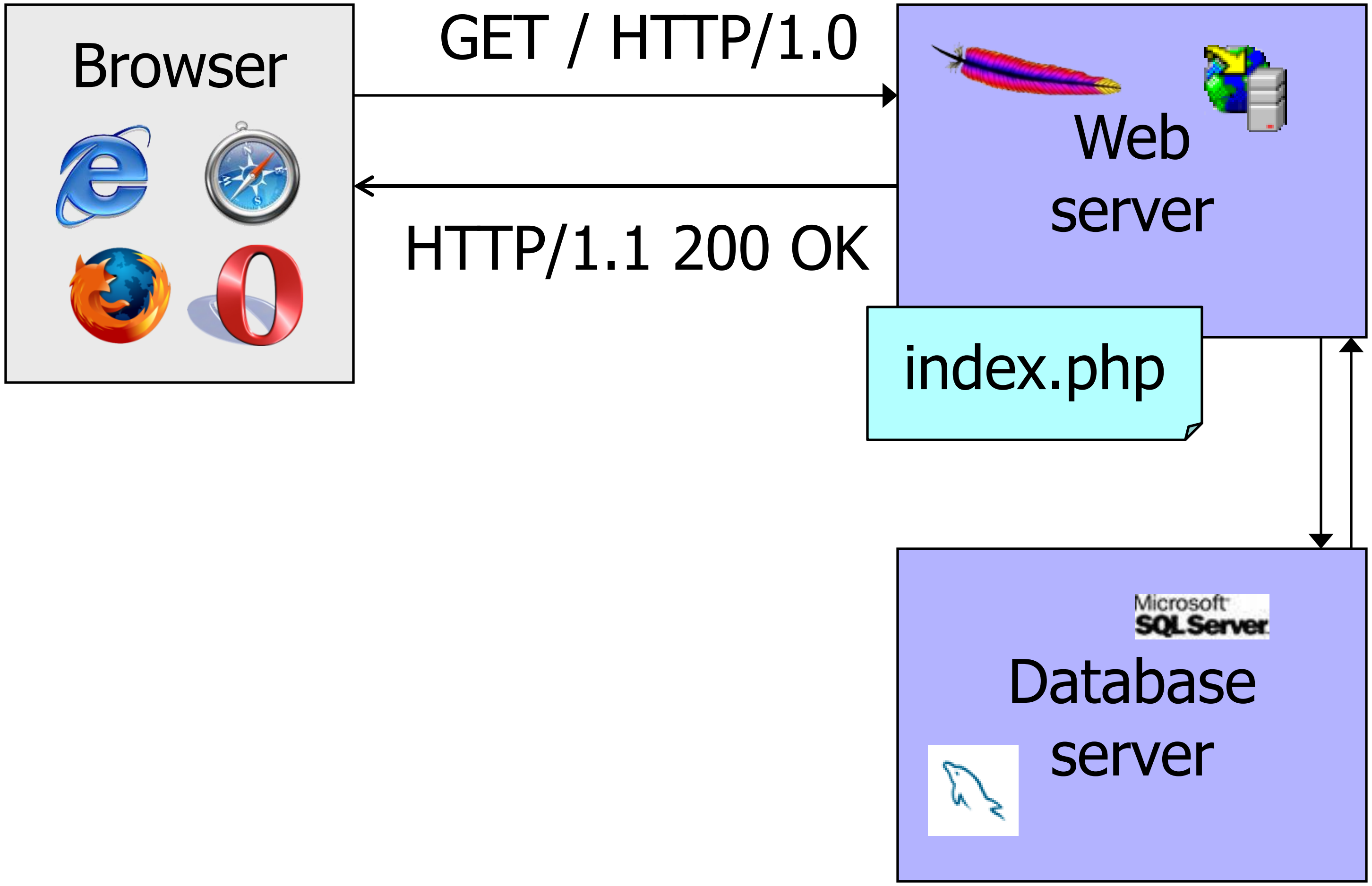
Web Applications

- Big trend: software as a Web-based service
  - Online banking, shopping, government, bill payment, tax prep, customer relationship management, etc.
  - Cloud computing
- Applications hosted on Web servers
  - Written in a mixture of PHP, Java, Perl, Python, C, ASP
- Security is rarely the main concern
  - Poorly written scripts with inadequate input validation
  - Sensitive data stored in world-readable files
  - Recent push from Visa and Mastercard to improve security of data management (PCI standard)

Typical Web Application Design

- Runs on a Web server or application server
- Takes input from remote users
- Interacts with back-end databases and third parties
- Prepares and outputs results for users
  - Dynamically generated HTML pages
  - Content from many different sources, often including users themselves
    - Blogs, social networks, photo-sharing websites…

Dynamic Web Application



PHP: Hypertext Preprocessor

- Server scripting language with C-like syntax
- Can intermingle static HTML and code

```
<input value=<?php echo $myvalue; ?>>
```

- Can embed variables in double-quote strings

```
   $user = “world”; echo “Hello $user!”;
or $user = “world”; echo “Hello” . $user . “!”;
```

- Form data in global arrays $_GET, $_POST, …

SQL

- Widely used database query language
- Fetch a set of records

```
SELECT * FROM Person WHERE Username='Vitaly'
```

- Add data to the table

```
INSERT INTO Key (Username, Key) VALUES ('Vitaly', 3611BBFF)
```

