# Input Validation

Ryan Cranston, Petal Michaud, Julian Welge

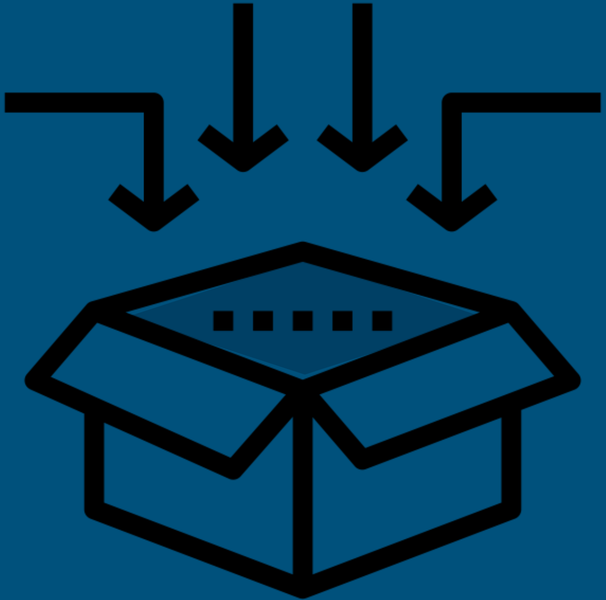
## Overview

- Definition - R
- Goals of input validation - R
- Strategies -R
    - Syntax and semantic validation
    - Client side versus server side
    - Whitelisting versus Blacklisting
- Implementation -P
    - File upload validation
    - Email address validation
    - Regular Expressions
- Consequences - P
    - SQL Injection
    - Buffer Overflow
    - XSS
- Input Validation in Java/C/JavaScript -J
    - Sample code
- Limitations -J

## Definition of Input Validation

*Input validation* (also known as data validation) is the proper testing of any input entered by a user or application. Input validation prevents improperly formed data from entering an system. Incorrect input validation can lead to injection attacks, memory leakage, and compromised systems.

## Goals



- ➔ Ensure only properly formed data is entering the program
- ➔ Preventing malformed data from persisting in the database
- ➔ Prevent malfunction of various downstream components.
- ➔ All potentially untrusted sources should be subject to input validation

Input Validation should **not** be used as the primary method of preventing XSS, SQL Injection and other attacks

## Strategies

# Syntactic and Semantic Validation

Syntactic validation should enforce correct syntax of structured fields (e.g. SSN, date, currency symbol).

Semantic validation should enforce correctness of their values in the specific business context (e.g. start date is before end date, price is within expected range).

## Client-side vs Server-side

Any JavaScript input validation performed on the client can be bypassed by an attacker that disables JavaScript or uses a Web Proxy. Ensure that any input validation performed on the client is also performed on the server.

## Allow listing vs Block listing

Allow list is a mechanism which explicitly allows a list of things when everything is denied by default

Block list is a mechanism which a list of things are regarded as unacceptable or untrustworthy and should be excluded or avoided.

A common mistake to use Block list validation in order to try to detect possibly dangerous characters and patterns(ex: ‘ ,1=1, <script>), but this thinking is flawed as it is difficult to account for all possibilities.On the other hand, Allow list is appropriate for all input fields provided by the user, since defining exactly what is authorized, by definition everything else is not permitted.

