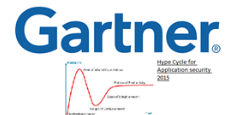
Input Validation

Eoin Keary
CTO BCC Risk Advisory

www.bccriskadvisory.com
www.edgescan.com



Where are we going?

- Don’t ever trust user input
- Where possible, use whitelist validation
- Perform input validation at earliest possible stage
- Layers of defense
- Use in-built validator routines

Data Validation

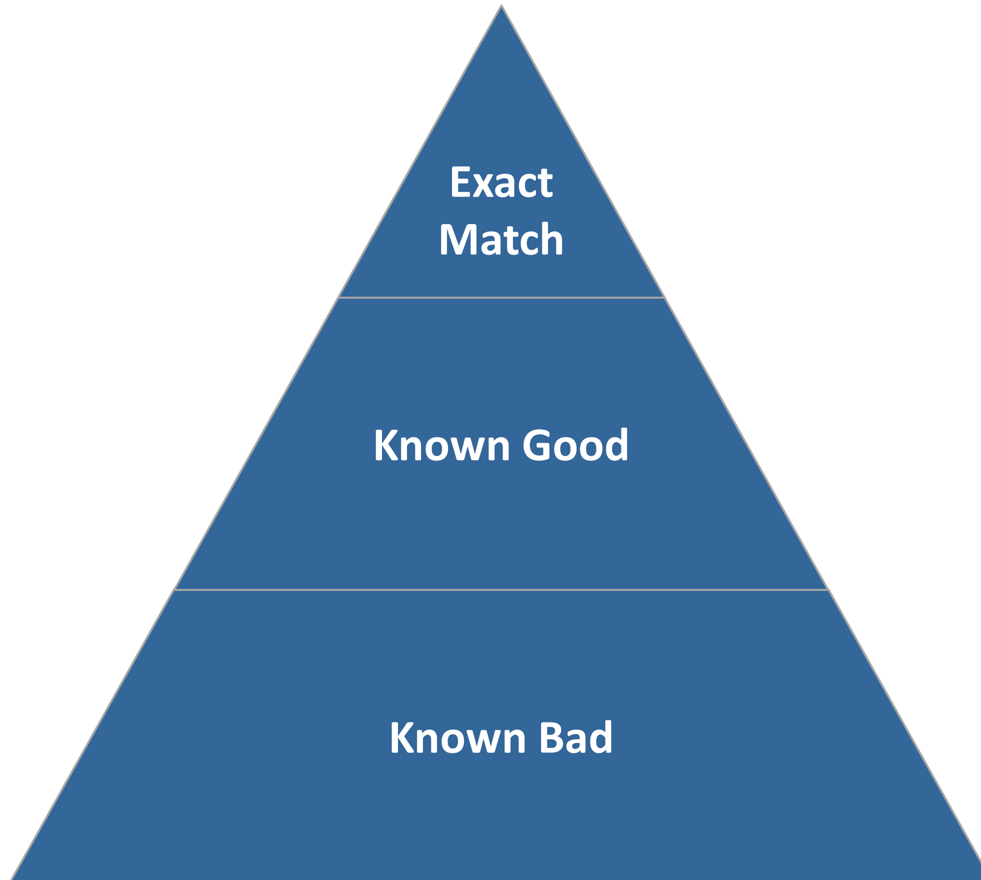
Input that is not directly entered by the user is typically less prone to validation

Attacks discussed in this section apply to external input from **any** client-side source

- Standard form input control
- Read-only HTML form controls (drop down lists, radio buttons, hidden fields, etc)
- HTTP Cookie Values
- HTTP Headers
- Embedded URL parameters (e.g., in the GET request)

RISK ADVISORY

Data Validation



- Data Validation is typically done using one of three basic approaches
- All input must be properly validated on the server (not the client) to ensure that malicious data is not accepted and processed by the application

Exact Match Validation

- Data is validated against a list of explicit known values
- Application footprint or “application attack surface” defined
- Provides the strongest level of protection against malicious data
- Often not feasible when a large number of possible good values are expected
- May require code modification any time input values are changed or updated

Example: Acceptable input is **yes** or **no**
if ($input eq“yes” or $input eq “no”)

Exact Match Validation Example

Validates the variable *gender* against 2 known values (.NET)

```
static bool validateGender(String gender)
  {
      if (gender.equals("Female"))
          return true;
      else if (gender.equals("Male"))
          return true;
      else
          return false; //attack SOUND THE ALARM! BEEP BEEP!
}
```

