Introduction to String Analysis

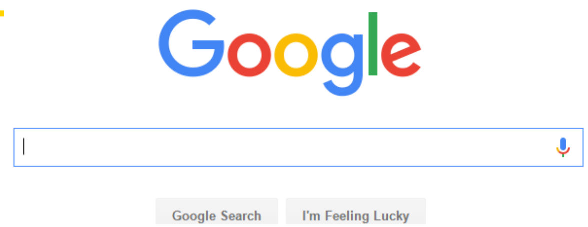
292C

Tevfik Bultan

Modern Software Applications

Common Usages of Strings

- Input validation and sanitization

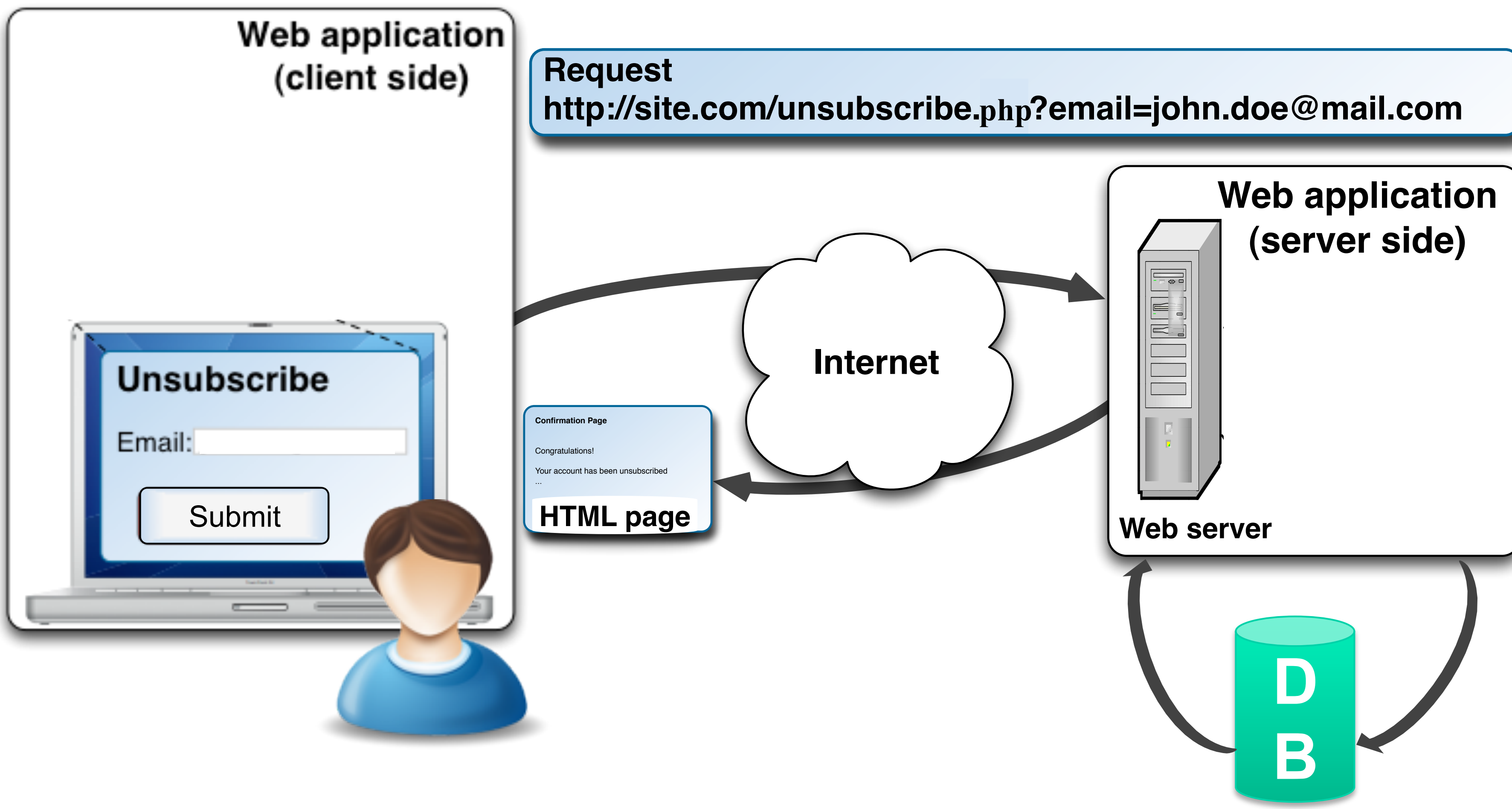


- Database query generation

- Formatted data generation
- Dynamic code generation
- Dynamic class loading and method invocation

