SANS Institute
Information Security Reading Room

AppSec - Protecting Your Web Apps: Two Big Mistakes and 12 Practical Tips to Avoid Them

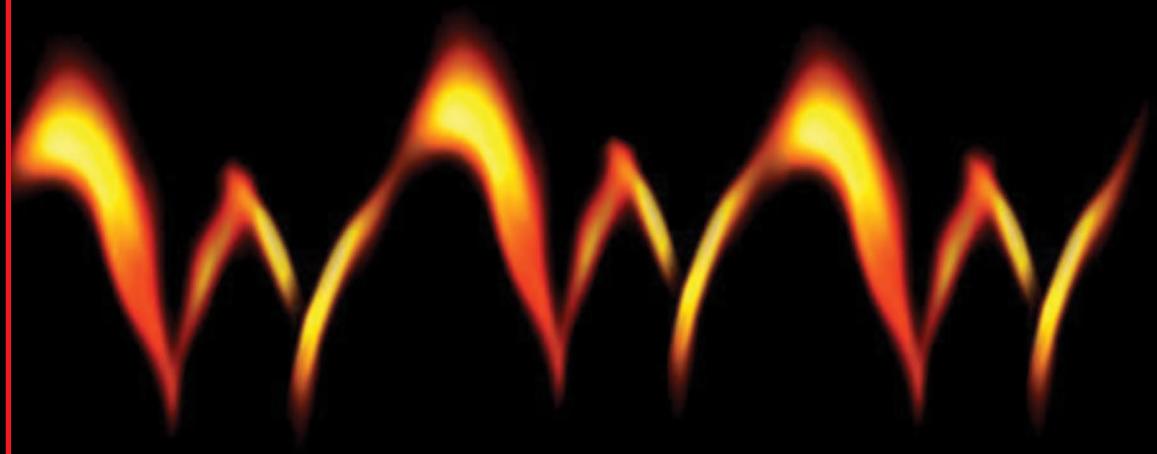
Ed Skoudis and Frank Kim

Copyright SANS Institute 2021. Author Retains Full Rights.

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

SANS

Working Papers in Application Security



Protecting Your Web Apps

Two Big Mistakes and 12 Practical Tips to Avoid Them

Written by Frank Kim and Ed Skoudis

Volume 1.1

SANS

Working Papers in Application Security

Protecting Your Web Apps

Two Big Mistakes and 12 Practical Tips to Avoid Them

Written by
Frank Kim and
Ed Skoudis

This article is the first in a series dedicated to application security and secure coding practices. In coming months, the SANS Institute will release additional articles like this that cover other aspects of application security and secure coding. However, please don't wait until the end of the series to start following these tips. Make a commitment today to ensuring that your applications and code are more secure.

www.sans.org/info/39149

Increasingly, computer attackers are exploiting flaws in Web applications, exposing enterprises to significant threats, including Personally Identifiable Information breaches and uploads of malware onto vulnerable corporate Websites for distribution to customer browsers. Many of these Web application vulnerabilities are a direct result of improper input validation and output filtering, which leads to numerous kinds of attacks, including cross-site scripting (XSS), SQL injection, command injection, buffer overflows and many others. This article describes some of the best defenses against such attacks, which every Web application developer should master.

Whenever an application needs to gather information from a user or a browser, that information must be validated carefully to remove potential attack strings. Likewise, data sent back from the Web server to a browser should be filtered to make sure that exploits that an attacker has managed to sneak onto a Web server aren't served back to unwitting consumers who visit the site.

As an example, consider a Web site that needs to gather from a user a specific name and age for processing. One alpha field and one numeric field would likely suffice to gather this information. Without proper validation, however, attackers might be able to use other characters, including semicolons, greater-than or less-than symbols, and quotation marks to exploit the application. If a software developer does not properly screen all forms of user input to prevent certain characters from being entered, the software may be exploitable in numerous different ways. Input validation acts as a shield by deflecting potentially malicious characters. Likewise, output filtering prevents the Website from shooting back an exploit to browsers. Protection in both directions is needed, and Web application developers are a crucial line of such defense.

Unfortunately, some software developers either leave out input-validation and output-filtering code altogether, or they implement such code poorly so that it does not filter out a truly comprehensive set of potential attack characters. Also, there are dozens of ways to alter or encode data to dodge validation filters: UTF-8, Hex, Unicode, mixed case and many more. Noted Web app security guru RSnake has written a great Web page that shows some different encoding tricks designed to slip cross-site scripting attacks past weak input-validation code http://ha.ckers.org/xss.html.