## Best Practices

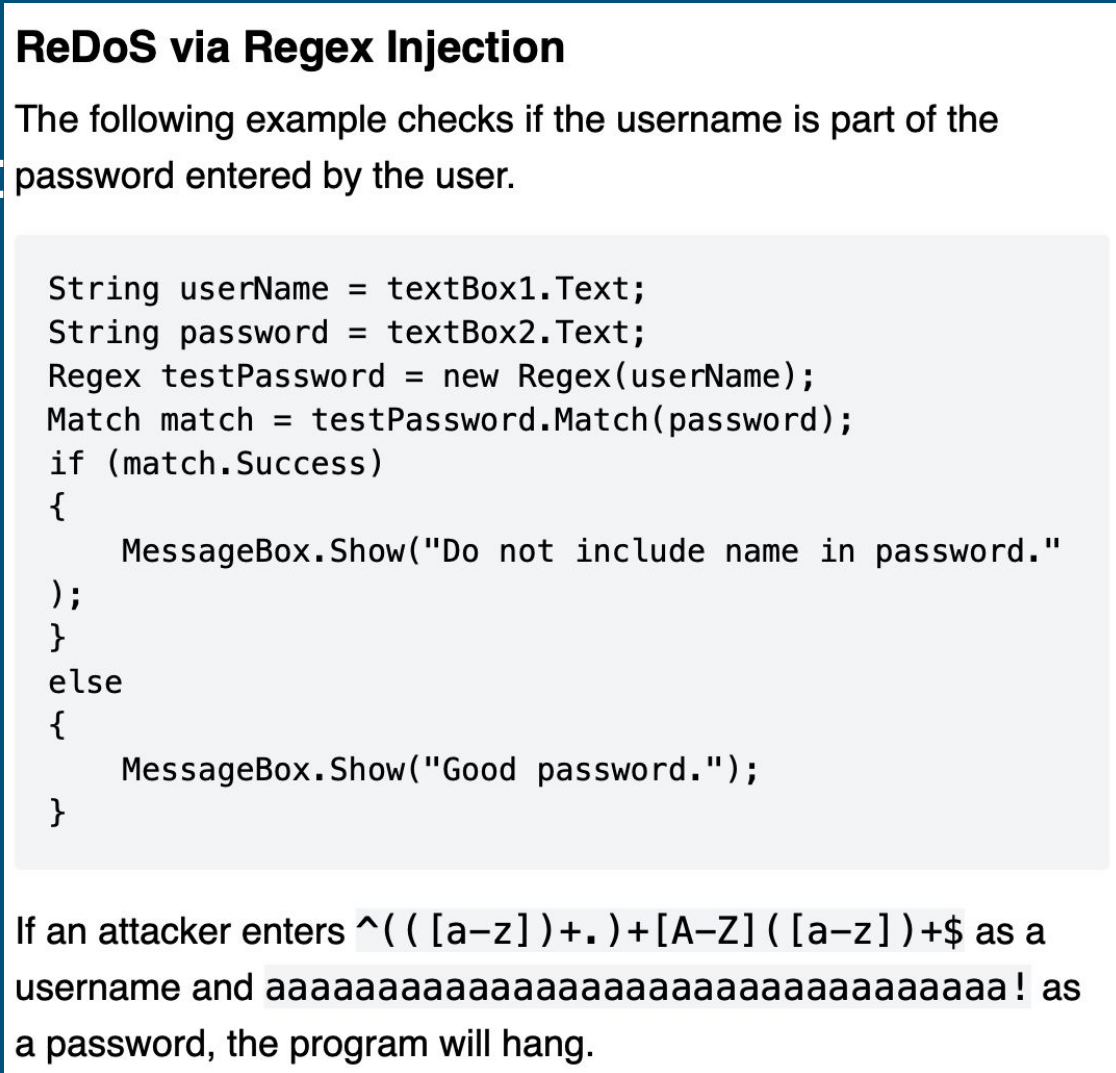
- Validate all inputs
- Use a centralized validation engine
- Use a whitelist filter with range
- Check input for embedded HTML scripts
- If you accept UTF-8 encoding, verify that the input does not contain illegal UTF-8 sequences
- Validate email addresses
- If the input is a URL, decode it in the host system before validation
- If the input is a filename, resolve the name in the host OS during validation.
- When the input contains XML documents, validate it against its schema before using it
- Escape special characters that cause SQL injection

## Regular Expressions

- A simple language for describing text patterns
- Variants - BRE, ERE, PCRE
- The “|” operator

# Regular Expressions:

- ❖ Risk of ReDOS attacks
- ❖ Evil Regexes
  - ➢ Repetition
  - ➢ Overlapping
- ❖ Backtracking
- ❖ Example ->



## ReDoS via Regex Injection

The following example checks if the username is part of the password entered by the user.

```
String userName = textBox1.Text;
String password = textBox2.Text;
Regex testPassword = new Regex(userName);
Match match = testPassword.Match(password);
if (match.Success)
{
    MessageBox.Show("Do not include name in password."
);
}
else
{
    MessageBox.Show("Good password.");
}
```