- Modify data

```
UPDATE Keys SET Key=FA33452D WHERE PersonID=5
```

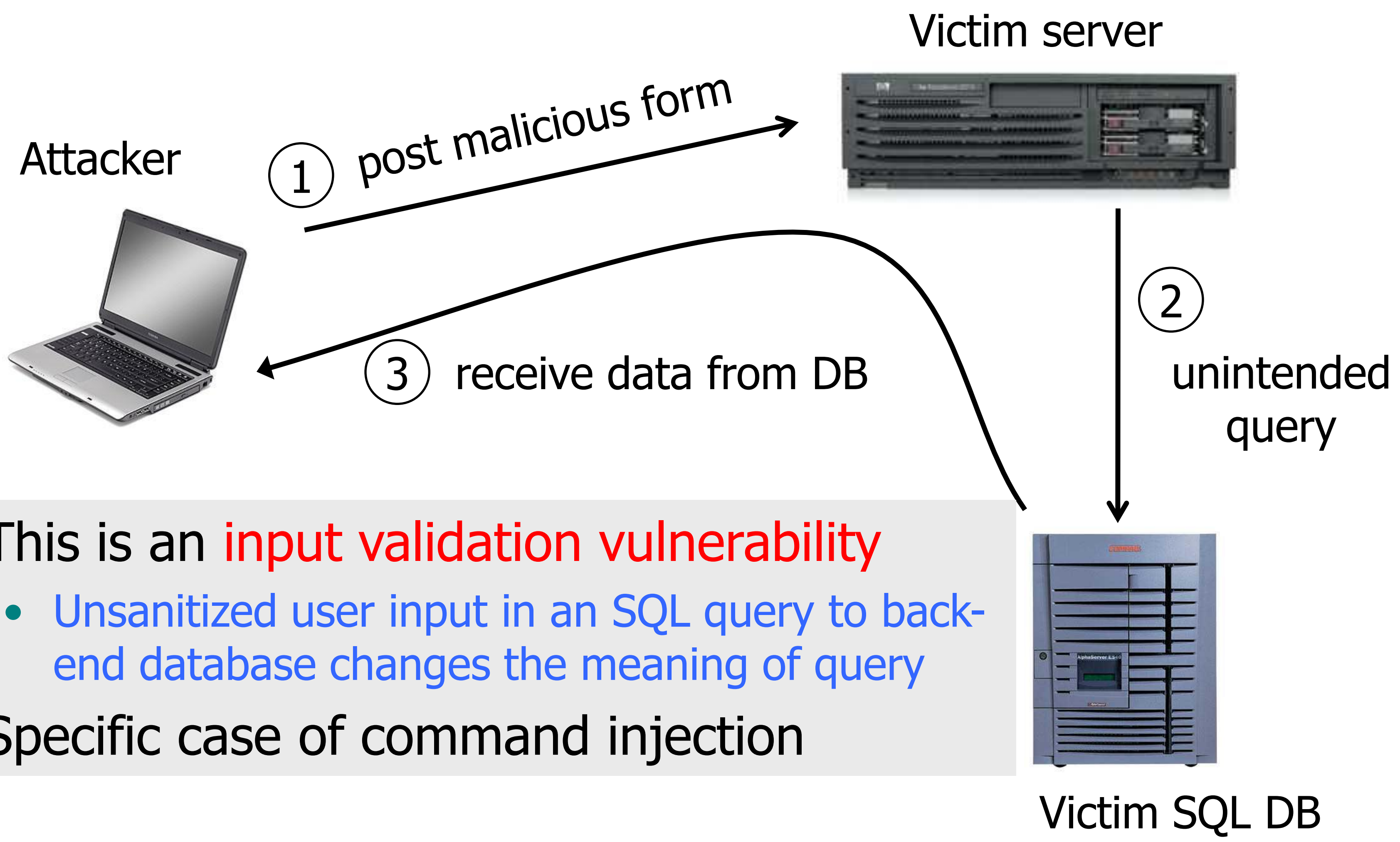
- Query syntax (mostly) independent of vendor

Sample Code

```
$selecteduser = $_GET['user'];
$sql = "SELECT Username, Key FROM Key " .
          "WHERE Username='$selecteduser'";
$rs = $db->executeQuery($sql);
```

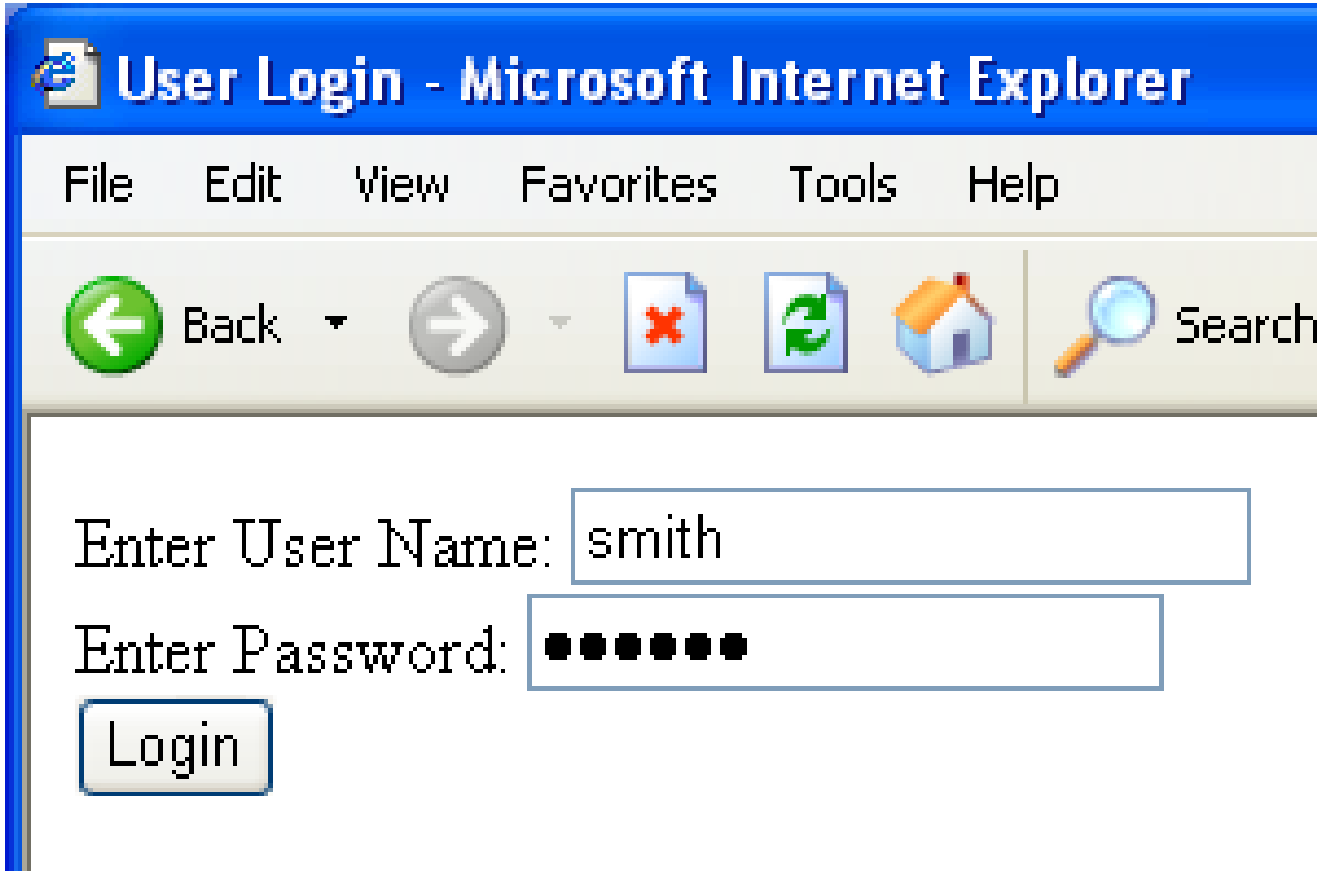
- What if 'user' is a malicious string that changes the meaning of the query?

