How Does GitHub Security Advisory GHSA-r68h-jhhj-9jvm Impact ESAPI?

Kevin W. Wall <kevin.w.wall@gmail.com>

Summary

Description:	The `Validator.isValidSafeHTML` interfaces describe methods with over-promised results and after further analysis, they cannot possibly deliver. As such, they have both been deprecated and marked for removal within a year's time.
Module:	ESAPI Validator
Announced:	As part of the ESAPI 2.5.3.0 release, in this security bulletin and GitHub Security Advisory GHSA-r68h-jhhj-9jvm.
Credits:	ESAPI team
Affects:	All versions of ESAPI 2.x,as well as all versions of ESAPI 1.x (dating back to at least 1.3). This will not be officially "fixed" until these two `isValidSafeHTML` methods are completely removed from ESAPI in a future release in one year's time from the ESAPI 2.5.3.0 release date.
Details:	**May be exploitable (likely as XSS), depending on how you use it.**
GitHub Issue #:	None.
Related:	https://github.com/ESAPI/esapi-java-legacy/security/advisories/GHSA-r68h-jhhj-9jvm
CWE:	CWE-79 or CWE-80, take your pick. Also, if there is a CWE for overreaching promises or developer brain farts, then add those as well.
CVE Identifier:	None. See notes as to why we did not file for a CVE.
CVSSv3 score:	None calculated, as we did not file for a CVE. Take your own stab at it. You certainly will be able to arrive at a more accurate figure than the worst case scenario for libraries as demanded by both FIRST's CVSS User Guide and NIST NVD.

Background

OWASP ESAPI (the OWASP Enterprise Security API) is a free, open source, web application security control library that makes it easier for programmers to write lower-risk applications. The ESAPI for Java library is designed to make it easier for programmers to

retrofit security into existing applications. ESAPI for Java also serves as a solid foundation for new development.

One of the security controls provided by ESAPI for Java is its provision to validate various data types. In particular, ESAPI attempted (in hindsight, mistakenly) to verify if certain input was “safe HTML”, where “safe” in this context meant that the validated input was safe from XSS vulnerabilities. ESAPI leveraged the functionality of OWASP AntiSamy and a very strict AntiSamy policy file to attempt to achieve this functionality.

Problem Description

Before understanding the folly behind the pompous promise of the implied claim that `isValidSafeHTML` was going to—without fail—be able to discern safe from unsafe input, we first need to understand how it has been implemented and how it interacts with AntiSamy.

The implementations of the 2.5.2.0 version of the `isValidSafeHTML` methods may be found here:

> https://github.com/ESAPI/esapi-java-legacy/blob/esapi-2.5.2.0/src/main/java/org/owasp/esapi/reference/DefaultValidator.java#L384-L405

Since there are short and very similar, I will reproduce the simpler one here for more convenient referencing:

```
     public boolean isValidSafeHTML(String context, String input, int maxLength, boolean
allowNull) {
        try {
            getValidSafeHTML( context, input, maxLength, allowNull);
            return true;
        } catch( Exception e ) {
            return false;
        }
    }
```

The following explanation is a bit of an oversimplification, but is essentially correct at a high level. The `getValidSafeHTML` method first canonicalizes ‘input’ and then passes that to AntiSamy to scan with and sanitize it with the rules found in ESAPI’s AntiSamy policy file (by default, antisamy-esapi.xml). It is important to keep in mind that while ESAPI provides a default AntiSamy policy file and most ESAPI clients just seem to use it as-is, it is *intended* to be customized to an application’s specific needs.

AntiSamy (and thus ESAPI’s `getValidSafeHTML` method) will either return the “cleansed” (i.e., scrubbed) sanitized input with its “dangerous bits”, i.e., the portions not permitted by its policy files, completely removed, or it will throw AntiSamy’s own `org.owasp.validator.html.ScanException`, which ESAPI re-throws as its own `ValidationException`. In the above method signature, the ‘context’ parameter is merely used for logging and exception messages. There are also early checks for see it the length of the ‘input’ exceeds ‘maxLength’ parameter and whether or not the ‘input’ is allowed to be null or empty as determined by the ‘allowNull’ parameter.

The problem with the `isValidSafeHTML` is method is that it returns true even when 'input' is potentially dangerous if that specific 'input' used as-is but is something that AntiSamy is able to scrub the naughty bits.

So, why is this a problem? Because `isValidSafeHTML` is a boolean and the typical use will be using it to check 'input' and if true is returned, assume that 'input' is "safe" for consumption. (Because, after all, isn't that what the method name is promising?) Except the problem is that the 'input' may not be safe at all. Rather, only the discarded return result from `getValidSafeHTML` is going to be "safe" (as defined by your possibly customized AntiSamy policy file).