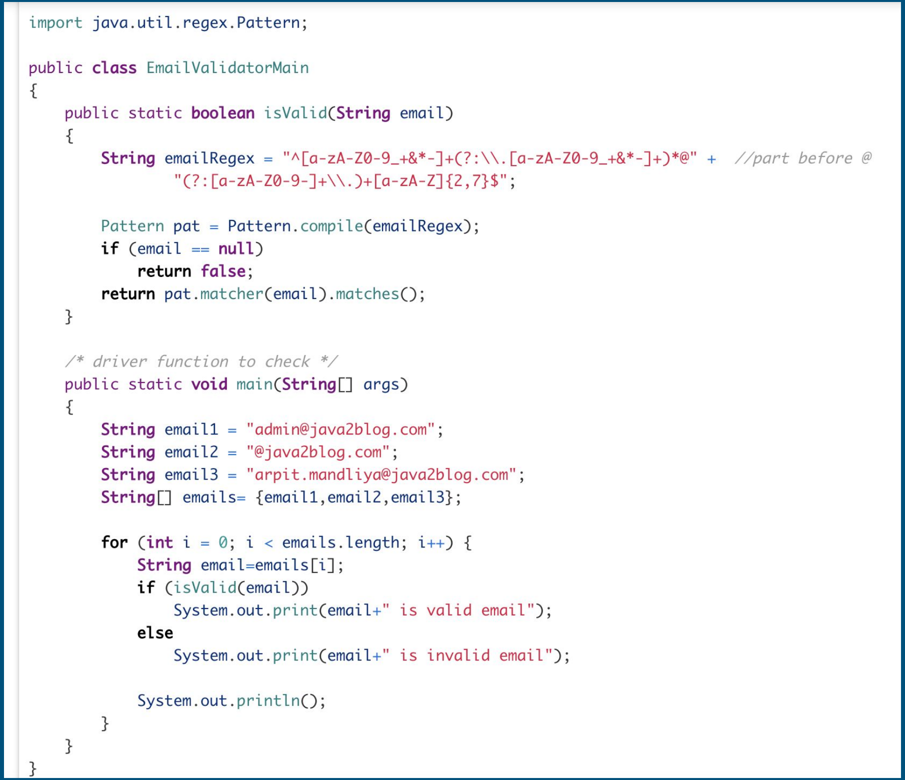
If an attacker enters `^(([a-z])+.)+[A-Z]([a-z])+$` as a username and `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!` as a password, the program will hang.

## Tools/Examples

# Java

- Appropriate Scanner methods ie nextInt()
- Algorithms that check sum or check length (ie Luhn Algorithm)
- Type safe