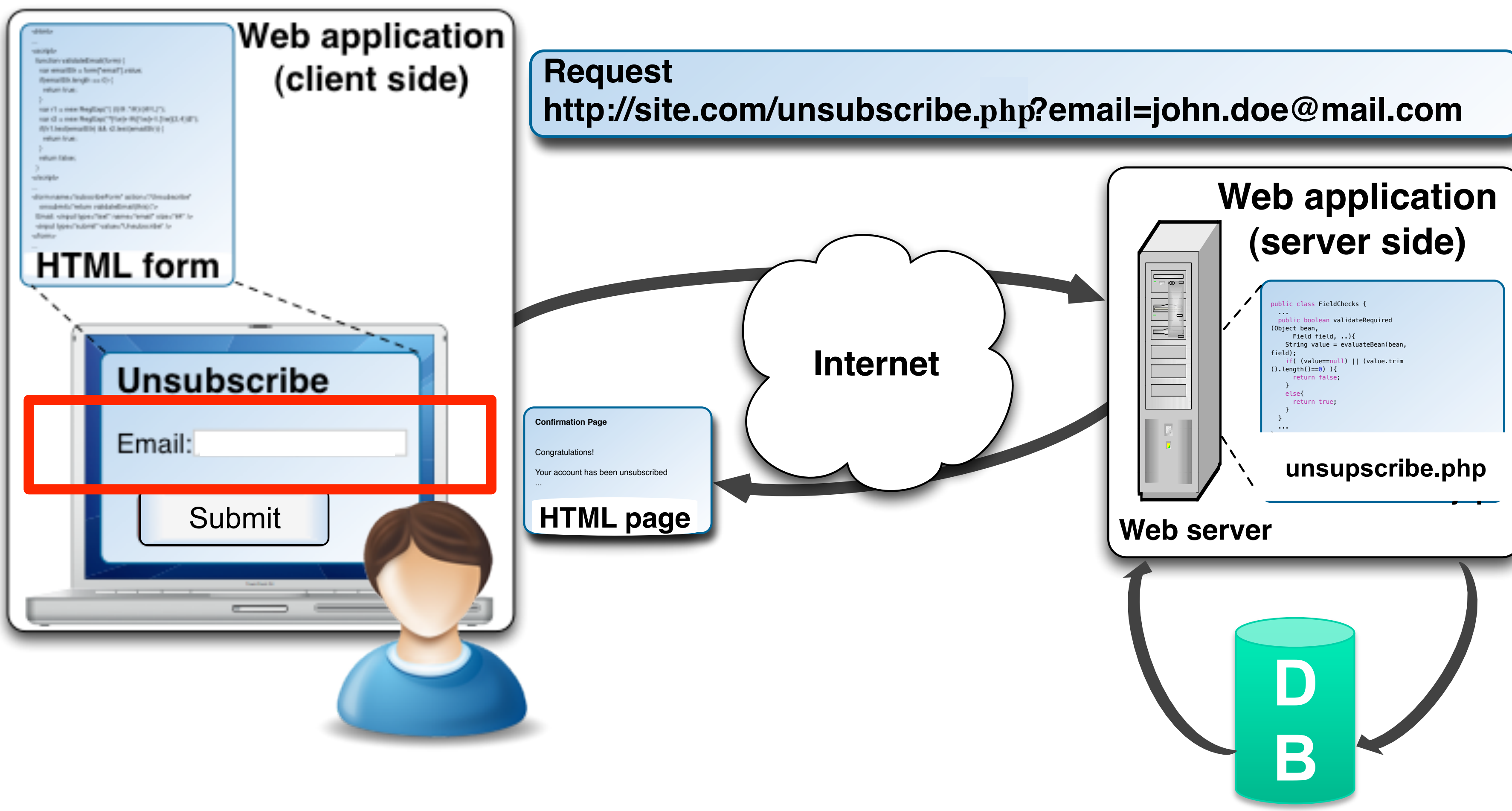
- Access control in the cloud

Anatomy of a Web Application

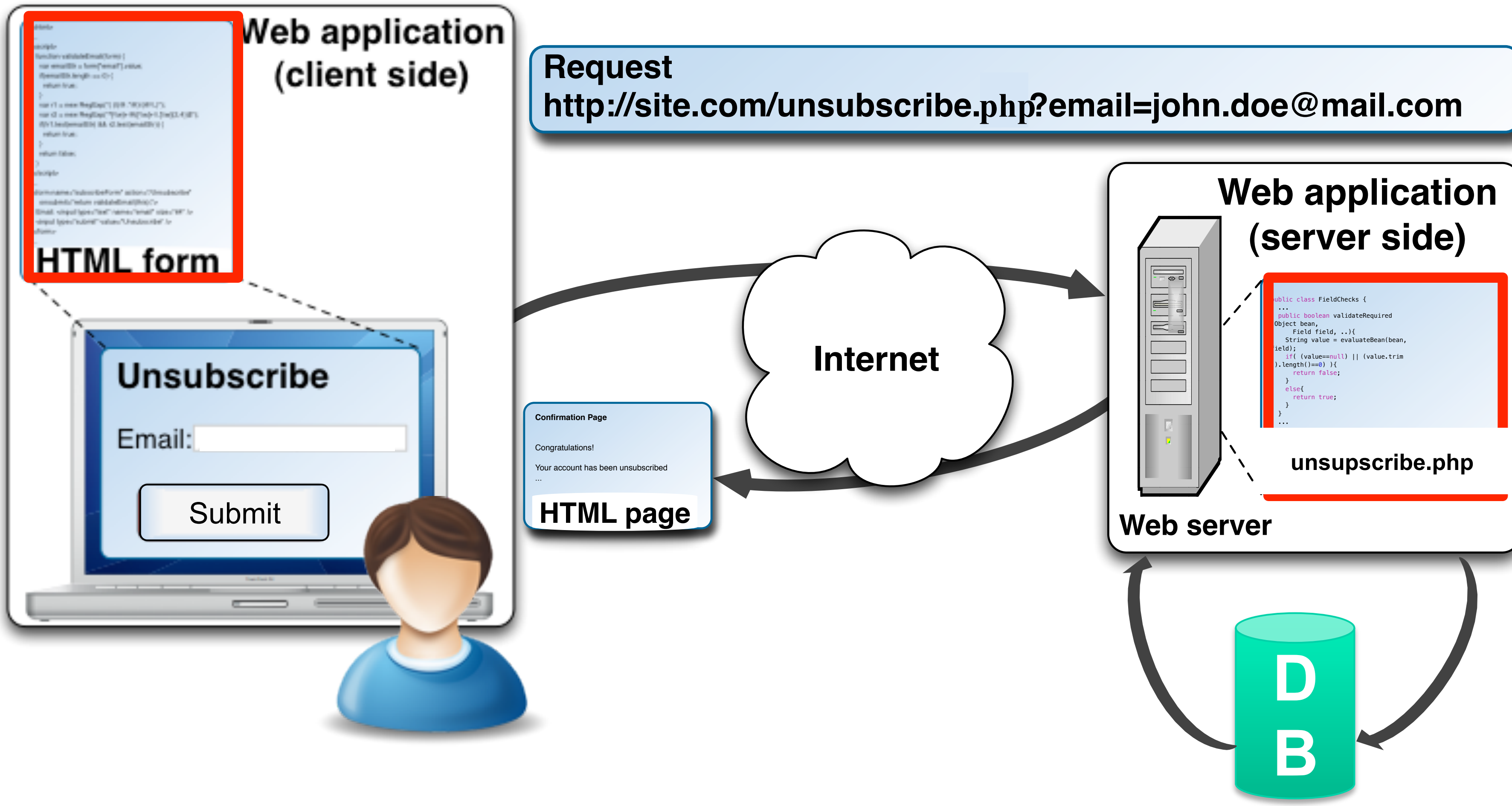


Web Application Inputs are Strings

Web Application Inputs are Strings



Input Needs to be Validated and/or Sanitized

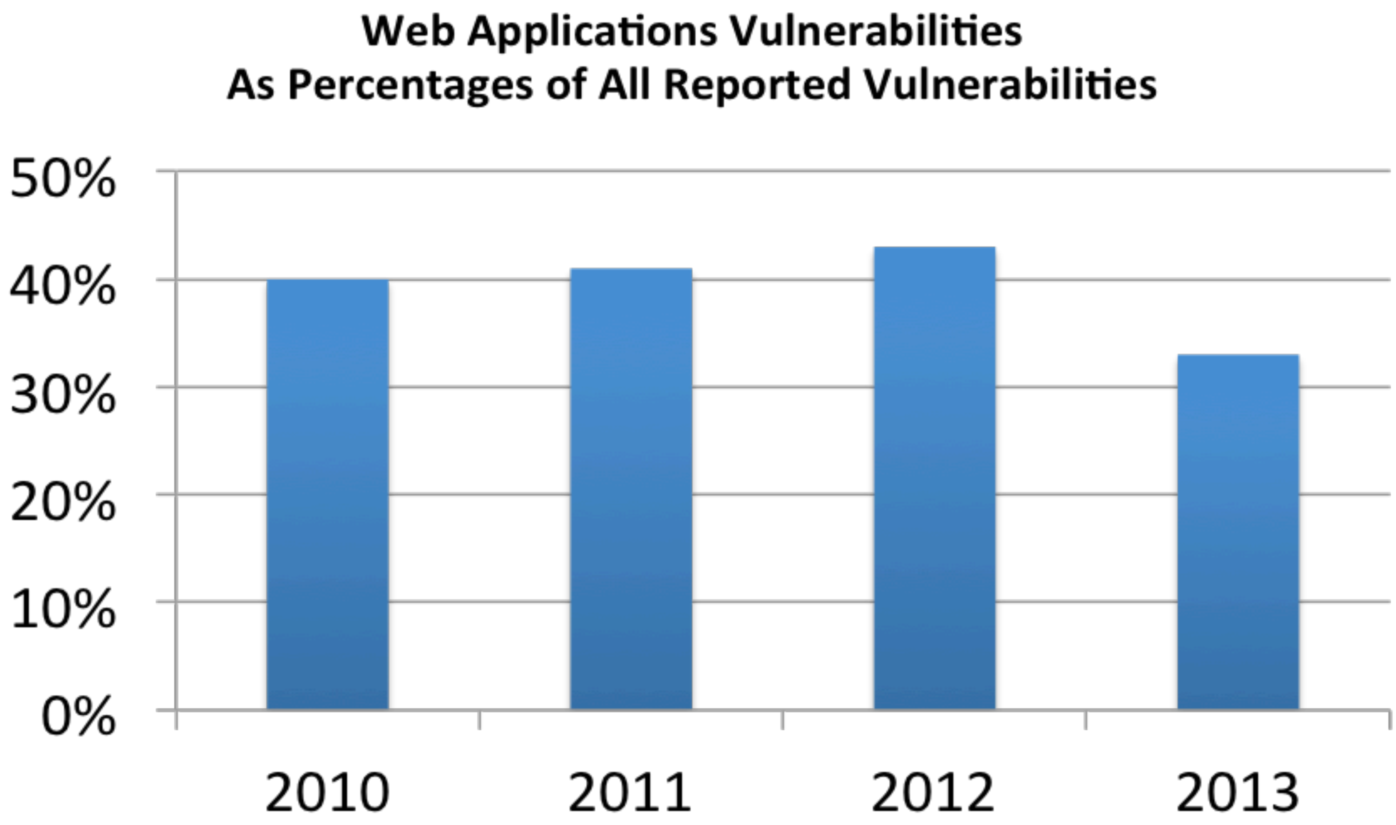


Vulnerabilities in Web Applications

- There are many well-known security vulnerabilities that exist in many web applications. Here are some examples:
  - **SQL injection:** where a malicious user executes SQL commands on the back-end database by providing specially formatted input
  - **Cross site scripting (XSS):** causes the attacker to execute a malicious script at a user's browser
  - **Malicious file execution:** where a malicious user causes the server to execute malicious code

- These vulnerabilities are typically due to
  - errors in user input validation and sanitization or
  - lack of user input validation and sanitization