SQL Injection: Basic Idea



- This is an input validation vulnerability
  - Unsanitized user input in an SQL query to back-end database changes the meaning of query
- Specific case of command injection

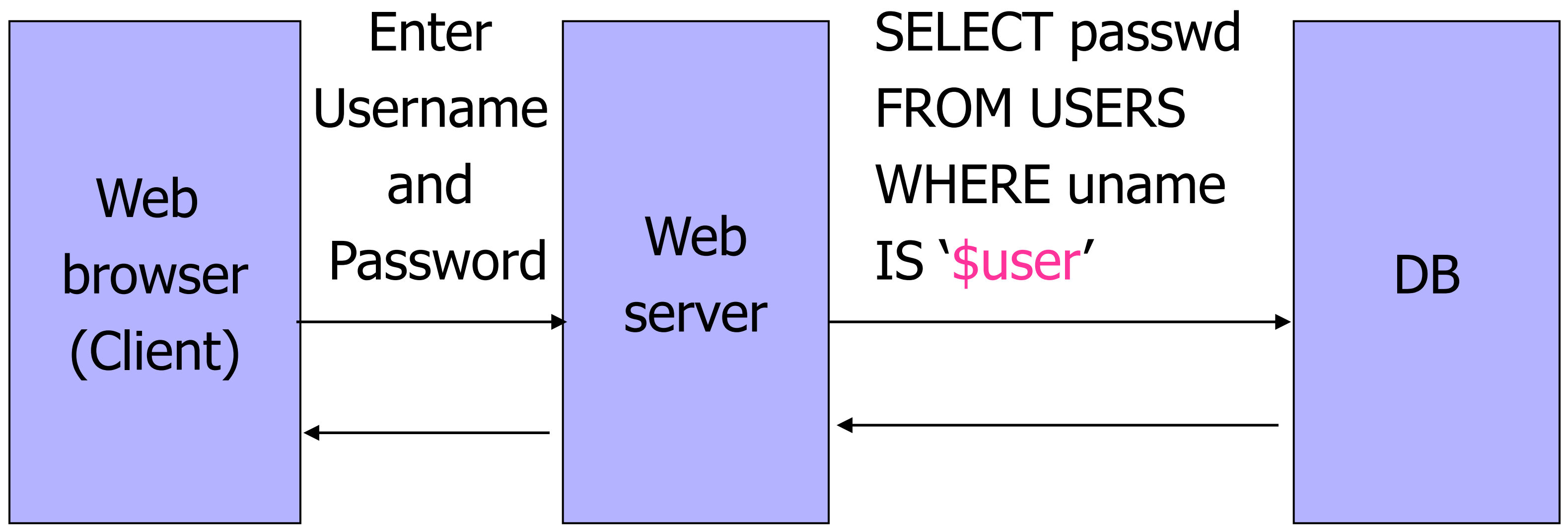
Typical Login Prompt


User Login - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search

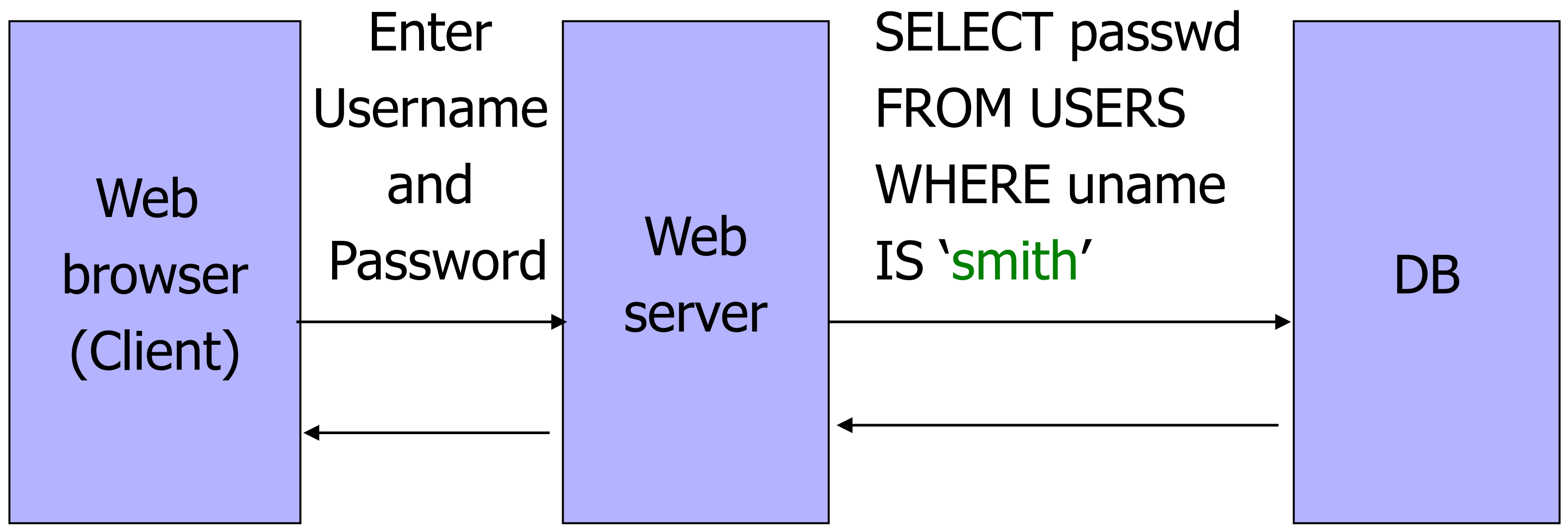
Enter User Name: smith

Enter Password: ●●●●●●●

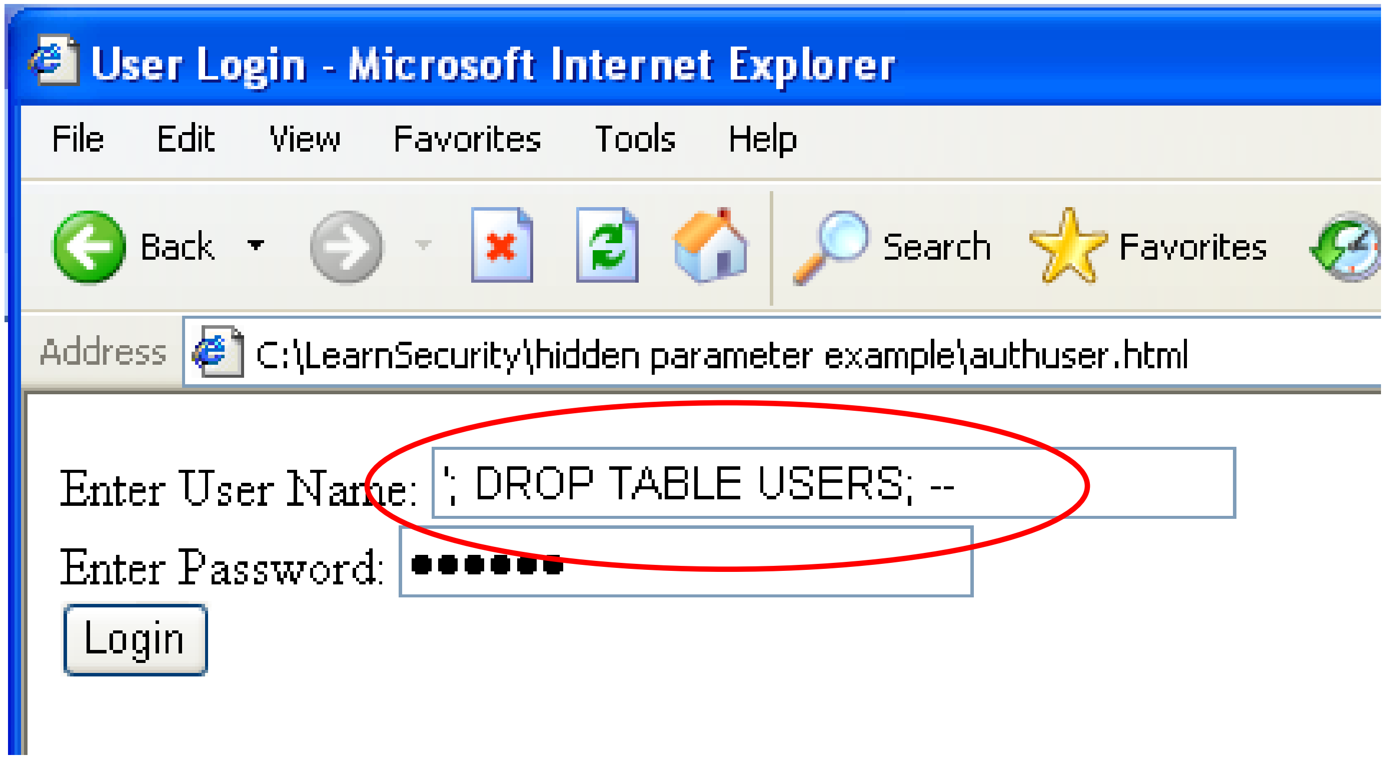
User Input Becomes Part of Query


Web browser (Client)
Enter Username and Password
Web server