## examples

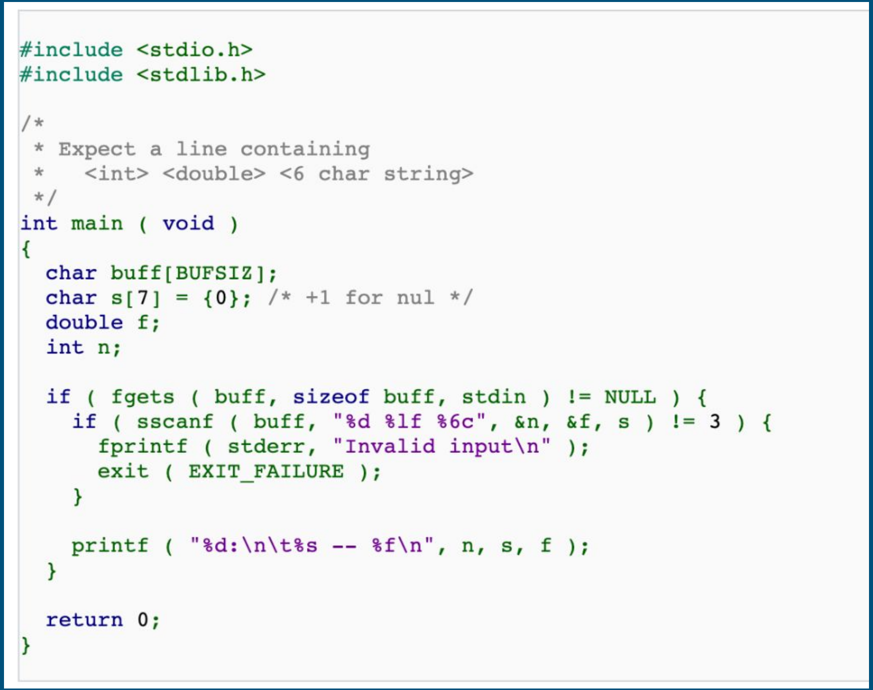


```
import java.util.regex.Pattern;

public class EmailValidatorMain
{
    public static boolean isValid(String email)
    {
        String emailRegex = "^[a-zA-Z0-9_+&*-]+(?:\\.[a-zA-Z0-9_+&*-]+)*@" +  //part before @
                "(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,7}$";

        Pattern pat = Pattern.compile(emailRegex);
        if (email == null)
            return false;
        return pat.matcher(email).matches();
    }

    /* driver function to check */
    public static void main(String[] args)
    {
        String email1 = "admin@java2blog.com";
        String email2 = "@java2blog.com";
        String email3 = "arpit.mandliya@java2blog.com";
        String[] emails= {email1,email2,email3};

        for (int i = 0; i < emails.length; i++) {
            String email=emails[i];
            if (isValid(email))
                System.out.print(email+" is valid email");
            else
                System.out.print(email+" is invalid email");

            System.out.println();
        }
    }
}
```

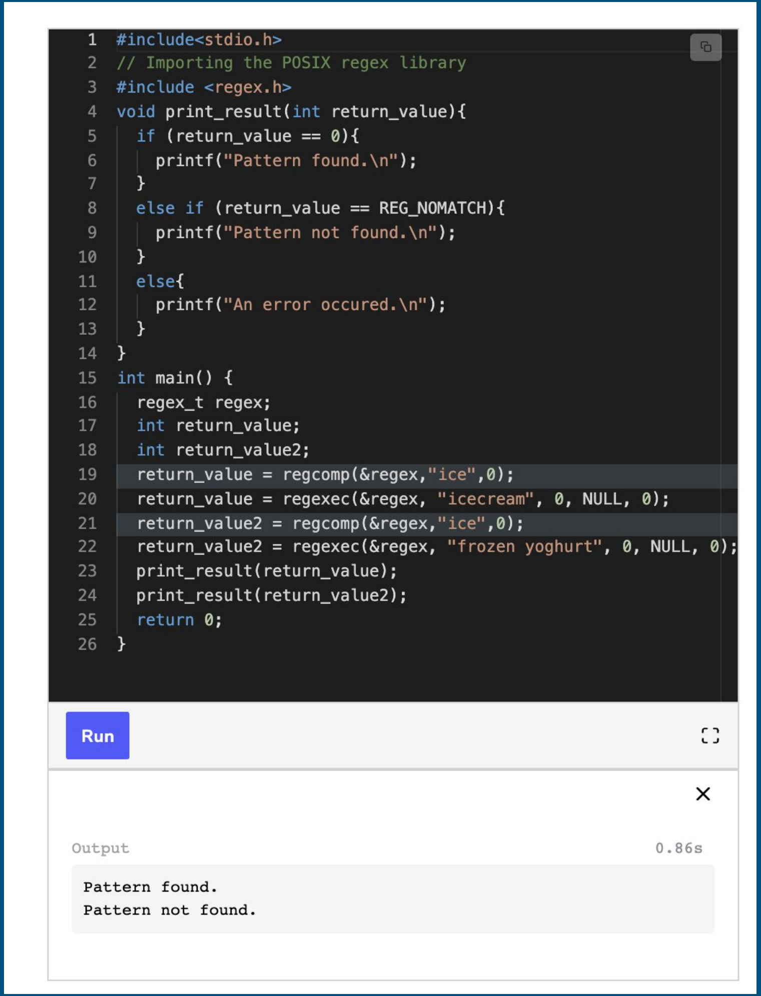
## C

- Utilization of buffers
- Specifying length of what’s stored with fgets()
- Not type safe