Web Applications are Full of Bugs



Source: IBM X-Force report

Top Web Application Vulnerabilities

2007

1. Injection Flaws
2. XSS
3. Malicious File Execution

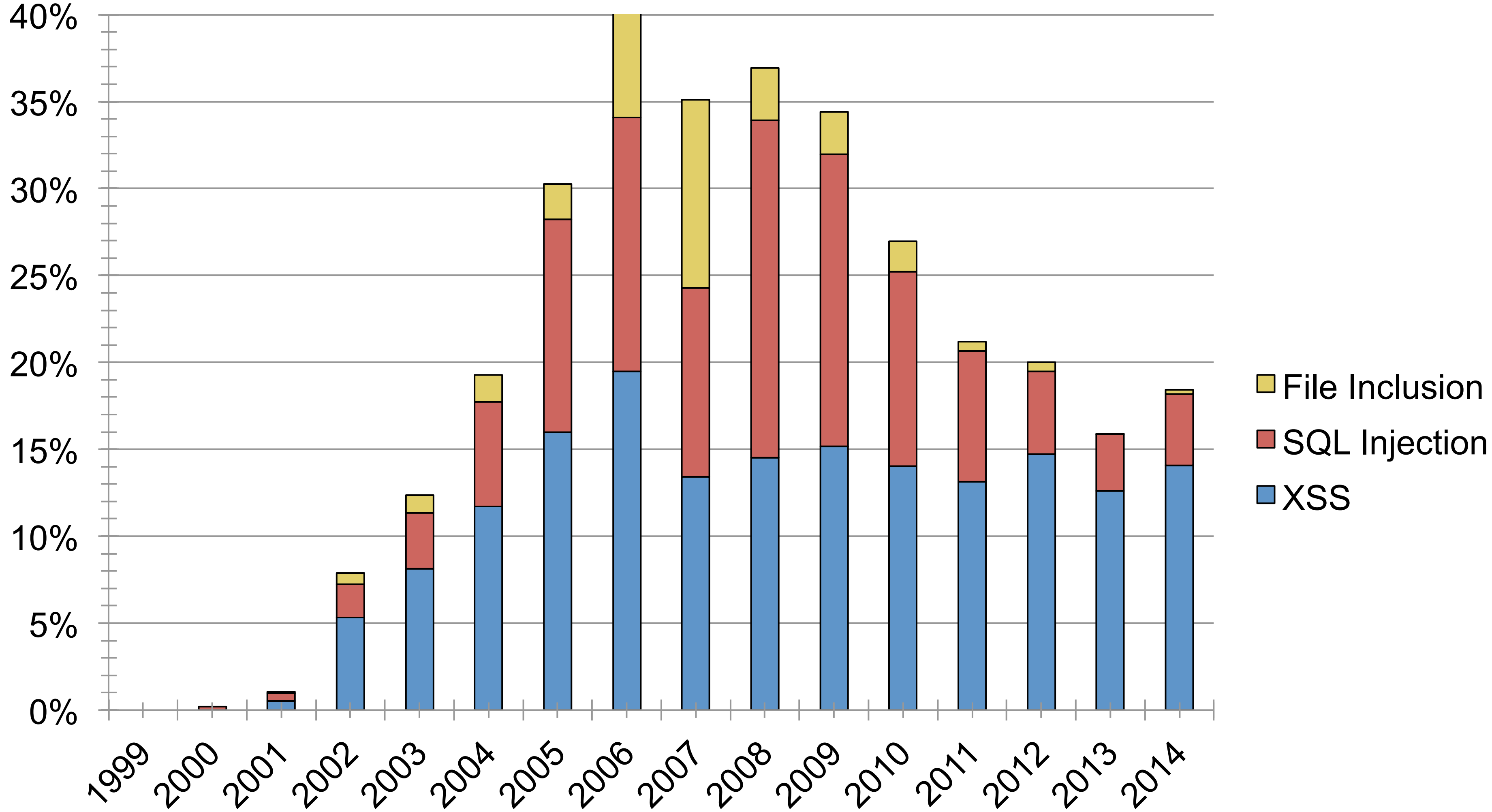
2010

1. Injection Flaws
2. XSS
3. Broken Auth. Session Management

2013

1. Injection Flaws
2. Broken Auth. Session Management
3. XSS

As Percentage of All Vulnerabilities



- SQL Injection, XSS, File Inclusion as percentage of all computer security vulnerabilities (extracted from the CVE repository)

Why Is Input Validation Error-prone?

- ***Extensive string manipulation***:
  - Web applications use extensive string manipulation
    - To construct html pages, to construct database queries in SQL, etc.
  - The user input comes in string form and must be validated and sanitized before it can be used
    - This requires the use of complex string manipulation functions such as string-replace
  - String manipulation is error prone