SELECT passwd FROM USERS WHERE uname IS '$user'
DB

Normal Login


Web browser (Client)
Enter Username and Password
Web server
SELECT passwd FROM USERS WHERE uname IS ‘smith’
DB

Malicious User Input

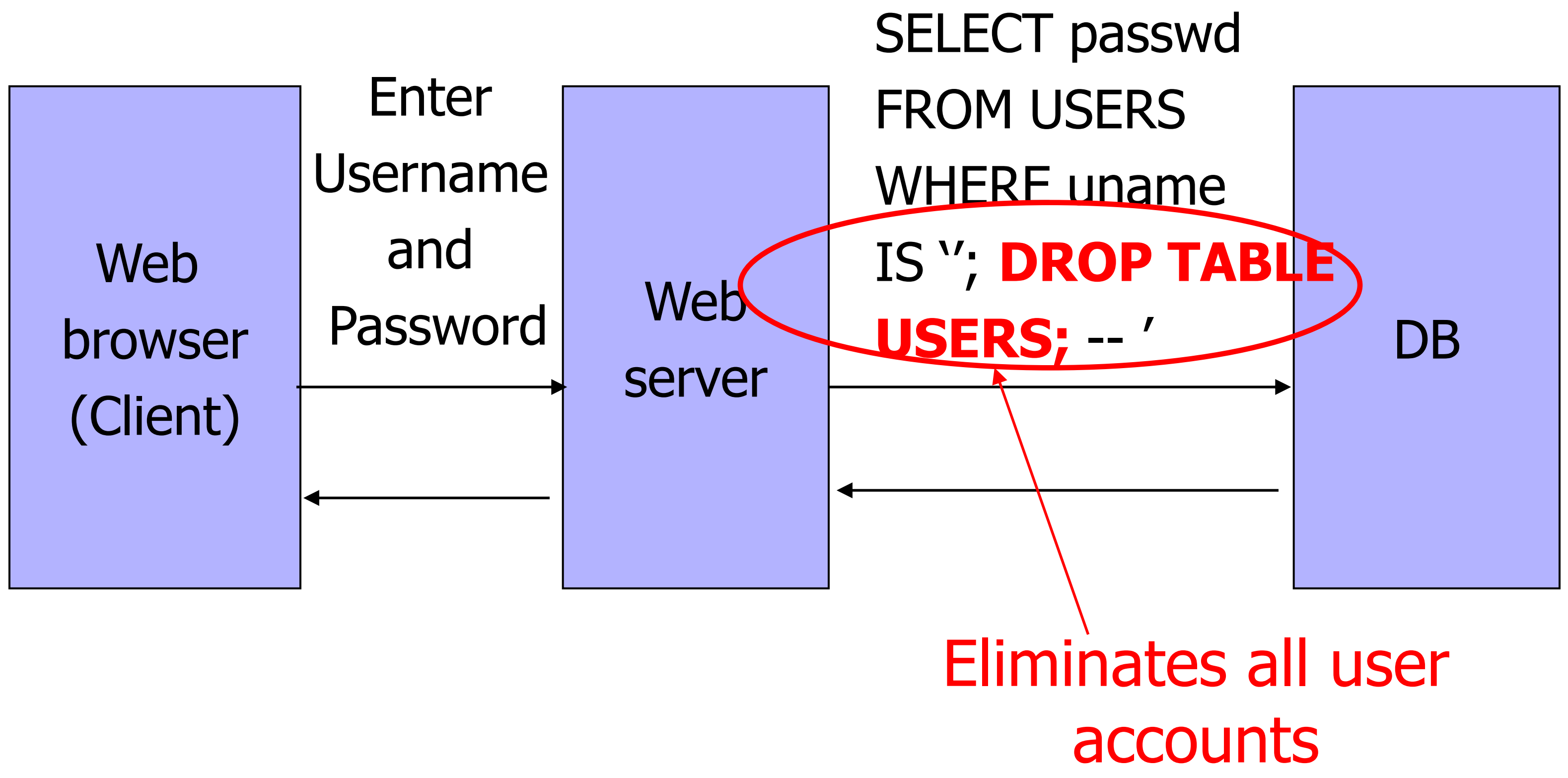


C:\LearnSecurity\hidden parameter example\authuser.html

Enter User Name: '; DROP TABLE USERS; --

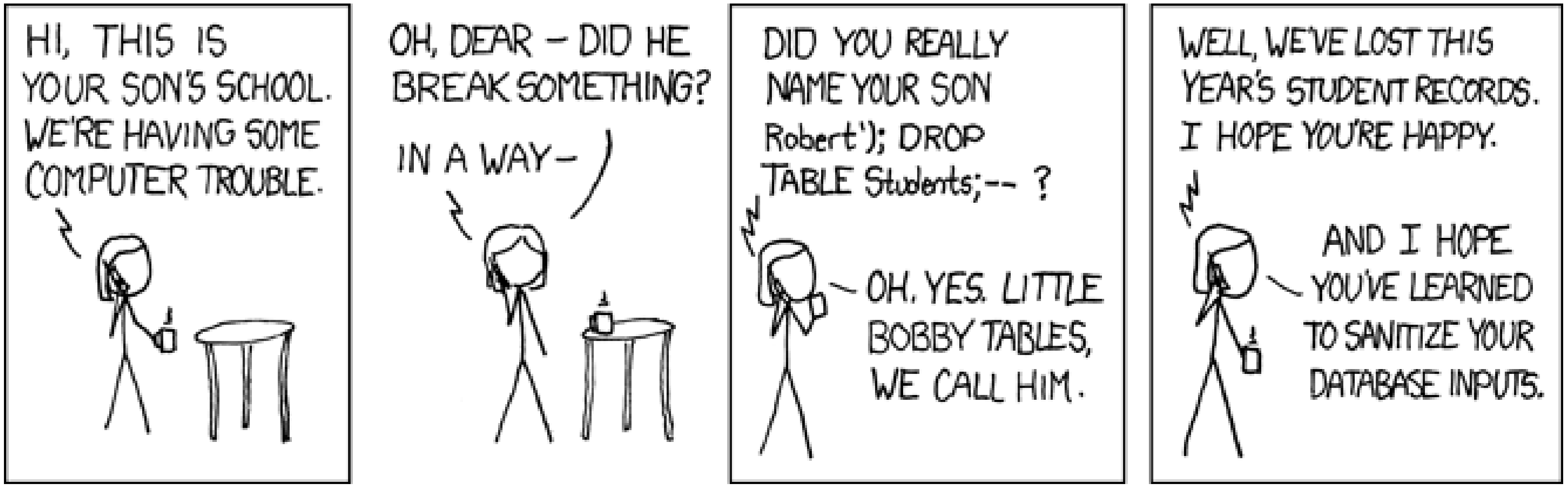
Enter Password: ●●●●●●●

SQL Injection Attack


Web browser (Client)
Enter Username and Password
Web server
SELECT passwd FROM USERS WHERE uname IS ‘’; **DROP TABLE USERS;** -- ’
DB
Eliminates all user accounts