## example



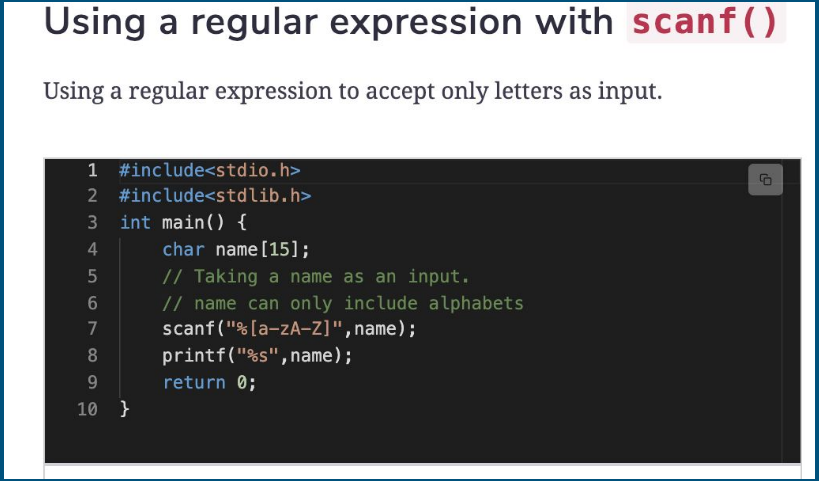
```c
#include <stdio.h>
#include <stdlib.h>

/*
 * Expect a line containing
 *   <int> <double> <6 char string>
 */
int main ( void )
{
  char buff[BUFSIZ];
  char s[7] = {0}; /* +1 for nul */
  double f;
  int n;

  if ( fgets ( buff, sizeof buff, stdin ) != NULL ) {
    if ( sscanf ( buff, "%d %lf %6c", &n, &f, s ) != 3 ) {
      fprintf ( stderr, "Invalid input\n" );
      exit ( EXIT_FAILURE );
    }

    printf ( "%d:\n\t%s -- %f\n", n, s, f );
  }

  return 0;
}
```



```c
#include<stdio.h>
// Importing the POSIX regex library
#include <regex.h>
void print_result(int return_value){
  if (return_value == 0){
    printf("Pattern found.\n");
  }
  else if (return_value == REG_NOMATCH){
    printf("Pattern not found.\n");
  }
  else{
    printf("An error occured.\n");
  }
}
int main() {
  regex_t regex;
  int return_value;
  int return_value2;
  return_value = regcomp(&regex,"ice",0);
  return_value = regexec(&regex, "icecream", 0, NULL, 0);
  return_value2 = regcomp(&regex,"ice",0);
  return_value2 = regexec(&regex, "frozen yoghurt", 0, NULL, 0);
  print_result(return_value);
  print_result(return_value2);
  return 0;
}
```

