Vulnerabilidades en el tratamiento de los datos de entrada

Una vulnerabilidad es algo que no está bien implementado o implementado de forma débil, de forma que esta debilidad pude ser aprovechada por un atacante para causar algún daño. La vulnerabilidad más habitual es que los datos de un usuario no se validen de forma adecuada. Para esto se pueden usar formularios o webs que se consultan de forma programática.

Inyección de código

En la comunicación entre aplicaciones y bases de datos suelen haber unos tokens como “;” que se pueden utilizar de forma que usando los datos del usuario se puede ejecutar una consulta en una base de datos que no se debería poder ejecutar normalmente. Esta vulnerabilidad era la más importante hasta que en 2021 paso a la tercera posición del OWASP. Existen varios tipos de inyección:

- SQL
- De Log
- Comandos de sistema operativo
- XML
- XPath

Inyección SQL

Conocida cambién como CWE-89: Improper neutralization of Special Elements used in an SQL command. Se produce cuando los datos de entrada proporcionados por el usuario utilizan algunos de los tokens de SQL de forma que no se procesan de forma adecuada, resultando en que la consulta de validación no se ejecuta de la forma esperada. Por ejemplo, una consulta de login sql puede ser:

```
String q = "SELECT * FROM users Where email='" + email + "'AND password='" +
password +"'";
```

Un ejemplo de ataque de inyección sería el siguiente:

```
"SELECT * FROM users Where email = 'bob@acme.com' OR '1' = '1' AND password
= 'any'"
```

Al meter un OR de por medio puede engañar al sistema de validación para dejar loguear.

Otro ejemplo sería este:

```
SELECT email, password, full_name
FROM users
WHERE email  = 'any' OR full_name LIKE '%Bob%';
```

También se pueden realizar ataques destructivos metiendo un DROP en la consulta.

Otras técnicas de inyección SQL intentan explotar características de algún gesto de base de datos concreto. Para ello es necesario identificar qué gestor de base de datos almacena la información a base de analizar los mensajes de error.

Otra variante es el Blind SQL Ijection que consiste en que el atacante genera consultas booleanas, de forma que con esto y un diccionario de datos se puede llegar a obtener una base de datos de usuarios completa. Estas vulnerabilidades se pueden probar en vulnweb.

Para prevenir esto, Java tiene consultas parametrizadas, también conocidas como Prepared Statements que evita la construcción de una consulta de forma manual a través de concatenación. Un ejemplo sería el sisguiente en caso de consultar datos usando un correo electrónico:

```
String query 0 "SELECT email, password, full_name FROM users WHERE email =
?";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setString(1, email);
```

```
string q = "SELECT * FROM users WHERE name = ? AND birthdate > ?";
PreparedStatement preparedStatement =
DriverManager.getConnection(url).prepareStatement(q);
preparedStatement.setString(1, "bob")
preparedStatement.setDate(2, getDate(2000,1,1));
```

En caso de usar Java Persisctence API el formateado de los parámetros de entrada se hace de forma similar.

De todas formas esta solución tiene limitaciones ya que solo se pueden parametrizar valores de columna, pero no identificadores.

por ejemplo en:

```
SELECT * FROM ? WHERE runtime > 90;
```

No se pueden usar consultas parametrizadas.

Si no se pudieran utilizar consultas parametrizadas, otra técnica son listas blancas, que acotan los valores válidos que se pueden utilizar, si los datos recibidos no están en la lista blanca, se rechaza la consulta.

En caso de que nada de esto funcione se puede usar el escapado manual. Generalmente hay librerías que pueden hacer esto de forma automatizada. Las reglas de escapado son un poco diferentes en función de la base de datos que se esté utilizando.

