|  |
| --- |
| **INFORME DE VULNERABILIDAD** |
| Ruta(s) de la aplicación involucrada(s) |
| POST /insert\_question |
| Tipo de vulnerabilidad |
| Inyección SQL |
| Causante de la vulnerabilidad |
| La función *insert\_question()* en la línea 52 causante de la vulnerabilidad, en la sección :  Se debe al formateo de cadenas donde se construye la consulta SQL, específicamente en el método *format,* los valores de las variables autor, title, tags, body se insertan directamente en la cadena de la consulta SQL sin ninguna validación previa. |
| Situaciones peligrosas o no deseadas que puede provocar |
| Manipular los datos a través de consultas UPDATE, eliminar datos con consultas DELETE, lectura de datos sensibles o tablas de base de datos a través de consultas SELECT. |
| Ejemplo paso a paso de cómo explotar la vulnerabilidad (con capturas de pantalla) |
| Ejemplo: Intentaremos borrar la tabla Questions Ejecutando la consulta: "INSERT INTO Questions(author, title, tags, body, time) VALUES ('Jose','Prueba','SQLInjection','Test', CURRENT\_TIMESTAMP); DROP TABLE Questions; --',CURRENT\_TIMESTAMP)"  Como vemos en la siguiente captura se ha ejecutado con éxito:    Nos lanza el mensaje de que la tabla de base de datos no existe: |
| Medidas para mitigar la vulnerabilidad |
| Para prevenir este fallo utilizaremos mejor sentencias preparadas, que manejan automáticamente el escape de caracteres especiales. Suprimimos también el comando executeScript, quedando el código modificado así: |

|  |
| --- |
| **INFORME DE VULNERABILIDAD** |
| Ruta(s) de la aplicación involucrada(s) |
| GET /show\_question (mostraremos la pregunta con id=2)  POST /insert\_reply |
| Tipo de vulnerabilidad |
| XSS Persistente |
| Causante de la vulnerabilidad |
| La falta de validación y sanitar los datos introducidos por el usuario, como “body” y “author”, permite que se almacenen en la BD valores que pueden tener código malicioso.  El código afectado es:  author = request.form['author']  body = request.form['body']  qbody = """INSERT INTO Replies(author,body,time,question\_id)             VALUES (:author, :body, CURRENT\_TIMESTAMP, :question\_id)"""  params = {'author': author, 'body': body, 'question\_id': question\_id}  cur.execute(qbody, params)  conn.commit()  conn.close()  Posteriormente, estos datos almacenados se recuperan e integran directamente en la respuesta HTML de la función “show\_question”, mediante la plantilla “message\_detail.html”:  return render\_template("message\_detail.html", q=question, replies=replies, ident=ident)  Debido a que no se desinfectan ni validan los datos antes de almacenarse o renderizarse, un atacante puede inyectar código HTML o JavaScript malicioso en los campos “author” y “body”. Este código malicioso se ejecutará en el navegador del usuario que visualice la página, lo que representa un XSS Persistente. |
| Situaciones peligrosas o no deseadas que puede provocar |
| Las situaciones peligrosas o no deseadas que puede provocar son:   * Robo de información sensible, como cookies o credenciales * Redirección involuntaria a sitios inseguros o fraudulentos * Engaño visual, con cambios en el diseño y contenido de la página * Phishing, al insertar formularios maliciosos que robar datos del usuario |
| Ejemplo paso a paso de cómo explotar la vulnerabilidad (con capturas de pantalla) |
| * Paso 1: Ejecutamos coladero.py y abrimos http://localhost:5000/show\_all\_questions * Paso 2: Pulsamos ver la pregunta deseada (en este caso probamos id=2)      * Paso 3: Introducimos <script>alert(‘Hacked!’)</script> (El mensaje no tiene por qué ser “Hacked!”, podemos introducir cualquier mensaje) en el campo “author” y pulsamos en contestar.      * Paso 4: Pulsar el botón Volver y accedemos a las preguntas respondidas por el usuario.      * Paso 5: Una vez hecho los pasos anteriores, cada acceso a la pregunta mostrará el mensaje introducido, en este caso “Hacked!”. |
| Medidas para mitigar la vulnerabilidad |
| 1. Escapar y desinfectar todas las entradas del usuario: Para evitar que los datos introducidos por los usuarios tengan scripts maliciosos, se debe escapar cualquier entrada que se reciba en el servidor.   En el método “insert\_reply” tenemos que introducir este cambio:  author = html.escape(request.form['author'])  body = html.escape(request.form['body'])  question\_id = html.escape(request.form['question\_id'])   1. Sanitización de los datos recuperados en la base de datos: Además de desinfectar las entradas de los usuarios, es fundamental sanitar cualquier información obtenida de la BD antes de mostrarla en la página web.   En el método “show\_question” tenemos que introducir la siguiente implementación:  question = []  for i in cur.fetchone():      question.append(html.escape(i))  cur.execute(qbody2, params)  replies = []  for rp in cur.fetchall():      ok = []      for i in rp:          ok.append(html.escape(i))      replies.append(ok)  conn.close()  return render\_template("message\_detail.html", q=question, replies=replies, ident=ident) |