At first, I thought, I naively thought I could fix this by comparing (with whitespace excluded and using `String.equalsIgnoreCase`) to compare 'input' to the cleansed value returned by `getValidSafeHTML`. (See the outlined method, `stricterIsValidSafeHTML`, described in the "Workarounds" section below for details.) However, I soon realized that turned out to be a fool's errand because whether AntiSamy throws a `ScanException` or successfully is able to cleanse the output is largely dependent on one's AntiSamy policy file, which we and the AntiSamy team encourage developers to tweak to their specific application needs. Is is for this reason that the ESAPI team decided to deprecate these two methods. We would have in fact flat out removed it in ESAPI 2.5.3.0 had that not likely broke applications using ESAPI. (And we know from experience that generally that means that many developers would choose to simply forego updating to ESAPI 2.5.3.0 had that been the case and the end result would have been even worse.) But, if you are bent on continuing to use something that cannot be guaranteed to be safe, a slightly better alternative is offered in the aforementioned "Workarounds" section.

The ESAPI team believes that both the original `Validator.isValidSafeHTML` approach as well as the stricter approach shown in the "Workarounds" section are both misguided at best, so consider yourself warned should you decide to continue using either of them.

Impact

Depending on how you are using `isValidSafeHTML`, the worst case scenario is that you check tainted input that is deemed "safe" by this function and then just use it as-is without any futher data validation and no output encoding or CSP headers so that it results in an (persistent) XSS vulnerability. However, if you are just using this method to say "yep, this is input that I should use ESAPI's safe logger to log", then you ought to be okay. YMMV.

But if you are fortunate enough for your brains not to have leaked out your ears onto the sidewalk and avoided it, kudos to you. In the past 15+ years since it has been like this in ESAPI release 1.3, thousands of developers have probably read through the `isValidSafeHTML` Javadoc or even looked at it's code and to our knowledge, no one has ever noticed this. (Or if they did, no one had ever mentioned anything to us about the potential issues.) So much for "Many eyes make all bugs shallow." 😉

Why no CVE for this issue?

The ESAPI team realizes that we may get some flak for not creating a CVE for this issue. There are several reasons why we (and mostly me [Kevin]) decided not to do so. Here are the main ones:

- Creating a CVE is a royal PITA. It is less painful now that I can leverage GitHub as the CNA and do it all through the GitHub Security Advisories, but the level of detail that I feel I owe to the FOSS community is a much higher bar for CVEs and I just don't have the time to deal with that right now. I probably also would feel compelled to try much harder and longer for a fix before announcing this and that could endanger those using this method.

- We consider this unfixable, other than giving these two methods a less pretentious name. We considered just renaming it to something more realistic like `mightThisBeValidSafeHTML` to dissuade developers from using it, but we feel the best this is to burn these 2 methods to the ground without causing an uproar from the ESAPI community.

- Realistically, we can't simply flat out remove this. It would break so much code that people would just refuse to upgrade. So, a compromise is that we deprecate these methods and add a bold CAVEAT that points people to the GitHub Security Advisory. However, if it were a CVE, it would still not be considered fixed. (Deprecation might be fine if the deprecation includes a fix, but we are not aware of any way to achieve that for the previously noted reasons discussed under the "Problem Description" section. Then once we did finally "fix" it by eventually removing it (the plan is to do so after only one year from this current release date), we would have to revise the CVE to note that it was closed in that future version...yet another PITA.

- Since it would not truly be fixed until both of the offending `isValidSafeHTML` methods are removed (see previous bullet), a probably High (or possibly Critical) CVSSv3 rating (what XSS vulnerabilities in libraries are typically rated as; typically comes out as a 7.5 or higher) would likely cause a minor panic among many cybersecurity engineers who would smack developers in the head and force them upgrade to ESAPI 2.5.3.0 [assuming we had actually "fixed" it there by removing the methods] even if those applications were not using either of these interfaces. Or if in the case where we chose not to remove but only deprecate the methods, possibly the better-staffed, more reasonable vulnerability management teams would only force application developers to go back and remove all instances of calls to the `isValidSafeHTML` methods from their code, assuming they had a way to detect its use. (This is in fact the only practical approach when a new patched version is released that is not "secure by default".)

- We think that between a GitHub Security Advisory for this (which should alert cybersecurity staff who monitor SCA tools) as well as prominent Javadoc warning, warning in the release notes, and deprecating both of the methods ought to provide sufficient warning.

Lastly, if you want to be the brave soul to take the security bulletin to some CNA and fight for it being a CVE, knock yourself out. Better the pain be in your @$$ than mine. However, since many SCA tools (at least the commercial ones) now also flag GitHub Security Advisory in addition to CVEs, there probably is little value in doing so. If you do this, we

just ask that you reference this GitHub Security Advisory GHSA-r68h-jhhj-9jvm and this ESAPI Security Bulletin in the CVE.

Once we have finally *removed* the pretentiously named `isValidSafeHTML` methods from a future ESAPI release, we may reconsider creating a CVE so that FOSS SCA tools that are not looking at GitHub Security Advisories as a data source can flag the use of this in ESAPI. We'll just have to see how it plays out.

Workaround

Stop using either of the `Validator.isValidSafeHTML`, or if you do use it, do not use the tainted input without first passing it through `Validator.getValidSafeHTML` and using its sanitized output.