Output 0.86s

```
Pattern found.
Pattern not found.
```



## Using a regular expression with `scanf()`

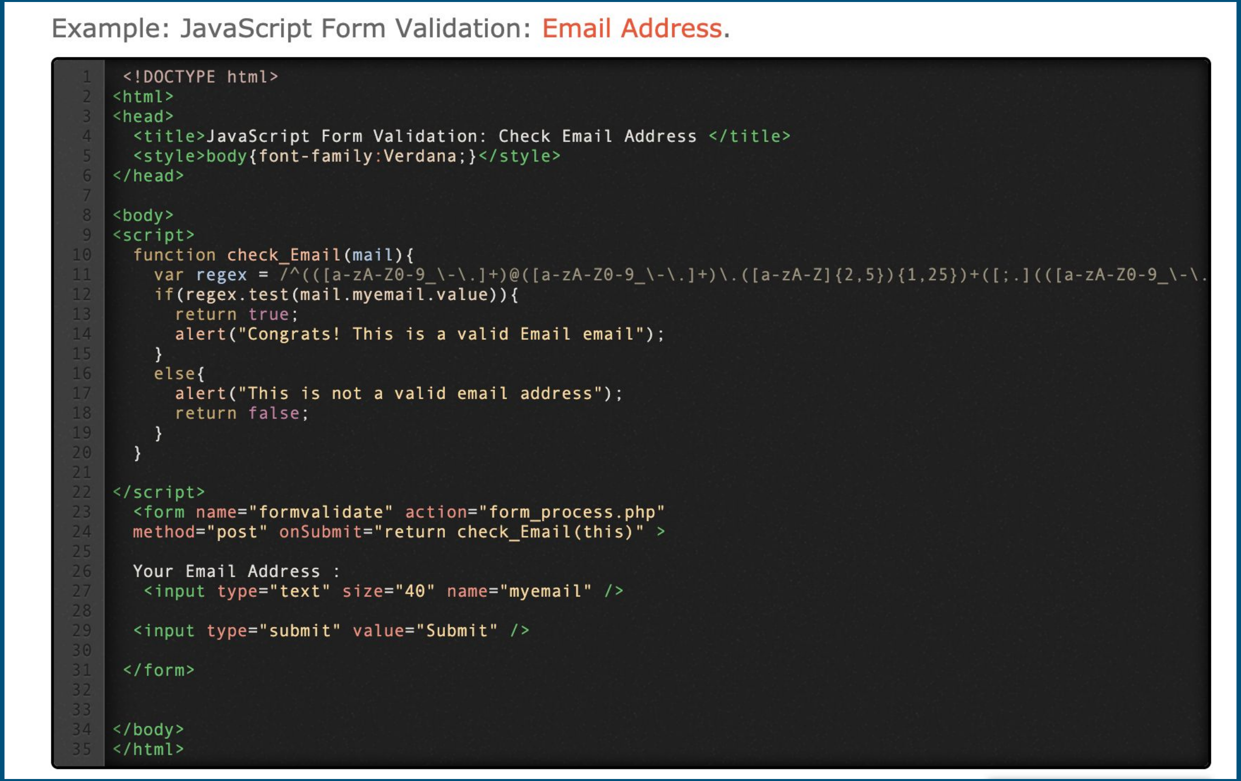
Using a regular expression to accept only letters as input.

```c
#include<stdio.h>
#include<stdlib.h>
int main() {
    char name[15];
    // Taking a name as an input.
    // name can only include alphabets
    scanf("%[a-zA-Z]",name);
    printf("%s",name);
    return 0;
}
```

## javascript

- Forms
- client - side

# example



Example: JavaScript Form Validation: Email Address.

```
 <!DOCTYPE html>
<html>
<head>
  <title>JavaScript Form Validation: Check Email Address </title>
  <style>body{font-family:Verdana;}</style>
</head>

<body>
<script>
  function check_Email(mail){
    var regex = /^(([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5}){1,25})+([;.](([a-zA-Z0-9_\-\.
    if(regex.test(mail.myemail.value)){
      return true;
      alert("Congrats! This is a valid Email email");
    }
    else{
      alert("This is not a valid email address");
      return false;
    }
  }

</script>
  <form name="formvalidate" action="form_process.php"
  method="post" onSubmit="return check_Email(this)" >

  Your Email Address :
   <input type="text" size="40" name="myemail" />

  <input type="submit" value="Submit" />

 </form>


</body>
</html>
```