Exact Match Validation Example

Validates the variable *gender* against 2 known values (Java)

```
static boolean validateGender (String
gender) {
    if (gender.equals (“Female“))
        return true;
    else if (gender.equals (“Male“))
        return true;
    else
        return false; //attack
}
```

Known Good Validation

- Often called "white list" validation
- Data is validated against a list of allowable characters and patterns
- Typically implemented using regular expressions to match known good data patterns
- Data type cast/convert functions can be used to verify data conforms to a certain data type (i.e. Int32)
- Expected input character values must be clearly defined for each input variable
- Care must be taken if complex regular expressions are used
- A common mistake is to forget to anchor the expression with ^ and $

Regular Expression Syntax

Symbol	Match
^	Beginning of input string
$	End of input string
*	Zero or more occurrences of previous character, short for {0,}
+	One or more occurrences of previous character, short for {1,}
?	Zero or one occurrences of previous character, short for {0,1}
{n,m}	At least n and at most m occurrences of previous character
.	Any single character, except '\n'
[xyz]	A character set (i.e. any one of the enclosed characters)
[^xyz]	A negative character set (i.e. any character except the enclosed)
[a-z]	A range of characters

Regular Expression Syntax - shortcuts

Symbol	Match
\d	Any digit, short for [0-9]
\D	A non-digit, short for [^0-9]
\s	A whitespace character, short for [\t\n\x0b\r\f]
\S	A non-whitespace character, for short for [^\s]
\w	A word character, short for [a-zA-Z_0-9]
\W	A non-word character [^\w]
\S+	Several non-whitespace characters

Examples 123-56-3454

Example 1	Validating SSN entry `if ($input=~/^[0-9]{9}$/)`
Example 2	Validating entry of a last name `if ($input=~/^[A-Za-z][-.'0-9A-Za-z]{1,256}$/)`
Example 3	Validating SSN entry (Short cut) `if ($input=~/^\d{9}$/)`

RISK ADVISORY

Regular Expressions - Templates

Field	Expression	Format Samples	Description
Name	^[a-zA-Z-'\s]{1,40}$	John Doe	Validates a name. Allows up to 40 uppercase and lowercase characters and a few special characters that are common to some names. You can modify this list.
Social Security Number	^\d{3}-\d{2}-\d{4}$	111-11-1111	Validates the format, type, and length of the supplied input field. The input must consist of 3 numeric characters followed by a dash, then 2 numeric characters followed by a dash, and then 4 numeric characters.
Phone Number	^[01]?[- .]?(\([2-9]\d{2}\)|[2-9]\d{2})[- .]?\d{3}[- .]?\d{4}$	(425) 555-0123	Validates a U.S. phone number. It must consist of 3 numeric characters, optionally enclosed in parentheses, followed by a set of 3 numeric characters and then a set of 4 numeric characters.
E-mail	^[a-zA-Z0-9.!#$%&'*+/=?^_`{\|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$	someone@example.com	Validates an e-mail address. (per the HTML5 specification http://www.w3.org/TR/html5/forms.html#valid-e-mail-address).
URL	^(ht|f)tp(s?)\:\/\/[0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*(:(0-9)*)*(\/?)([a-zA-Z0-9\-\.\?\,\'\/\\\+&%\$#_]*)?$	http://www.microsoft.com	Validates a URL
ZIP Code	^(\d{5}-\d{4}|\d{5}|\d{9})$\|^([a-zA-Z]\d[a-zA-Z] \d[a-zA-Z]\d)$	12345	Validates a U.S. ZIP Code. The code must consist of 5 or 9 numeric characters.
Password	(?!^[0-9]*$)(?!^[a-zA-Z]*$)^([a-zA-Z0-9]{8,10})$		Validates a strong password. It must be between 8 and 10 characters, contain at least one digit and one alphabetic character, and must not contain special characters.
Non- negative integer	^\d+$	0	Validates that the field contains an integer greater than zero.
Currency (non-negative)	^\d+(\.\d\d)?$	1	Validates a positive currency amount. If there is a decimal point, it requires 2 numeric characters after the decimal point. For example, 3.00 is valid but 3.1 is not.
Currency (positive or negative)	^(-)?\d+(\.\d\d)?$	1.2	Validates for a positive or negative currency amount. If there is a decimal point, it requires 2 numeric characters after the decimal point.