Exploits of a Mom

http://xkcd.com/327/


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Authentication with Back-End DB

```
◆set UserFound=execute(
     "SELECT * FROM UserTable WHERE
     username=' "  &  form("user")  & " ' AND
     password= ' "  &  form("pwd")  & " ' " );
```

User supplies username and password, this SQL query checks if user/password combination is in the database

```
◆If not UserFound.EOF
     Authentication correct
 else Fail
```

Only true if the result of SQL query is not empty, i.e., user/pwd is in the database

Using SQL Injection to Log In

- User gives username ′ OR 1=1 --
- Web server executes query

```
set UserFound=execute(
        SELECT * FROM UserTable WHERE
        username=‵′ OR 1=1 -- … );
```

Always true!

Everything after -- is ignored!

- Now <u>all</u> records match the query, so the result is not empty ⇒ correct “authentication”!

Another SQL Injection Example

[From "The Art of Intrusion"]

- To authenticate logins, server runs this SQL command against the user database:

  SELECT * WHERE user='name' AND pwd='passwd'

- User enters ' OR WHERE pwd LIKE '% as both name and passwd

Wildcard matches any password

- Server executes

  SELECT * WHERE user='' OR WHERE pwd LIKE '%'
  AND pwd='' OR WHERE pwd LIKE '%'

- Logs in with the credentials of the first person in the database (typically, administrator!)

Pull Data From Other Databases

- User gives username

  ```
  ' AND 1=0
  UNION SELECT cardholder, number,
  exp_month, exp_year FROM creditcards
  ```
- Results of two queries are combined
- Empty table from the first query is displayed together with the entire contents of the credit card database

More SQL Injection Attacks

- Create new users

  ’; INSERT INTO USERS (‘uname’,‘passwd’,‘salt’) VALUES (‘hacker’,‘38a74f’, 3234);

- Reset password

  ’; UPDATE USERS SET email=hcker@root.org WHERE email=victim@yahoo.com

Second-Order SQL Injection

- Second-order SQL injection: data stored in database is later used to conduct SQL injection
- For example, user manages to set uname to admin’ --
  - This vulnerability could exist if string escaping is applied inconsistently (e.g., strings not escaped)
  - UPDATE USERS SET passwd=‘cracked’ WHERE uname=‘admin’ --’ why does this work?
- Solution: treat all parameters as dangerous

CardSystems Attack (June 2005)

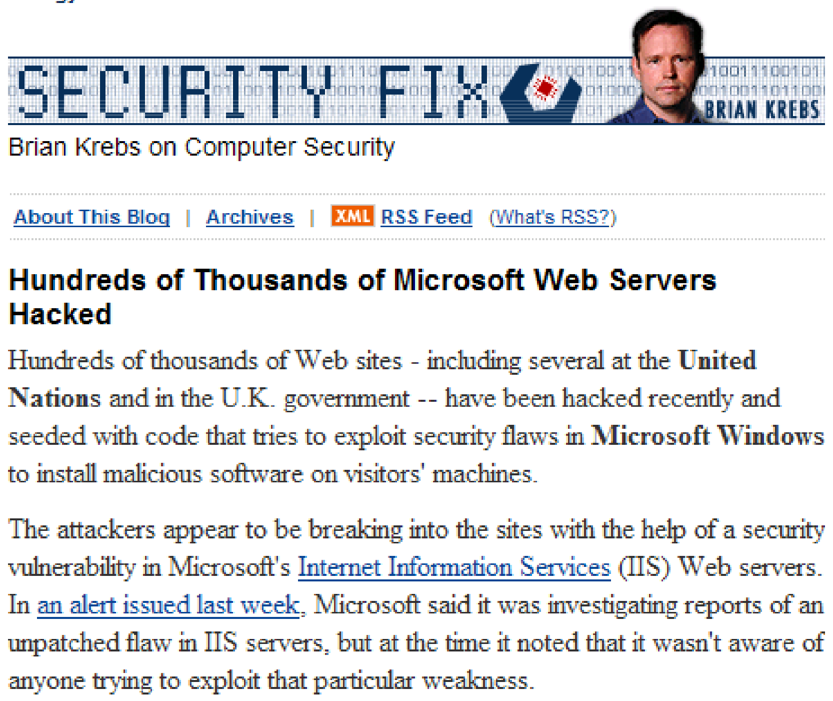
- CardSystems was a major credit card processing company
- Put out of business by a SQL injection attack
  - Credit card numbers stored unencrypted
  - Data on 263,000 accounts stolen
  - 43 million identities exposed

SQL Injection in the Real World

http://www.ireport.com/docs/DOC-11831

- Oklahoma Department of Corrections divulges thousands of social security numbers (2008)
  - Sexual and Violent Offender Registry for Oklahoma
  - Data repository lists both offenders and employees
- "Anyone with a web browser and the knowledge from Chapter One of SQL for Dummies could have easily accessed – and possibly, changed – any data within the DOC's databases"

Attack on Microsoft IIS (April 2008)



SECURITY FIX BRIAN KREBS

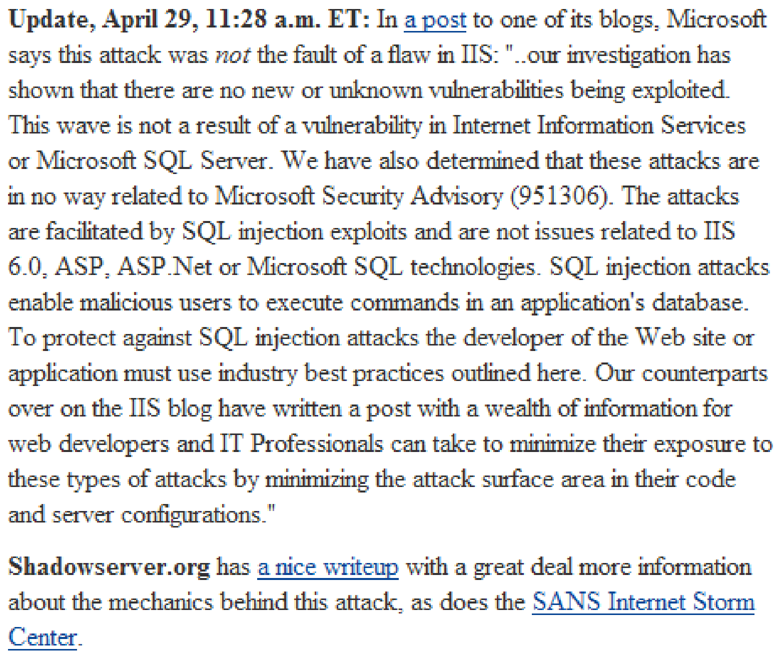
Brian Krebs on Computer Security

About This Blog | Archives | XML RSS Feed (What's RSS?)