With that overview in mind, let's look carefully at the methods that developers should employ to prevent input-validation attacks and implement better output filtering. For software developers who want to make sure that their input-validation and output filtering code is up to snuff, use the following as a checklist:

1. Use well known and carefully vetted validation code

Rather than rolling your own code for user input validation, use APIs that have been carefully developed and rigorously tested by others. Some software development firms and large enterprises doing in-house development have defined reusable input validation code for all software they create. Find out if your organization has such code, learn how it works, and then use it. If you don't have such code in-house, you can utilize code from a variety of free sources. For example, Apache Struts is a very popular MVC framework for Web applications built in Java. If you are using Struts, you can leverage its built-in validation framework to apply input validation rules to your data. Additionally, the OWASP Enterprise Security API (ESAPI) project provides interfaces and implementations of important security functions including input validation and output encoding. ESAPI is currently implemented in Java, with .NET and PHP versions in the works. The code examples in this article will utilize the Java version of ESAPI.

SANS

Working Papers in Application Security

Protecting Your Web Apps

Two Big Mistakes and 12 Practical Tips to Avoid Them

Written by
Frank Kim and
Ed Skoudis

2. Specify variable types

Limit the kind of data that can be entered into some fields. For example, if you only need to accept an integer in a field, ensure that your code rejects any other characters that may be entered. One way to accomplish this is by using the **Validator** class from ESAPI. This class has an **isValidInteger** method that can be used as follows in Java code:

```
Validator validator = ESAPI.validator();
boolean isValidInteger =
    validator.isValidInteger("Test_Num", "42", 0, 999, false);
```

As described in the ESAPI Javadoc, the **isValidInteger** method takes five parameters:

- **context** - This is a descriptive name for the parameter we are validating. In this case, we've used the string "Test_Num".
- **input** - The actual data we want to validate. In this case it is the string "42".
- **minValue** - The lowest legal value.
- **maxValue** - The highest legal value.
- **allowNull** - A boolean indicating whether or not the field can be null.

Our code first calls **ESAPI.validator()**. The **ESAPI** class is a locator class that can be used to retrieve default implementations of various ESAPI classes. In this case we use the **validator** method to retrieve ESAPI's default validator and then call the **isValidInteger** method on that **Validator** instance. If the input is a valid integer per our specifications, then the method returns true. Otherwise, our code needs to reject that input. Simple as that!

3. Don't define all possible bad characters; instead accept only the good ones

Trying to create a comprehensive list of all bad characters for all the different kinds of attacks is next to impossible. Thus, when creating validation code, define which characters are acceptable using a whitelist, such as A-Z, a-z, and 0-9, and reject everything else. Such a "deny-all-except-for-certain-allowable-characters" approach is a far stronger way to write validation code. We'll cover some example code that implements this recommendation below, bundled with the next suggestion.

4. Limit the size of input

If you ask for someone's age, keeping it to a three-digit field will cover every reasonable case, even the centenarians in your user population. If you ask for someone's name, a hundred or so characters are reasonable. That way, even if you aren't properly filtering for characters, you are still limiting the real estate that the attacker has to pull off an attack.

To both validate input using a whitelist approach and limit the size of our input, we can once again use the **Validator** class in Java. This time, however, we use the **isValidInput** method as follows:

```
Validator validator = ESAPI.validator();
boolean isValidFirstName =
    validator.isValidInput("Test_AccountName", "Checking1",
    "AccountName", 100, false);
```

Protecting Your Web Apps

Two Big Mistakes and 12 Practical Tips to Avoid Them

Written by
Frank Kim and
Ed Skoudis

The **isValidInput** method takes five parameters:

- **context** - As before, this is a descriptive name for the parameter we are validating. In this case we use the string "Test_AccountName".
- **input** - The actual data we want to validate. Here, we are analyzing the string "Checking1".
- **type** - The name of a regular expression that maps to the actual regular expression in the ESAPI.properties configuration file.
- **maxLength** - The maximum length allowed for the input data.
- **allowNull** - A boolean indicating whether or not the field can be null.

The "AccountName" regular expression we defined in the ESAPI configuration file is **^[a-zA-Z0-9]*$**

This regex essentially says that the input is allowed to only contain lower and uppercase letters and numbers from zero to nine. Additionally, in the **isValidInput** call, we specified that the maxLength for the input cannot exceed 100 characters. These input validation rules will block many types of attack.

5. Canonicalize before filtering

If user input is encoded in a fashion that your filters aren't designed to handle, your application very well might get hacked. Thus, whenever you receive user input, convert it to a standard encoding scheme, such as plain ASCII, before applying your filters. This process is known as canonicalization, and it converts a character stream of different encoding patterns to a simpler, consistent format that your filters can handle properly.

Fortunately, ESAPI performs canonicalization for us before doing any input validation. For example, when we called the **isValidInput** method above, ESAPI automatically canonicalized the input data by using the **Encoder** class under the covers. We can also explicitly invoke this functionality by calling the **canonicalize** method as follows (note that Exception handling has been removed for readability):

```
String encoded = "%3Cscript&#x3E;alert%28%27xss&#39%29%3C%2Fscript%3E";
Encoder encoder = ESAPI.encoder();
String decodedString = encoder.canonicalize(encoded);
```

When this code is executed, the **decodedString** variable will contain the value **<script>alert('xss')</script>**

There are many ways to represent different characters depending on the encoding scheme in use. In the example above both **>** and **%3E** are encoded representations of the greater-than character (>). Some simple validation filters may look for the less-than and greater-than characters in an attempt to prevent XSS attacks. However, filters may be bypassed if data is not properly canonicalized before input validation checks are performed.