Alternately, if you are really desperate, you can try an alternative that I considered and rejected. It is a stricter form of `Validator.isValidSafeHTML`. On the plus side, this should have considerably less false negatives (that is, it should reject more variations of dangerous tainted input), with the side effect of having more false positives (that is, rejecting additional safe variants). One major reason that I rejected it was a concern that ESAPI users who are using `Validator.isValidSafeHTML` would deliberately weaken their antisamy-esapi.xml AntiSamy policy files to reduce the false positives. Sadly, not breaking code usually seems to take precedent over security issues, so I am fairly confident that this will happen and hence weighed heavily in my decision to not make this approach part of the official ESAPI stance. In the end, I feel that deprecating this and then removing it after a one year period is the best way to deal with it. So, while I **don't** actually *recommend* this approach, I suggest it over continuing to use ESAPI's deprecated `Validator.isValidSafeHTML` methods. I also am aware that if I didn't offer this, there would be a lot of you who would simply continue snarfing the deprecated code and using it as-is, and I feel that is a less secure situation. Hence I offer this as a reasonable compromise, but be wary of weakening your antisamy-esapi.xml policy file simply because it now rejects things that `Validator.isValidSafeHTML` did not reject:

```
/**
 * A somewhat stricter version of ESAPI's Validator.isValidSafeHTML method.
 * It should be noted that this still cannot guarantee that the {@code input}
 * can always be considered "safe" HTML markup, but is should be better than
 * {@code Validator.isValidSafeHTML}.
 *
 * The idea behind this method (one that is still not fool-proof by the way) is
 * that after deleting all the (irrelevant) white space from both the
 * tainted input and the AntiSamy sanitized "cleansed" output, we compare
 * them via {@code String.equalsIgnoreCase} and if they are the same the
 * cleansed output and tainted input are equivalent, so is is safer than at
 * least the original ESAPI Validator.isSafeValidHTML that is now
 * deprecated.
 *
 * There are several reasons why an approach such as this was not considered
 * as an alternative to deprecated the original ESAPI method. They are
 * provided in ESAPI Security Bulletin #12 if you are interested.
 *
 */
```

```
public static boolean stricterIsValidSafeHTML(String context, String input,
                                              int maxLength, boolean allowNull) {
    try {
        //
        // If allowNull is true and input is null or empty, then null is returned for
        // the cleansed string. If allowNull is false and input is null or empty, then
        // a ValidationException is thrown. So, we need this first 'if' check to prevent
        // NullPointerExceptions.
        //
        String cleansed = ESAPI.validator().getValidSafeHTML( context, input,
                                                              maxLength, allowNull);
        if ( allowNull && input == null ) {
            return true;
        } else {
            //
            // getValidSafeHTML() (via AntiSamy) trims the cleansed results as well
            // as deleting consecutive white space, so before the comparison,
            // we simply delete ALL whitespace. I think using the Apache Commons
            // Lang StringUtils.deleteWhitespace() is probably faster than
            // using String.replaceAll() since no regex is involved, but use
            // whatever approach you like here.
            //
            String inputNoWS =
                    org.apache.commons.lang.StringUtils.deleteWhitespace( input );
            String cleansedNoWS =
                    org.apache.commons.lang.StringUtils.deleteWhitespace( cleansed );

            //
            // Note: We need to use equalsIgnoreCase() here because if the input
            // used upper case tags (e.g., <B>bold</B>), AntiSamy (at least the when`
            // the DOM parser is used), translates it to lowercase markup.
            //
            return inputNoWS.equalsIgnoreCase( cleansedNoWS );
        }
    } catch( Exception e ) {
        return false;
    }
}
```

As mentioned, even using a stricter check such as described above will not make you safe if you continue to use the tainted input as-is, but it will make you a little *safer*. Specifically, it may give you some additional time to make your application a bit more secure while you add some other defensive mechanism such as contextual output encoding or CSP headers, etc.

Solution

This section describes the high level changes made to ESAPI in version 2.5.3.0 to address this issue.

We debated whether to just flat-out to remove the code or replace it with throwing an unchecked RuntimeException that referred to the GitHub security advisory. However, since we didn't want to be responsible for flat out breaking people's production code, we decided the better course of action was to deprecate the two methods ***which we plan to remove after only a single year*** rather than the normal two year deprecation period.

Lessons Learned

Naming things, including method names, is important. Think hard about what you name things, especially "public" things that developers will use. Don't pick overly pretentious names when you can't deliver the goods.

Also, we cannot rely on Linus's law for security purposes. Several people much smarter than me created ESAPI and added the `Validator.isValidSafeHTML` interfaces and its implementation in `DefaultValidator.isValidSafeHTML` and they didn't realize the danger. We have evidence that this dates back to at least ESAPI 1.3, which seems to date back to at least September 2008. Presumably "many eyes" have read the Javadoc and likely the source code for these methods and yet no one has reported a problem with them, until now, more than 15 years later. It's time to go back to full mandatory manual security code reviews ESAPI Team!

Acknowledgments

Reviewers: Matt Seil, Jeremiah Stacey, Jon Moroney (GitHub Advanced Security team)

References

https://github.com/ESAPI/esapi-java-legacy/security/advisories/GHSA-r68h-jhhj-9jvm