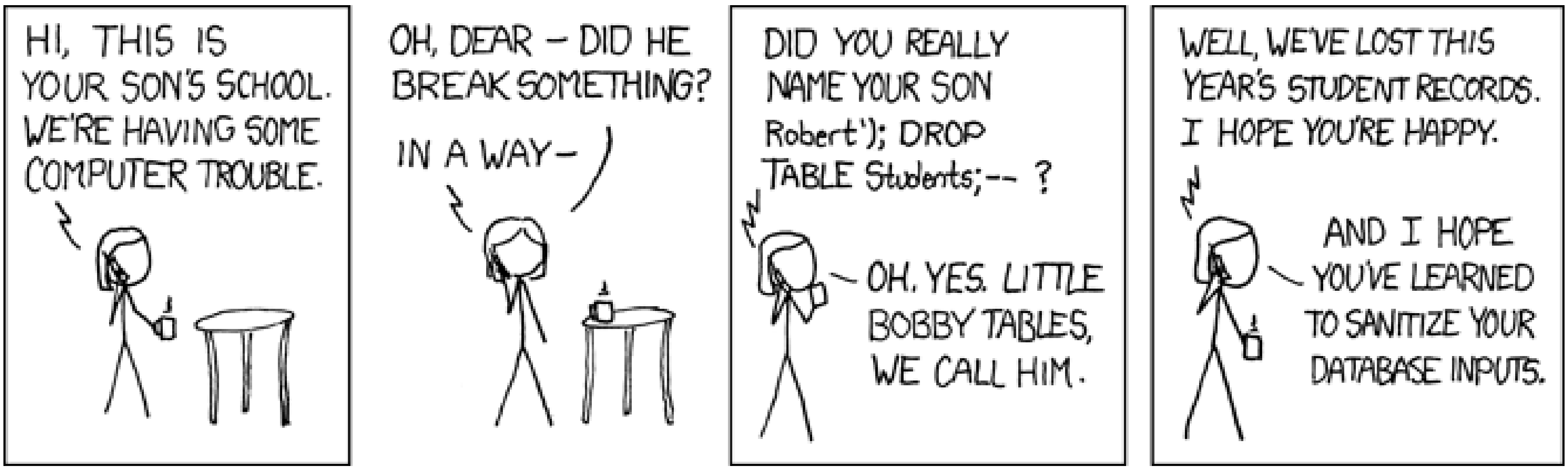
String Related Vulnerabilities

- String related web application vulnerabilities occur when:
  - a ***sensitive function*** is passed a ***malicious string input from the user***
  - This input contains an ***attack***
  - It is not ***properly sanitized*** before it reaches the sensitive function

- Using **string analysis** we can discover these vulnerabilities automatically

Computer Trouble at School

Exploits of a Mom.



Source: XKCD.com

SQL Injection

- A PHP example
- Access students' data by $name (from a user input).

```
<?php
 $name = $GET["name"];
 $user data = $db->query("SELECT * FROM students
                              WHERE name = '$name'");
?>
```

SQL Injection

- A PHP Example:
- Access students' data by $name (from a user input).

```
<?php
$name = $GET["name"];
$user data = $db->query("SELECT * FROM students
WHERE name = 'Robert '); DROP TABLE students; - -");
?>
```

What is a String?

- Given alphabet Σ, a string is a finite sequence of alphabet symbols

  $\mathbf{<c_1, c_2, \ldots, c_n>}$ for all i, c_i is a character from Σ
- Σ = English = {a,…,z, A,…Z}

Σ = {a}

Σ = {a, b},

Σ = ASCII = {NULL, …, !, “, …, 0, …, 9, …, a, …, z, …}

Σ = Unicode

Σ = ASCII	Σ = English	Σ = {a}	Σ = {a,b}
“Foo”	“Hello”	“a”	“a”
“Ldkh#$klj54”	“Welcome”	“aa”	“aba”
“123”	“good”	“aaa”	“bbb”
		“aaaa”	“ababaa”
		“aaaaa”	“aaa”

String Manipulation Operations

- Concatenation
  - "1" + "2" → "12"
  - "Foo" + "bAaR" → "FoobAaR"

- Replacement
  - replace(s, "a", "A") bAaR → bAAR
  - replace (s, "2","") 234 → 34
  - toUpperCase(s) abC → ABC

String Filtering Operations

- Branch conditions

  length(s) < 4 ?

  - “Foo”
  - “bAaR”

  match(s, /[0-9]+$/) ?

  - “234”
  - “a3v%6”

  substring(s, 2, 4) == “aR” ?

  - ”bAaR”
  - “Foo”

A Simple Example

- Another PHP Example:

<script ...

```
1:<?php
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
4: echo ”<td>” . $l_otherinfo . ”: ” . $www . ”</td>”;
5:?>
```

- The **`echo`** statement in line 4 is a sensitive function
- It contains a Cross Site Scripting (**XSS**) vulnerability

Is It Vulnerable?

- A simple ***taint analysis*** can report this segment vulnerable using taint propagation

tainted

```
<?php
 $www = $_GET[”www”];
 $l_otherinfo = ”URL”;
 echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
?>
```

- **`echo`** is tainted → script is **vulnerable**

How to Fix it?

