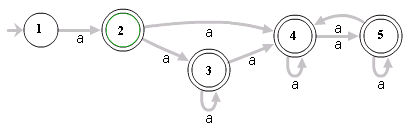
**Denial of Service: Regular Expression Development Mitigation SOP**

Denial of Service vulnerabilities occur when untrusted data is passed to the application and used as a regular expression. Regular expression evaluators and related methods can be the cause of a thread to hang when evaluating repeating and alternating overlapping of nested and repeated regex groups. Threads often hang when a regular expression implementation reaches an extreme situation that causes it to work slowly, which could allow an attacker to make the system vulnerable. This defect can be used to execute a Denial of Service attack.

If the regular expression used is ^(a+)+$, the input ‘aaaaX’ could have 16 possible paths as shown in the graph below. However, if the input is ‘*aaaaaaaaaaaaaaaaX*,’ the number of possible paths grows to 65536, causing an extreme slowdown.



**Defense Against Denial of Service: Regular Expression**

Untrusted data cannot be used as a regular expression pattern. Trusted input sources must meet criteria, such as the input being validated prior to database storage. More details about trusted input sources can be found in section 4d of the OSwA ReadMe document (resource [3] below).

**Examples**

**General Example**

public static void setUserRegex( String regex ) {

if( !StringUtils.equals(regex, userRegex) ) {

userRegex = regex;

logger.info( “setting userRegex to ” + regex );

if( StringUtils.isBlank( regex ) ) {

userRegexPattern = null;

} else {

try {

userRegexPattern = Pattern.compile(regex);

} catch (PatternSyntaxException e) {

logger.warn( “Caught ”, e );

}

}

}

}

**Explanation**

Here, the issue is in the try block where the regular expression is compiled. It is important to follow the setUserRegex function and find all usages. In some cases, the regex will come from a hardcoded regex that is not input by the user, making it a trusted source.

**Recommendations**

There are several recommendations that can be employed to protect against Denial of Service: Regex attacks, depending on the use case. If user input must be used as a regex but the input is a limited set of characters, a Java pattern can be applied to the regex in order to ensure that no user supplied evil regex characters are used. In the below example, a filterRegex pattern is created and applied to the supplied regex. The particular use case for this function is on matching names that include a first name, space, and last name.

public static void setUserRegex( String regex ) {

if( !StringUtils.equals(regex, userRegex) ) {

userRegex = regex;

logger.info( “setting userRegex to ” + regex );

if( StringUtils.isBlank( regex ) ) {

userRegexPattern = null;

} else {

try {

// p is a regex for ‘word space word’ only

String p = “^([a-zA-Z])\*(\s)\*([a-zA-Z])\*”;

filterRegex = Pattern.compile(p);

// matches throws error if evil regex

Pattern.matches(filterRegex, regex);

// Now regex can be used safely

userRegexPattern = Pattern.compile(regex);

} catch (PatternSyntaxException e) {

logger.warn( “Caught ”, e );

}

}

}

}

Another mitigation strategy is to limit the time allowed for evaluating the user supplied regex. Since the root of the Denial of Service: Regex problem is the exhaustion of resources due to exponential path searches, limiting the search to a set amount of time will get around the edge cases created by an evil regex. Java does not have a built-in thread time limitation mechanism as of time of writing, but there are third party libraries that add the functionality, such as the Aspect Oriented Programming (AOP) library JCABI.

@Timeable(limit = 5, unit = TimeUnit.SECONDS)

public static void setUserRegex( String regex ) {

if( !StringUtils.equals(regex, userRegex) ) {

userRegex = regex;

logger.info( “setting userRegex to ” + regex );

if( StringUtils.isBlank( regex ) ) {

userRegexPattern = null;

} else {

try {

userRegexPattern = Pattern.compile(regex);

}

} catch (PatternSyntaxException e) {

logger.warn( “Caught ”, e );

}

}

}

}

Finally, the last alternative is to statically define the regexes within the application. This ensures that untrusted data is not being used as a regex.

private String REGEX = “([a-zA-Z])\*(\s)\*(a-zA-Z)\*”

public static void setUserRegex() {

try {

userRegexPattern = Pattern.compile(REGEX);

}

catch (PatternSyntaxException e) {

logger.warn( “Caught ”, e );

}

}

**Example**

// A example is yet to be found

**Explanation**

An explanation of custom code goes here.

**Recommendation**

// A example is yet to be found

**Resources**

1. [HP Enterprise Security – Denial of Service: Regular Expression](http://www.hpenterprisesecurity.com/vulncat/en/vulncat/dotnet/denial_of_service_regular_expression.html)
2. [OWSAP - ReDoS](https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS)
3. [JCABI AOP Reference](http://aspects.jcabi.com/annotation-timeable.html)