Inyección en ficheros de log

Los ficheros de log se usan para tener trazabilidad de los eventos que ocurran en el sistema, almacenan todos los sucesos que ocurren en un sistema. Ocurre cuando se busca falsificar (Log Forgery) los ficheros de log con entradas falsas o inyectar código javascript en el log, de forma que si

la persona que visualiza el log usa alguna herramienta de navegador se pueda producir un ataque

Por ejemplo, este código genera una entrada de log en función al usuario:

```
String val = request.fetParameter("val");
try{
  int value = integer.parseInt(val);
}catch(NumberFormatException){
  log.info("Failed to parse val =" + val);
}
```

El problema es que si el usuario introduce algo como:

```
twenty-one%0a%0aINFO:+user+logged+out%3dbadguy
```

Esto permitiría añadir más cosas al log. correspondiendo los %0a a saltos de línea y lo restante a un mensaje de log. Esto se puede paliar con librerías como log4j que permiten gestionar los mensajes de log enviados desde la aplicación. Tiene un fichero (log4j.xml) que indica que mensajes del log se pasan cuando se consulta el log o clasificar estos mensajes en ficheros diferentes en función al tipo de entrada de log.

Inyección en cabeceras HTTP

Se producen cuando se utilizan entradas de usuario incorrectamente escapadas para añadir cabeceras HTTP de forma dinámica e inesperada. Esto puede dar lugar al HTTP Response Spliting. Esta vulnerabilidad puede ser combinada con inyección en javascript. Si usáramos por ejemplo al siguiente cookie:

```
String author = request.getParametter(AUTHOR_PARAM);
Cookie cookie = new Cookie("author", author);
cookie.setMaxAge()
```

Si un usuario malintencionada mandara lo siguiente:

```
author=Wiley Hacker\r\nContent-Length:999\r\n\r\n<html> mal....
```

Produciría saltos de línea que rompen la petición, insertando contenido malicioso.

Para prevenir este tipo de ataque, si tenemos un grupo de autores pequeños, se podría hacer una lista blanca y en caso de no ser posible, se utilizaría escapados.

Inyección en SMPT

SMTP (Simple Mail Transfer Protocol) es un protocolo para el envío de correos electrónicos. Este tipo de ataque se produce cuando a través de entradas de usuario, un atacante introduce cabeceras adicionales. Se pueden prevenir con listas blancas y escapados, al igual que con las demás vulnerabilidades.

Inyección de comandos del sistema operativo

Es una vulnerabilidad que permite la ejecución arbitraria de comandos en el sistema en el que se ejecuta la aplicación. Un ejemplo de comando de inyección sería el siguiente:

```
ping -c 5 127.0.0.1; rm -rf /opt
```

Esto solo funciona si no se valida de forma adecuada las entradas.

Inyección LDAP

Se parece a la inyección SQL, se utiliza para afectar al active directory. También permite ejecutar consultas a ciegas (Blind LDAP injection) a través de filtros booleanos. Para prevenir este tipo de ataques se recomienda utilizar un framework de escape y una lista blanca.

Inyección XML y XPath

Similar a las otras inyecciones. Si en los datos de una entrada se añaden caracteres reservados en XML, el resultado puede ser una modificación inesperada del documento. Un ejemplo de inyección sería el siguiente:

```
<transaction>
  <total>300</total>
  <address>hacking street \n</address><total>50</total><address></address>
</transaction>
```

Para prevenir este tipo de ataques se crean escapados con los siguientes caracteres:

- “ = "
- ' = '
- < = <
- > = &rt;

Las reglas de escapado varían en función a donde se va a añadir el contenido. Para validar documentos XML se suelen usar esquemas (DTD), que es un documento de definición donde se indica como debe ser un documento XML. Esto permite limitar cuantas veces se repite una etiqueta y donde van, pero hay que tener cuidado ya que se pueden invalidar demasiados documentos.

Conclusiones de inyección de código

Inyección de JavaScript (XSS)

Entidades externas en documentos XML (XEE)

Deserialización y carga dinámica

Práctica de SAP

Aplicación web hecha en java en la cual aparecerna las vulnerabilidades que se están viendo. el objetivo es localizar las vulnerabilidades y solucionarlas. Para otras vulnerabilidades hay que hacer exploits. Se valorará un exploit de calidad. Grupos de 3.

From:
https://knoppia.net/ - **Knoppia**

Permanent link:
https://knoppia.net/doku.php?id=app:vemp&rev=1726763442

Last update: **2024/09/19 16:30**