RISK ADVISORY

Regular Expressions

Regular Expressions is a term used to refer to a pattern-matching technology for processing text

Although there is no standards body governing the regular expression language, Perl 5, by virtue of its popularity, has set the standard for regular expression syntax

A Regular Expression itself is a string that represents a pattern, encoded using the regular expression language and syntax

RISK ADVISORY

Regular Expression - Zend

zend framework 2

```
$validator = new Zend_Validate_Regex(array('pattern' => '/^Test/');
$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns true
$validator->isValid("Pest"); // returns false
```

RISK ADVISORY

Data Validation Techniques

Validates against a regular expression representing the proper expected data format (10 alphanumeric characters) (.NET)

```
using System.Text.RegularExpressions;

static bool validateUserFormat(String userName) {

    bool isValid = false; //Fail by default

    // Verify that the UserName is 1-10 character alphanumeric

    isValid = Regex.IsMatch(userName, @"^[A-Za-z0-9]{10}$");

    return isValid;

}
```

Regular Expressions - assertions

Assert a string is 8 or more characters:
(?=.{8,})

Assert a string contains at least 1 lowercase letter (zero or more characters followed by a lowercase character):
(?=.[a-z])*

Assert a string contains at least 1 uppercase letter (zero or more characters followed by an uppercase character):
(?=.[A-Z])*

Assert a string contains at least 1 digit:
(?=.[\d])*

So if you want to match a string at least 6 characters long, with at least one lower case and at least one uppercase letter you could use something like:
^.(?=.{6,})(?=.*[a-z])(?=.*[A-Z]).*$*

RISK ADVISORY

Known Good Validation Example

Validates against a regular expression representing the proper expected data format (10 alphanumeric characters) (Java)

```
import java.util.regex.*;
static boolean validateUserFormat(String userName){
    boolean isValid = false; //Fail by default
    try{
        // Verify that the UserName is 10 character alphanumeric
        if (Pattern.matches("^[A-Za-z0-9]{10}$", userName))
            isValid=true;
    } catch(PatternSyntaxException e) {
      System.out.println(e.getDescription());
    }
    return isValid;
}
```

RISK ADVISORY

Known Good Example

zend framework 2

```
$validator = new Zend_Validate_Alnum();

if ($validator->isValid('Abcd12')) {
        // value contains only allowed chars
} else {
        // false
}
```

RISK ADVISORY

Known Good Example - Chains

zend framework 2

```
// Create a validator chain and add validators to it
$validatorChain = new Zend_Validate();

$validatorChain->addValidator(
new Zend_Validate_StringLength(array('min' => 6, 'max' => 12)))
		->addValidator(new Zend_Validate_Alnum());

// Validate the username
		if ($validatorChain->isValid($username)) {
		// username passed validation
		} else {
		// username failed validation; print reasons or whatever.....
		foreach ($validatorChain->getMessages() as $message) {
		echo "$message\n";
		}
```

RISK ADVISORY

Known Bad Validation

- Often called "BlackList" validation
- Data is validated against a list of characters that are deemed to be dangerous or unacceptable
- Useful for preventing specific characters from being accepted by the application
- Provides the weakest method of validation against malicious data
- Susceptible to bypass using various forms of character encoding

Example: Validating entry into generic text field

```
if ($input !~/[\r\t\n><();\\+&%’”*\|]/)
```

Known Bad Validation Example

Validates against a regular expression of known bad input strings (.Net)

```
using System.Text.RegularExpressions;
static boolean checkMessage(string messageText){
    bool isValid = false; //Fail by default
    // Verify input doesn’t contain any < , >
    isValid = !Regex.IsMatch(messageText, @"[><]");
    return isValid;
}
```

RISK ADVISORY

Known Bad Validation Example

Validates against a regular expression of known bad input strings (Java)

```
import java.util.regex.*;
static boolean checkMessage (string messageText) {
   boolean isValid = false; //Fail by default
   try {
         Pattern P = Pattern.compile (“<|>”,
             Pattern.CASE_INSENSITIVE |  Pattern.MULTILINE);
         Matcher M = p.matcher(messageText);
         if (!M.find())
             isValid = true;
   } catch(Exception e) {
        System.out.println(e.toString());
   }
   return isValid;
}
```

Bounds Checking

All external input must also be properly validated to ensure that excessively large input is rejected	**Length checking:** A maximum length check should be performed on all incoming application data. Careful about name length! Lokelani Keihanaikukauakahihuliheekahaunaele is a real person!
Input that exceeds the appropriate length or size limits must be rejected and not processed by the application	**Size checking:** A maximum size check should be performed on all incoming data files

Bounds Checking – Example

Unbounded Reading of a file

The following code reads a String from a file.