Hundreds of Thousands of Microsoft Web Servers Hacked

Hundreds of thousands of Web sites - including several at the **United Nations** and in the U.K. government -- have been hacked recently and seeded with code that tries to exploit security flaws in **Microsoft Windows** to install malicious software on visitors' machines.

The attackers appear to be breaking into the sites with the help of a security vulnerability in Microsoft's Internet Information Services (IIS) Web servers. In an alert issued last week, Microsoft said it was investigating reports of an unpatched flaw in IIS servers, but at the time it noted that it wasn't aware of anyone trying to exploit that particular weakness.



Update, April 29, 11:28 a.m. ET: In a post to one of its blogs, Microsoft says this attack was *not* the fault of a flaw in IIS: "..our investigation has shown that there are no new or unknown vulnerabilities being exploited. This wave is not a result of a vulnerability in Internet Information Services or Microsoft SQL Server. We have also determined that these attacks are in no way related to Microsoft Security Advisory (951306). The attacks are facilitated by SQL injection exploits and are not issues related to IIS 6.0, ASP, ASP.Net or Microsoft SQL technologies. SQL injection attacks enable malicious users to execute commands in an application's database. To protect against SQL injection attacks the developer of the Web site or application must use industry best practices outlined here. Our counterparts over on the IIS blog have written a post with a wealth of information for web developers and IT Professionals can take to minimize their exposure to these types of attacks by minimizing the attack surface area in their code and server configurations."

Shadowserver.org has a nice writeup with a great deal more information about the mechanics behind this attack, as does the SANS Internet Storm Center.

Main Steps in April 2008 Attack

- Use Google to find sites using a particular ASP style vulnerable to SQL injection
- Use SQL injection to modify the pages to include a link to a Chinese site nihaorr1.com
  - Do not visit that site – it serves JavaScript that exploits vulnerabilities in IE, RealPlayer, QQ Instant Messenger
- Attack used automatic tool; can be configured to inject whatever you like into vulnerable sites
- There is some evidence that hackers may get paid for each victim's visit to nihaorr1.com

Part of the SQL Attack String

```
DECLARE @T varchar(255),@C varchar(255)
DECLARE Table_Cursor  CURSOR
FOR select a.name,b.name from sysobjects a,syscolumns b where
a.id=b.id and a.xtype='u' and
(b.xtype=99 or b.xtype=35 or b.xtype=231 or b.xtype=167)
OPEN Table_Cursor
FETCH NEXT FROM  Table_Cursor INTO @T,@C
WHILE(@@FETCH_STATUS=0) BEGIN
 exec('update ['+@T+'] set
['+@C+']=rtrim(convert(varchar,['+@C+']))+'` ''')
FETCH NEXT FROM  Table_Cursor INTO @T,@C
END CLOSE Table_Cursor
DEALLOCATE Table_Cursor;
DECLARE%20@S%20NVARCHAR(4000);SET%20@S=CAST(
%20AS%20NVARCHAR(4000));EXEC(@S);--
```

Preventing SQL Injection

- Input validation
  - Filter
    - Apostrophes, semicolons, percent symbols, hyphens, underscores, …
    - Any character that has special meanings
  - Check the data type (e.g., make sure it's an integer)
- Whitelisting
  - Blacklisting "bad" characters doesn't work
    - Forget to filter out some characters
    - Could prevent valid input (e.g., last name O'Brien)
  - Allow only well-defined set of safe values
    - Set implicitly defined through regular expressions

Escaping Quotes

- For valid string inputs use escape characters to prevent the quote becoming part of the query
  - Example: escape(o'connor) = o''connor
  - Convert ' into \'
- Only works for string inputs
- Different databases have different rules for escaping

Prepared Statements

- Metacharacters such as ′ in queries provide distinction between data and control
- In most injection attacks data are interpreted as control – this changes the semantics of a query or a command
- Bind variables: ? placeholders guaranteed to be data (not control)
- Prepared statements allow creation of static queries with bind variables → preserves the structure of intended query

Prepared Statement: Example

[redacted] http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html

```
PreparedStatement ps =
    db.prepareStatement("SELECT pizza, toppings, quantity, order_day "
                   + "FROM orders WHERE userid=? AND order_month=?");
ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParamenter("month")));
ResultSet res = ps.executeQuery();
```

Bind variable
(data placeholder)

- Query parsed without parameters
- Bind variables are typed (int, string, …)

Parameterized SQL in ASP.NET

- Builds SQL queries by properly escaping args
  - Replaces ′ with \′

```
SqlCommand cmd = new SqlCommand(
      “SELECT * FROM UserTable WHERE
      username = @User AND
      password = @Pwd”, dbConnection);
cmd.Parameters.Add(“@User”, Request[“user”] );
cmd.Parameters.Add(“@Pwd”, Request[“pwd”] );
cmd.ExecuteReader();
```

G. Wassermann and Z. Su

Sound and Precise Analysis of Web Applications for Injection Vulnerabilities

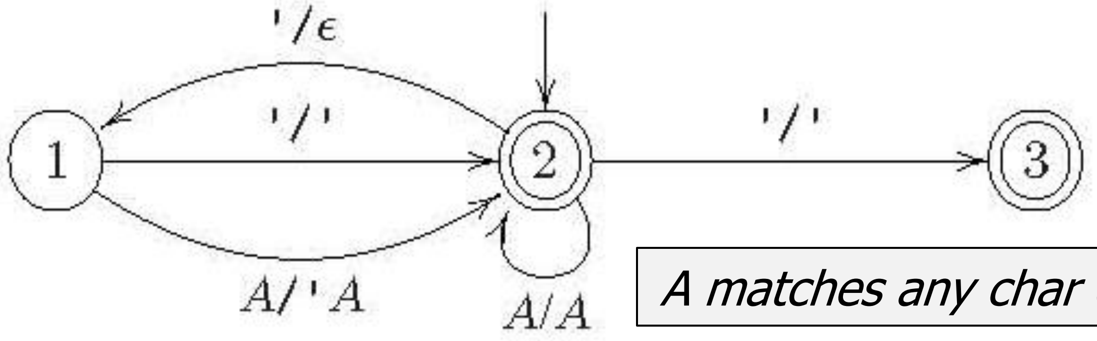
(PLDI 2007)

Wassermann-Su Approach

- Focuses on SQL injection vulnerabilities
- Soundness
  - Tool is guaranteed to find all vulnerabilities
- Precision
  - Models semantics of sanitization functions
  - Models the structure of the SQL query into which untrusted user inputs are fed