- To fix the vulnerability we added a sanitization routine at line ***s***
- Taint analysis will assume that $www is ***untainted*** and report that the segment is ***NOT*** vulnerable

```
1:<?php          tainted
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
     untainted
s: $www = ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
4: echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
5:?>
```

Is It Really Sanitized?

```
1:<?php          <script …>
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
   <script …>
s: $www = ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
4: echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
5:?>
```

Sanitization Routines can be Erroneous

- The sanitization statement is not correct!

```
ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
```

  - Removes all characters that are not in { A-Za-z0-9 .-@:/ }
  - `.-@` denotes **all characters between “`.`” and “`@`”** (including **“<”** and **“>”**)
  - “`.-@`” should be “`.\-@`”

- This example is from a buggy sanitization routine used in MyEasyMarket-4.1 (line 218 in file trans.php)

String Analysis

- String analysis determines all possible values that a string expression can take during any program execution

- Using string analysis we can identify all possible input values of the sensitive functions
  - Then we can check if inputs of sensitive functions can contain attack strings

- How can we characterize attack strings?
  - Use regular expressions to specify the attack patterns

  - An attack pattern for XSS: **`Σ*<scriptΣ*`**

Vulnerabilities Can Be Tricky

- Input <!sc+rip!t ...> does not match the attack pattern
  - but it matches the vulnerability signature and it can cause an attack

```
1:<?php          <!sc+rip!t …>
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
    <script …>
s: $www = ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
4: echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
5:?>
```

String Analysis

- If string analysis determines that the intersection of the attack pattern and possible inputs of the sensitive function is empty
  - then we can conclude that the program is secure

- If the intersection is not empty, then we can again use string analysis to generate a ***vulnerability signature***
    - characterizes all malicious inputs

  - Given **`Σ*<scriptΣ*`** as an attack pattern:
    - The vulnerability signature for $_GET["www"] is

      `Σ*<α*sα*cα*rα*iα*pα*tΣ*`

      where $\alpha \notin$ { A-Za-z0-9 .-@:/ }

String manipulation examples: Input validation & sanitization

- Server side input validation code in Java

```
public class Validator {
  public boolean validateEmail(Object bean, Field f, ..) {
    String val = ValidatorUtils.getValueAsString(bean, f);
    Perl5Util u = new Perl5Util();
    if (!(val == null || val.trim().length == 0)) {
      if ((!u.match("/( )|(@.*@)|(@\\.)/", val))
        && u.match("/^[\\w]+@([\\w]+\\.[\\w]{2,4})$/",
                    val)) {
        return true;
      } else {
        return false;
      }
    }
    return true;
  }
  ...
}
```

String manipulation examples: Input validation and sanitization

- Corresponding client side input validation code in JavaScript

```
<html>
...
<script>
function validateEmail(form) {
  var emailStr = form["email"].value;
  if(emailStr.length == 0) {
    return true;
  }
  var r1 = new RegExp("( )|(@.*@)|(@\\.)");
  var r2 = new RegExp("^[\\w]+@([\\w]+\\.[\\w]{2,4})$");
  if(!r1.test(emailStr) && r2.test(emailStr)) {
    return true;
  }
  return false;
}
</script>
...
<form name="subscribeForm" action="/Unsubscribe"
    onsubmit="return validateEmail(this);">
  Email: <input type="text" name="email" size="64" />
  <input type="submit" value="Unsubscribe" />
</form>
...
</html>
```

String manipulation examples: Input validation and sanitization

- Client side input validation code in JavaScript

```
function isValidEmail(emailField) {
       var email = emailField.value.trim();
       emailField.value = email;
       EMAIL_REGEXP =
        /^[a-z0-9!#$%&'*+/=?^_`{|}~-]+
        (?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@
        (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
        [a-z0-9](?:[a-z0-9-]*[a-z0-9])$/;
   if(!EMAIL_REGEXP.test(email)) {
        alert("Please enter a correct email address.");
        emailField.focus();
        return false;
   }
   return true;
}
```

