Software Assurance Tips

A product of the Software Assurance Tips Team[1]

Kevin Keen

Monday 15th May, 2023

1 Checkmarx: Use of Obsolete Function

Updated Friday 12th May, 2023

One of the things that Checkmarx can scan for is *Use of Obsolete Functions*. While scanning some Java code recently, we were pleasantly surprised to find that the scanner is smart enough to look at the definition of a method and, if the implementation is marked with the `@Deprecated` annotation, it will mark the call site as a finding. This is desired behavior, and we applaud the scanner for considering that annotation. That having been said, there are several frequent false positives for which to watch. We speculate that much of the scanning for this particular finding is based on a purely text search.

Some of the false positives we saw were flagging `hashCode` as an obsolete function. This made no sense to us, as `hashCode` is an integral part of the language and has never been deprecated to our knowledge. Upon further inspection, it became apparent that the only uses of `hashCode` which were being flagged were on instances where the variable was named `identity`. There was an `Identity` class (`java.security.Identity`) which has long since been deprecated. As the whole class was deprecated, that included the `hashCode` method of that class. We presume a text search is being performed that will flag any `identity.hashCode` as deprecated regardless of whether or not the variable `identity` is actually a `java.security.Identity` object.

Another group of false positives stem from the use of `component` and `size`. These findings flag the use of the `size()` method as obsolete. It appears that methods or variables ending in `component`, which have a `size()` method called on it such as `component().size()` will trigger this finding. There is a deprecated `size` method which is a part of the awt `Component` class. We speculate that this false positive is a result of text searches that assume any `component.size()` is an awt `Component` whether it is or not.

Another class of false positives is particularly bothersome. It appears that if any version of a method carries the `@Deprecated` annotation, then all overloads of that method will be considered deprecated by this scanner. Ideally, we would like to see the scanner take into account the type / order of parameters and only flag the invocations of the method that were actually annotated as deprecated.

Lastly, we have seen uses of `toString` marked as obsolete. There did appear to be a bit more context awareness with these findings, as the variable names were allowed a wider degree of variation. The commonality in these `toString` findings was the class of the variable. In all cases they were of class `Permission`. There is an interface in Java named `Permission` which has been deprecated (`java.security.acl.Permission`). We speculate that all objects declared to be of a class named `Permission` which call `toString` will be flagged regardless of whether or not they implement the `java.security.acl.Permission` interface.

There are likely other false positives to be found. Although this scanner can provide some useful information, we hope to see it updated in the future to incorporate more type checking.

References

[1] Jon Hood, ed. SwATips. https://www.SwATips.com/.