Because it uses the readLine() method, it will read an unbounded amount of input until a <newline> (\n) charter is read.

```
InputStream Input = inputfileFile.getInputStream(Entry);

Reader inpReader = new InputStreamReader(Input);

BufferedReader br = new BufferedReader(inpReader);

String line = br.readLine();
```

This could be taken advantage of and cause an OutOfMemoryException or to consume a large amount of memory which shall affect performance and initiate costly garbage collection routines.

RISK ADVISORY

Bounds checking

```
$validator = new Zend_Validate_StringLength(array('max' => 6));
$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns false
```

Bounds checking – File size

zend framework 2

```
$upload = new Zend_File_Transfer();

// Limit the size of all files to be uploaded to 40000 bytes
$upload->addValidator('FilesSize', false, 40000);

// Limit the size of all files to be uploaded to maximum 4MB and mimimum 10kB
$upload->addValidator('FilesSize', false, array('min' => '10kB', 'max' => '4MB'));
```

Native Validation Controls

Many development platforms have native validator controls, such as Jakarta and .NET

.NET Validator Controls:

RequiredFieldValidator, CompareValidator, RangeValidator, RegularExpressionValidator, CustomValidator, ValidateRequest

Jakarta Commons Validator:

required, mask, range, maxLength, minLength, datatype, date, creditCard, email, regularExpression

Escaping vs. Rejecting

When validating data, one can either reject data failing to meet validation requirements or attempt to “clean” or escape dangerous characters

Failed validation attempts should always reject the data to minimise the risk that sanitisation routines will be ineffective or can be bypassed

Error messages displayed when rejecting data should specify the proper format for the user to enter appropriate data

Error messages should not redisplay the input the user has entered

The Problem with Escaping

```
--- Snip ---
 page_template = request.queryString("page")
 replace(page_template , "/", "\")
 replace(page_template, "..\", "")
 getFile(page_template)
--- End Snip ---
```

http://www.example.com/content/default.jsp?page=info.htm

- Page_template = info.htm <- First Pass
- Page_template = info.htm <- Second Pass

http://www.example.com/content/default.jsp?page=../web.xml

- Page_template = ..\ web.xml <- First Pass
- Page_template = web.xml <- Second Pass

http://www.example.com/content/default.jsp?page=....//web.xml

- Page_template = ...\\web.xml <- First Pass
- Page_template = ..\web.xml <- Second Pass

RISK ADVISORY

Input Based Attacks

Malicious user input can be used to launch a variety of attacks against an application. These attacks include, but are not limited to:

Parameter Manipulation	■ Cookie poisoning ■ Hidden field manipulation
Content injection	■ SQL ■ HTTP Response Splitting ■ Operating system calls ■ Command insertion ■ XPath
Cross site scripting	■ …
Buffer overflows	■ Format string attacks

Parameter Manipulation

Applications typically pass parameters that determine what the user can do or see

Unauthorised access to application data can be obtained by manipulating parameter values, if record-level authorisation checks are not performed within the application code

Many applications tend to pass sensitive parameters back and forth rather than using server session variables to store the information

Very common to see this in

- HTTP cookies
- “hidden” HTML form fields

Parameter Manipulation

Database record index numbers are often passed as page parameters to view a specific record

If there is a relatively small range of possible index values, sequential parameter values can by cycled to view other records

Even-non sequential indexing can be attacked within a small range of potential values

Manipulating parameter values can be trivial when other obvious valid Values exist:

- Username=joe (change to username=mary)
- Privilege=user (change to privilege=admin)
- Price=100.00 (change to price=1.00)

RISK ADVISORY

Testing for Parameter Manipulation

Identify areas in the application where records appear to be displayed based on input parameters

Attempt to access records using other known valid parameter values

For parameter values that seem to fall within a variable range, cycle through the range to search for other valid records

Identify all data being passed in hidden fields. Attempt to determine:

- What the data is being used for
- Whether use of a hidden field seems appropriate

RISK ADVISORY

Defenses Against Parameter Manipulation

Data that should not be altered should never be passed to the client

Always perform validation on the server

Retrieve the information from the client once, validate it and keep it on the server associated with that user's session

Use the most restrictive input validation method possible

- Use **Exact Match Validation** to verify that parameters values are appropriate for the specific transaction or user's permission level
- In situations where a query or hash table lookup is required, **Known Good Validation** should be used to validate the parameter before performing the lookup

Summary

- Don’t ever trust user input
- Where possible, use whitelist validation
- Perform input validation at earliest possible stage
- Layers of defense
- Use in-built validator routines

RISK ADVISORY