“Essence” of SQL Injection

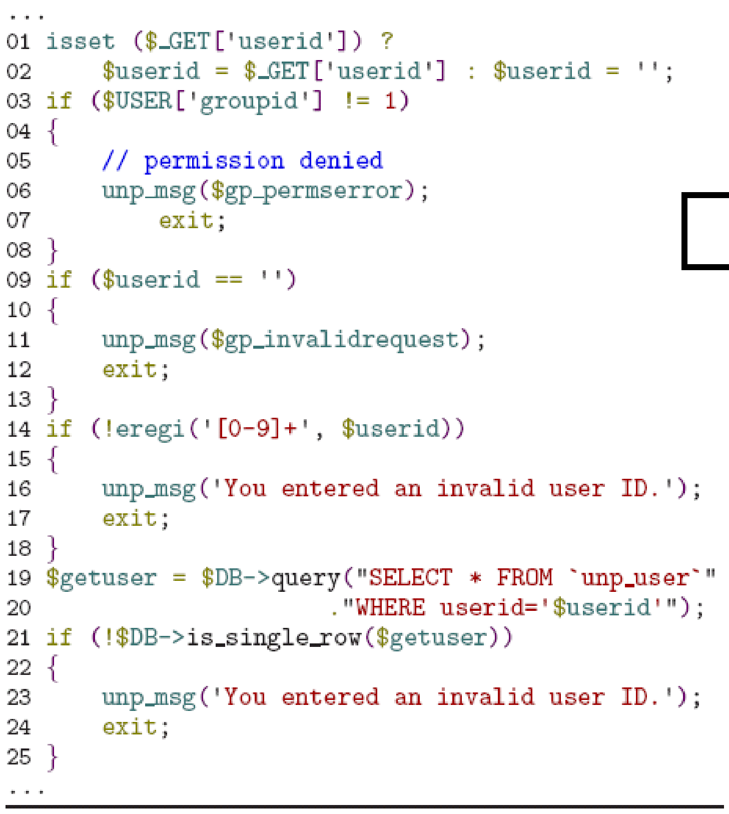
- Web app provides a template for the SQL query
- Attack = any query in which user input changes the intended structure of SQL query
- Model strings as context-free grammars (CFG)
  - Track non-terminals representing tainted input
- Model string operations as language tranducers
  - Example: str_replace(“ ’ ’ “, “ ’ “, $input)



A matches any char except " ’ "

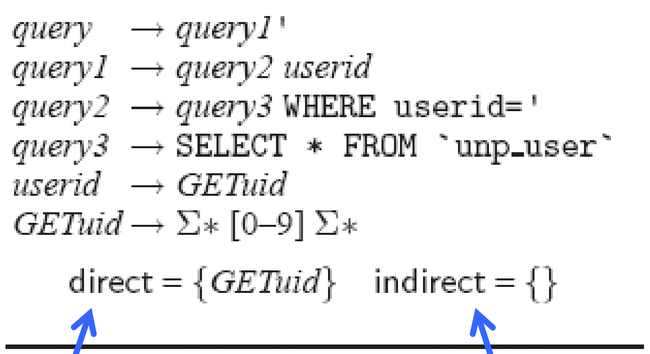
Phase One: Grammar Production

- Generate annotated CFG representing set of all query strings that program can generate



```
...
isset ($_GET['userid']) ?
    $userid = $_GET['userid'] : $userid = '';
if ($USER['groupid'] != 1)
{
    // permission denied
    unp_msg($gp_permserror);
        exit;
}
if ($userid == '')
{
    unp_msg($gp_invalidrequest);
    exit;
}
if (!eregi('[0-9]+', $userid))
{
    unp_msg('You entered an invalid user ID.');
    exit;
}
$getuser = $DB->query("SELECT * FROM `unp_user`"
                    ."WHERE userid='$userid'");
if (!$DB->is_single_row($getuser))
{
    unp_msg('You entered an invalid user ID.');
    exit;
}
...
```

Example code with an SQLCIV.



$$
\begin{array}{lll}
\mathit{query} & \rightarrow & \mathit{query1}\ \texttt{'} \\
\mathit{query1} & \rightarrow & \mathit{query2}\ \mathit{userid} \\
\mathit{query2} & \rightarrow & \mathit{query3}\ \texttt{WHERE userid='} \\
\mathit{query3} & \rightarrow & \texttt{SELECT * FROM `unp\_user`} \\
\mathit{userid} & \rightarrow & \mathit{GETuid} \\
\mathit{GETuid} & \rightarrow & \Sigma* \ [0\text{–}9] \ \Sigma*
\end{array}
$$

direct = $\{\mathit{GETuid}\}$ indirect = $\{\}$

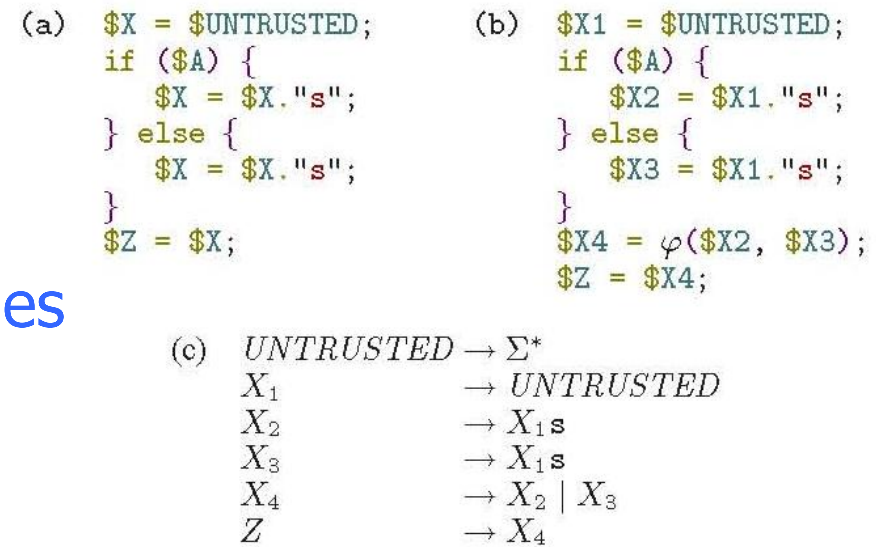
Grammar productions of possible query strings

Direct:
data directly from users (e.g., GET parameters)

Indirect:
second-order tainted data (means what?)