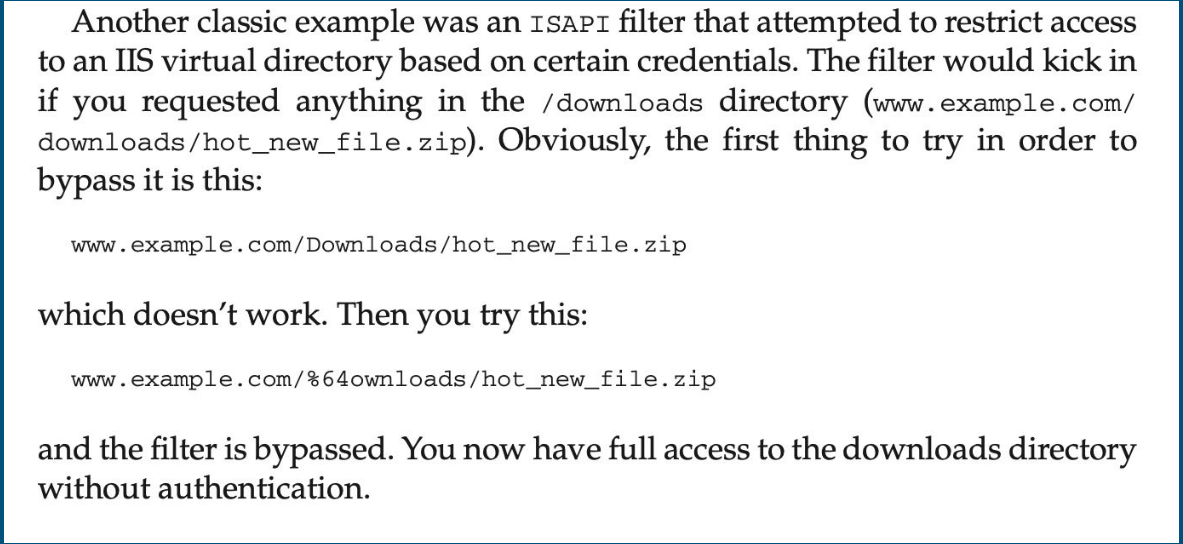
## Limitations

## Initial Limiting Factors

- Can be Tedious/Overwhelming
- Only using common provided input-form validation, relying on client-side validation
- Common first target

## Alternate encodings used to bypass filters

Ex. from The Shellcoder's Handbook



Another classic example was an ISAPI filter that attempted to restrict access to an IIS virtual directory based on certain credentials. The filter would kick in if you requested anything in the /downloads directory (www.example.com/downloads/hot_new_file.zip). Obviously, the first thing to try in order to bypass it is this:

```
www.example.com/Downloads/hot_new_file.zip
```

which doesn't work. Then you try this:

```
www.example.com/%64ownloads/hot_new_file.zip
```

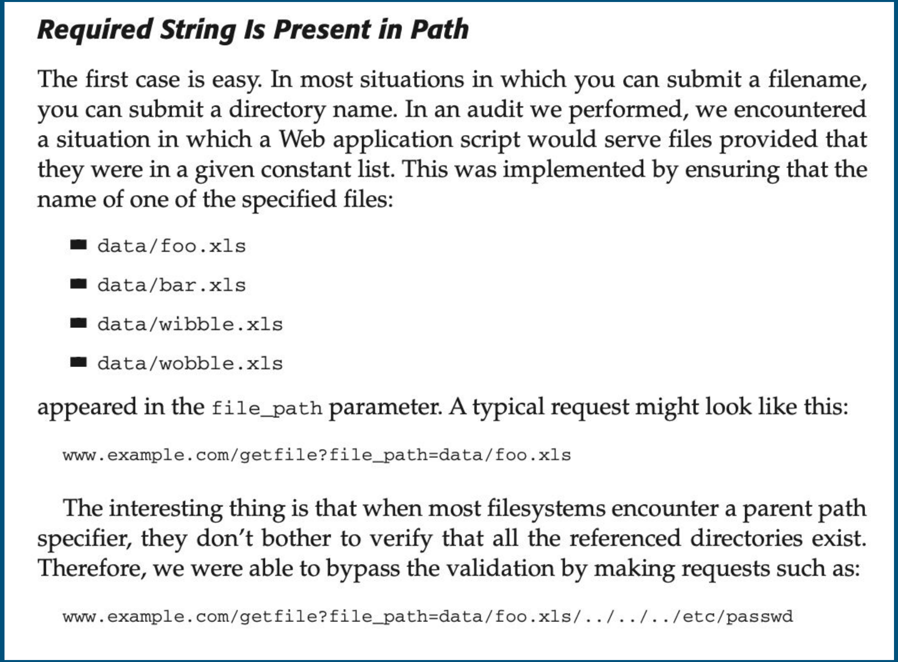
and the filter is bypassed. You now have full access to the downloads directory without authentication.