String manipulation examples: Input validation and sanitization

```
function validate() {
 ...
 switch(type) {
  case "time":
   var highlight = true;
   var default_msg = "Please enter a valid time.";
   time_pattern = /^[1-9]\:[0-5][0-9]\s*(\AM|PM|am|pm?)\s*$/;
   time_pattern2 = /^[1-1][0-2]\:[0-5][0-9]\s*(\AM|PM|am|pm?)\s*$/;
   time_pattern3 = /^[1-1][0-2]\:[0-5][0-9]\:[0-5][0-9]\s*(\AM|PM|
                    am|pm?)\s*$/;
   time_pattern4 = /^[1-9]\:[0-5][0-9]\:[0-5][0-9]\s*(\AM|PM|
                    am|pm?)\s*$/;
   if (field.value != "") {
    if (!time_pattern.test(field.value)
       && !time_pattern2.test(field.value)
       && !time_pattern3.test(field.value)
       && !time_pattern4.test(field.value)) {
        error = true;
       }
    }
   break;
  case "email":
   error = isEmailInvalid(field);
   var highlight = true;
   var default_msg = "Please enter a valid email address.";
   break;
  case "date":
   var highlight = true;
   var default_msg = "Please enter a valid date.";
   date_pattern = /^(\d{1}|\d{2})\/(\d{1}|\d{2})\/(\d{2}|\d{4})\s*$/;
   if (field.value != "")
    if (!date_pattern.test(field.value)||!isDateValid(field.value))
 ...
 if (alert_msg == "" || alert_msg == null) alert_msg = default_msg;
 if (error) {
  any_error = true;
  total_msg = total_msg + alert_msg + "|";
 }
 if (error && highlight) {
  field.setAttribute("class","error");
  field.setAttribute("className","error");    // For IE
 }
 ...
}
```

String manipulation examples: Dynamic class loading

- Dynamic class loading with objective C in iOS applications

```
NSBundle *b = [NSBundle bundleWithPath:@"/System/Library
                /Frameworks/AdSupport.framework"];
if(b){
  NSString *name = [NSString stringWithFormat:@"%s%s%s",
          "AS","Identifier","Manager"];
  Class c = NSClassFromString(name);
  id si = [c valueForKey:@"sharedManager"];
}
```

String manipulation examples: Reflective calls

- A reflective call example for Android applications

```
TelephonyManager telephonyManager = (TelephonyManager)
    getSystemService(Context.TELEPHONY_SERVICE);
String imei = telephonyManager.getDeviceId(); //source
Class c = Class.forName("de.ecspride.ReflectiveClass");
Object o = c.newInstance();
Method m = c.getMethod("setIme" + "i", String.class);
m.invoke(o, imei);
```

String manipulation examples: Access control

- Amazon access control policy example

```
{
  "policies": [

 {
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect":"Allow",
      "Principal":"*",
      "Action":"s3:GetObject",
      "Resource":"arn:aws:s3:::examplebucket/*",
      "Condition": {"StringLike": {"s3:prefix": ["${aws:username}/*"]}}
    }
  ]
},

 {
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect":"Allow",
      "Principal":"*",
      "Action":"s3:GetObject",
      "Resource":"arn:aws:s3:::examplebucket/*",
      "Condition": {"StringLike": {"s3:prefix": ["home/*"]}}
    }
  ]
}
]
}
```

String manipulation examples: Side channels

- String manipulating programs can leak information through side channels (such as execution time, memory usage)

```
public String compress(final String in) {
    // variable declarations
    while (readIndex < in.length()) {
        nextChar = in.charAt(readIndex); readIndex++;
        tempIndex = mSearchBuffer.indexOf(currentMatch + (char) nextChar);
        if (tempIndex != -1) {
            currentMatch += (char) nextChar; matchIndex = tempIndex;
        } else {
            final String codedString = "~" + matchIndex + "~" + currentMatch.length() + "~" + (char) nextChar;
            final String concat = currentMatch + (char) nextChar;
            if (codedString.length() <= concat.length()) {
                mOut += codedString; mSearchBuffer += concat;
                currentMatch = ""; matchIndex = 0;
            } else {
                for (currentMatch = concat, matchIndex = -1; currentMatch.length() > 1
                    && matchIndex == -1; currentMatch = currentMatch.substring(1,
                        currentMatch.length()), matchIndex = mSearchBuffer.indexOf(currentMatch)) {
                mOut += currentMatch.charAt(0); mSearchBuffer += currentMatch.charAt(0);
                }
            }
            if (mSearchBuffer.length() <= mBufferSize) {continue;}
            mSearchBuffer = mSearchBuffer.substring(mBufferSize);
        }
    }
    if (matchIndex != -1) {
        final String codedString = "~" + matchIndex + "~" + currentMatch.length();
        if (codedString.length() <= currentMatch.length()) {
            mOut += "~" + matchIndex + "~" + currentMatch.length();
        } else { mOut += currentMatch;}
    }
    return mOut;
}
```