String Analysis + Taint Analysis

- Convert program into static single assignment form, then into CFG
  - Reflects data dependencies
- Model PHP filters as string transducers
  - Some filters are more complex: preg_replace("/a([0-9]*)b/", "x\\1\\1y", "a01ba3b") produces "x0101yx33y"
- Propagate taint annotations



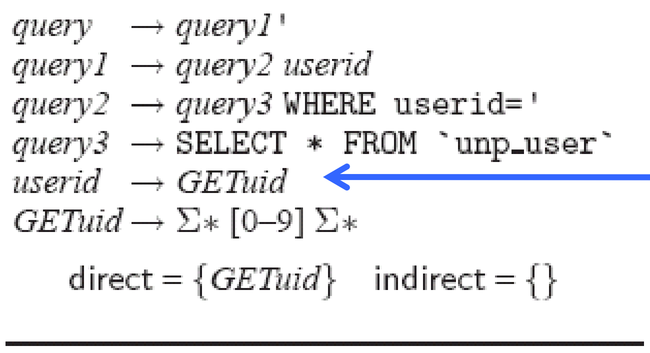
(a)
```
$X = $UNTRUSTED;
if ($A) {
    $X = $X."s";
} else {
    $X = $X."s";
}
$Z = $X;
```

(b)
```
$X1 = $UNTRUSTED;
if ($A) {
    $X2 = $X1."s";
} else {
    $X3 = $X1."s";
}
$X4 = φ($X2, $X3);
$Z = $X4;
```

(c)
$$\begin{aligned}
UNTRUSTED &\to \Sigma^* \\
X_1 &\to UNTRUSTED \\
X_2 &\to X_1\mathtt{s} \\
X_3 &\to X_1\mathtt{s} \\
X_4 &\to X_2 \mid X_3 \\
Z &\to X_4
\end{aligned}$$

Phase Two: Checking Safety

- Check whether the language represented by CFG contains unsafe queries
  - Is it syntactically contained in the language defined by the application’s query template?



```
query   → query1 '
query1  → query2 userid
query2  → query3 WHERE userid='
query3  → SELECT * FROM `unp_user`
userid  → GETuid
GETuid → Σ* [0–9] Σ*

    direct = {GETuid}   indirect = {}
```

Grammar productions of possible query strings

This non-terminal represents tainted input

For all sentences of the form σ_1 GETUID σ_2 derivable from query, GETUID is between quotes in the position of an SQL string literal (means what?)

Safety check:
Does the language rooted in GETUID contain unescaped quotes?

Tainted Substrings as SQL Literals

- Tainted substrings that cannot be syntactically confined in any SQL query
  - Any string with an odd # of unescaped quotes (why?)
- Nonterminals that occur only in the syntactic position of SQL string literals
  - Can an unconfined string be derived from it?
- Nonterminals that derive numeric literals only
- Remaining nonterminals in literal position can produce a non-numeric string outside quotes
  - Probably an SQL injection vulnerability
  - Test if it can derive DROP WHERE, --, etc.

Taints in Non-Literal Positions

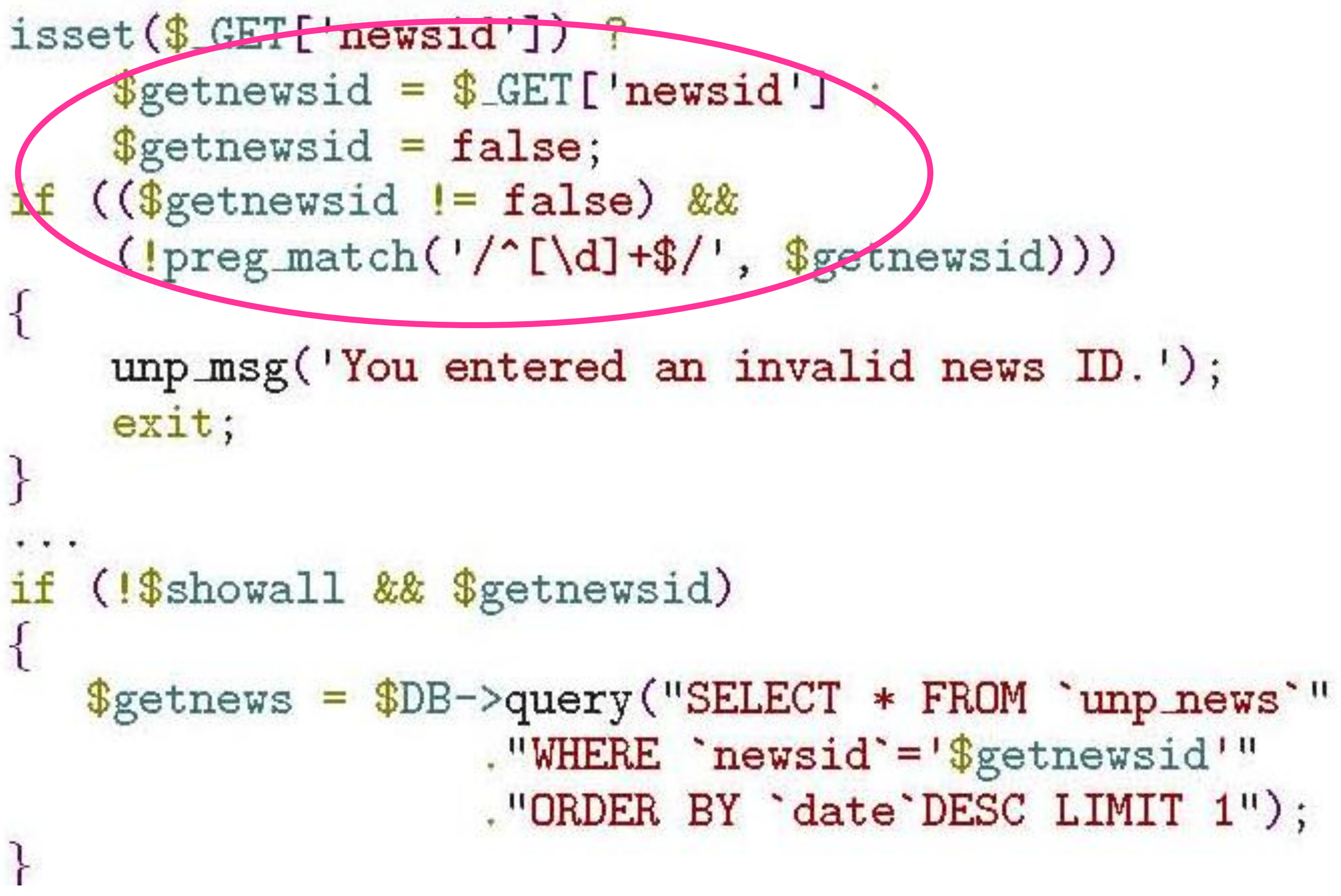
- Remaining tainted nonterminals appear as non-literals in SQL query generated by the application
  - This is rare (why?)
- All derivable strings should be proper SQL statements
  - Context-free language inclusion is undecidable
  - Approximate by checking whether each derivable string is also derivable from a nonterminal in the SQL grammar
    - Variation on a standard algorithm

Evaluation

- Testing on five real-world PHP applications
- Discovered previously unknown vulnerabilities, including non-trivial ones
  - Vulnerability in e107 content management system:

    a field is read from a user-modifiable cookie, used in a query in a different file
- 21% false positive rate
  - What are the sources of false positives?

Example of a False Positive



```
isset($_GET['newsid']) ?
    $getnewsid = $_GET['newsid'] :
    $getnewsid = false;
if (($getnewsid != false) &&
    (!preg_match('/^[\d]+$/', $getnewsid)))
{
    unp_msg('You entered an invalid news ID.');
    exit;
}
...
if (!$showall && $getnewsid)
{
   $getnews = $DB->query("SELECT * FROM `unp_news`"
                       ."WHERE `newsid`='$getnewsid'"
                       ."ORDER BY `date`DESC LIMIT 1");
}
```