6. Filter all input

Filter every form of input to your application, including data that comes in via the network, the GUI, hidden form elements, cookies, files read from the file system and so on. Don't assume that one of your fields or input vectors isn't important. Attackers will hunt for weaknesses and exploit them, so cover all of your bases.

SANS

Working Papers in Application Security

Protecting Your Web Apps

Two Big Mistakes and 12 Practical Tips to Avoid Them

Written by
Frank Kim and
Ed Skoudis

7. Filter on the server side

In many applications, attackers might be able to control clients, such as browsers or thick-client GUIs, tweaking their functionality to bypass filtering done at the client. Alternatively, particularly creative attackers may devise their own custom clients designed just to attack your Web application code. JavaScript, pull-down menus, and other client-side filters can improve usability, but they are trivial for attackers to bypass and therefore don't provide security. Thus, you can't rely on security checks at the client side, because it is territory outside of your control. Filter on the server side to protect all back-end functionality.

8. Don't worry about multiple layers of validation

Sometimes, what appears to be a single application will include multiple subsystems. For example, a Web application may utilize a Web service under the covers. The Web application performs data validation before calling the Web service and the Web service in turn performs the same validation checks before it begins processing the data. This design could result in a single set of input getting filtered multiple times as it snakes its way through the application. While that might be a performance concern, it's actually a good thing from a security perspective, especially considering that other clients could call the Web service directly in the future. Multiple layers of filtering help provide defense in depth, making the whole system more secure.

9. Utilize output encoding

Output encoding takes place when data is displayed back to the user or utilized by another component of the application. By properly escaping data before it is used, many types of attacks can be prevented. For example, certain characters, like the less-than and greater-than symbols, are commonly used in XSS attacks. By converting these characters to their HTML entity equivalents (i.e., "<" becomes "<" and ">" becomes ">"), many XSS attacks can be prevented. The **`Encoder`** class contains numerous methods that can be used to encode data for a number of different situations. Specifically, the **`encodeForHTML`** method can be used to prevent HTML-based XSS as follows:

```
Encoder encoder = ESAPI.encoder();
String encodedString = encoder.encodeForHTML("<script>alert
    ('xss')</script>");
```

When this code is executed, the **`encodedString`** variable will contain the following value, which is far less likely to be executed as a script in a browser during an XSS attack attempt:

```
&lt;script&gt;alert&#40;&#39;xss&#39;&#41;&lt;&#47;script&gt;
```

The **`encodeForHTML`** method actually takes a whitelist approach to encoding. It treats a limited set of characters (letters, numbers, commas, periods, dashes, underscores, and spaces) as safe and HTML encodes everything else. When the browser displays HTML encoded data, it renders it for display only and not execution, thereby preventing XSS attacks.

SANS

Working Papers in Application Security

Protecting Your Web Apps

Two Big Mistakes and 12 Practical Tips to Avoid Them

Written by
Frank Kim and
Ed Skoudis

10. Choose the appropriate output encoding

HTML output encoding is only one example of how data may need to be encoded. Depending on where your data is used, other encoding schemes may need to be applied. For example, XSS can also occur if malicious data is utilized in JavaScript. Moreover, data used as part of XML elements, XPATH queries, or operating system commands will also benefit from proper output encoding. Ensure that you perform the appropriate encoding depending on where your data will be utilized. Fortunately, the Encoder class contains **`encodeForJavaScript`**, **`encodeForXML`**, **`encodeForXPath`**, and **`encodeForOS`** methods that provide this functionality for you.

11. Conduct a secure code review

To help ensure that the source code for your application does not contain security vulnerabilities, conduct a professional code review to see if any security issues can be discovered and remediated.

12. Perform a penetration test

To check for any remaining vulnerabilities, subject your application to a controlled penetration test to see if flaws can be identified.

So there you have it. These 12 recommendations can go a long way toward making your application code more secure. They won't make your code bulletproof, but they can thwart a large number of common and very damaging attacks.

About the authors

Ed Skoudis is a SANS Fellow and co-founder of InGuardians, an infosec consulting and cutting-edge research company (www.inguardians.com). Author of the acclaimed SANS SEC504 and SEC560 courses, Ed's expertise includes in-depth penetration testing and incident response.

Frank Kim is a SANS author and instructor and founder of Think Security (www.thinksec.com). He has been developing, designing, and securing Web applications for over ten years. He currently focuses on integrating security into the SDLC by doing architecture reviews, security assessments, code reviews, penetration testing, and training.