## File-Handling Features

Tricking applications so that:

- It believes a required string is present in a file path
- It believes a prohibited string is not present in a file path
- Does some wrong behavior if file handling is based on the file extension

# Examples of file-handling dilemmas



### *Required String Is Present in Path*

The first case is easy. In most situations in which you can submit a filename, you can submit a directory name. In an audit we performed, we encountered a situation in which a Web application script would serve files provided that they were in a given constant list. This was implemented by ensuring that the name of one of the specified files:

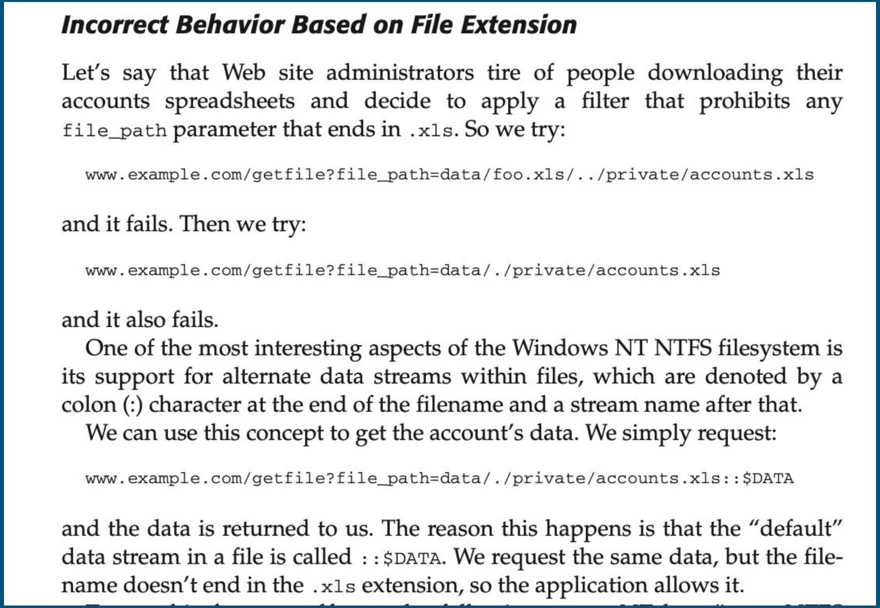
- `data/foo.xls`
- `data/bar.xls`
- `data/wibble.xls`
- `data/wobble.xls`

appeared in the `file_path` parameter. A typical request might look like this:

```
www.example.com/getfile?file_path=data/foo.xls
```

The interesting thing is that when most filesystems encounter a parent path specifier, they don't bother to verify that all the referenced directories exist. Therefore, we were able to bypass the validation by making requests such as:

```
www.example.com/getfile?file_path=data/foo.xls/../../../etc/passwd
```



### *Incorrect Behavior Based on File Extension*

Let's say that Web site administrators tire of people downloading their accounts spreadsheets and decide to apply a filter that prohibits any `file_path` parameter that ends in `.xls`. So we try:

```
www.example.com/getfile?file_path=data/foo.xls/../private/accounts.xls
```

and it fails. Then we try:

```
www.example.com/getfile?file_path=data/./private/accounts.xls
```

and it also fails.

One of the most interesting aspects of the Windows NT NTFS filesystem is its support for alternate data streams within files, which are denoted by a colon (:) character at the end of the filename and a stream name after that.

We can use this concept to get the account's data. We simply request:

```
www.example.com/getfile?file_path=data/./private/accounts.xls::$DATA
```

and the data is returned to us. The reason this happens is that the "default" data stream in a file is called `::$DATA`. We request the same data, but the filename doesn't end in the `.xls` extension, so the application allows it.

## Sources

"Input Validation Cheat Sheet." Input Validation - OWASP Cheat Sheet Series, cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html.

"Input Validation." WhiteHat Security Glossary, www.whitehatsec.com/glossary/content/input-validation.

"Best Practices in Input Validation" https://www.paladion.net/blogs/best-practices-in-input-validation

"Input Validation in Java using Scanner" java2blog, https://java2blog.com/input-validation-java/#Scanner_Methods_to_Validate_User_Input

https://www.netsparker.com/blog/web-security/input-validation-errors-root-of-all-evil/

"The Shellcoder's Handbook" - Chris Anley, John Heasman, et. al.

"Regular expression Denial of Service - ReDoS" https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS