Seguridad en un Sistema Operativo: XML Injection

Kevin González Sanabria
Instituto Tecnológico de Costa Rica
Ing. en Computadores
Cartago, Costa Rica
kevgonzalez@estudiantec.cr

Luis Martínez Ramírez
Instituto Tecnológico de Costa Rica
Ing. en Computadores
Cartago, Costa Rica
lumartinez@estudiantec.cr

Mariano Muñoz Masís
Instituto Tecnológico de Costa Rica
Ing. en Computadores
Cartago, Costa Rica
mmunoz@estudiantec.cr

Luis Prieto Sibaja
Instituto Tecnológico de Costa Rica
Ing. en Computadores
Cartago, Costa Rica
prieto.luisdaniel@estudiantec.cr

***Abstract*—There is a wide variety of attacks that target a web page, preventing the service from behaving properly and making it difficult for other clients to use it. Among these, the attack that concerns us in this investigation is XML Injection, more specifically the one known as the Billion Laughs Attack, which consists of introducing a portion of malicious XML code that makes many requests that paralyzes the service while it processes them. It is intended through the experiment to replicate this attack, in order to analyze its behavior, the impact it generates on the system and the different measures that exist to avoid them. To do so, a simple RESTful service was created with a minimal interface to attack, so that the service crashes and is inaccessible to clients during the time the parser tries to translate the code introduced with the injection, measuring this time and analyzing how this affects the service and the negative effect they have on real systems.**

***Index Terms*—OS, security, attack, XML, injection, billion-laughs**

I. Introducción

Las aplicaciones sobre servidores estan abiertas a muchos usuarios a nivel mundial, con usuarios haciendo solicitudes segundo a segundo, este tipo de interacción es muy común en la mayoría de paginas web, sin embargo, hay un tipo de interacción no tan común, que es multiples solicitudes desde un mismo punto, este tipo de solicitud genera una carga significativa en el servidor, que en muchas ocasiones puede generar perdidas en el rendimiento y vulnerabilidadeds, este tipo de cargas masivas han sido implementadas con fines maliciosos, con el fin de ”botar” una página y extraer información privada de la misma, estos ataques son conocidos como inyecciones al sistemas.

En esta investigación se busca replicar un ataque XML *Injection* sobre un servicio para una página web. En este caso, el ataque lo que busca específicamente es generar una cantidad de solicitudes lo suficientemente grande para paralizar el manejo de las mismas desde el servidor, lo cual impide que otros clientes puedan hacer uso del servicio. El experimento desarrollado consta de tres componentes principales: el primero es un archivo de HTML que contiene la estructura de la página web que se quiere atacar, esta muestra una sencilla lista con objetos que son ingresados por medio del teclado por parte del usuario, el segundo corresponde al servidor implementado por medio de un archivo de Python con Flask, y finalmente el último es la ”bomba”, un archivo XML que contiene el código malicioso que causa el fallo en el servicio. Se plantea encender el servidor y acceder al servicio a modo de cliente, posteriormente se cargará el archivo que contiene la bomba, esperando que como consecuencia, se vea como el servicio se interrumpe y la página web no responde.

II. Background

Un archivo XML es un tipo de archivo utilizado para guardar datos de una manera jerárquica. Este tipo de archivos es muy utilizado para enviar y recibir datos en aplicaciones de software. Dichos datos están delimitados por etiquetas determinadas por el usuario de una forma personalizada. El formato de XML es estándar, siendo compartido por diferentes plataformas y aplicaciones, de forma local y en línea, e incluso entre sistemas operativos. Esto lo consigue mediante el ”Lenguaje XML”, conocido originalmente como *”eXtensible Markup Language”*, el cual determina la sintaxis de los XML y es uno de los formatos de intercambio de información más utilizados por los sistemas computacionales en la actualidad. Cabe destacar que los archivos XML no pueden interactuar con los datos que poseen, solo los almacenan, el traducirlos e interpretarlos es la responsabilidad de otros elementos del software como lo son los *parsers*, que son traductores integrados en diferentes bibliotecas de los lenguajes de programación, para obtener y traducir los datos de un archivo de almacenamiento como XML o JSON, y aprovecharlos en las aplicaciones de software [1].

XML viene con una serie de vulnerabilidades conocidas, como XML Billion Laughs (BIL) y XML External Entities (XXE), que los atacantes maliciosos pueden explotar, comprometiendo así los sistemas SOA. Aunque estas vulnerabilidades se conocen desde hace casi dos décadas, siguen siendo muy comunes en las aplicaciones web y ocuparon el primer lugar en el top ten del Open Web Application Security Project (OWASP), junto con otras vulnerabilidades de inyección de

código. Esto se debe a que muchos desarrolladores no están debidamente capacitados en aspectos de seguridad o están bajo una intensa presión de tiempo para entregar software en un período de tiempo limitado. [2]

> "Dado el uso generalizado de los XML como archivos de intercambio de información entre sistemas, también se tienen ciertas vulnerabilidades importantes que se deben tomar en cuenta dado su uso. Un caso muy conocido de dicha vulnerabilidad es el conocido como *XML Injection*, por definicion "es una categoría de vulnerabilidades en las que una aplicación no valida correctamente la entrada del usuario antes de usarla en un documento o consulta XML. [...]Tener vulnerabilidades de inyección de XML dentro de su aplicación significa que los delincuentes tendrán rienda suelta para causar cualquier daño que puedan a sus documentos XML". [3]

Las inyecciones de XML también son una subcategoría de los ataques de inyección en general. Los delincuentes utilizan ataques de inyección para explotar las debilidades de sus aplicaciones y servicios front-end que les permiten implementar cargas útiles maliciosas y obtener acceso a sus datos confidenciales almacenados. [3]

> "La inyección de XML es una técnica de ataque que tiene como objetivo manipular la lógica de las aplicaciones o servicios basados en XML. Se lleva a cabo mediante la inyección de contenido malicioso a través de etiquetas y elementos XML en los parámetros de entrada para manipular los mensajes XML que produce el sistema, por ejemplo, para crear mensajes XML mal formados para bloquear un sistema de destino. La inyección XML también se utiliza para llevar a cabo ataques anidados (contenido malicioso incrustado en mensajes XML), por ejemplo, las cargas útiles para inyección SQL o secuencias de comandos entre sitios. El objetivo de este tipo de ataques es comprometer el propio sistema u otros sistemas que procesan los mensajes XML maliciosos, por ejemplo, una base de datos de back-end que devuelve información confidencial basada en consultas en los mensajes XML." [2]

Por tanto, un ataque de inyección XML pretende que mediante la carga de un archivo XML y mediante el uso del método XML *Bomb* o *XML Billion Laughs*(reflejado en el desarrollo del presente documento) exponer las vulnerabilidades del sistema, para así ser presa fácil de ataques computacionales.

III. Implementación

Para la implementación de este proyecto, se dividió el mismo en sus 2 componentes mayores, necesarios para la correcta verificación de su funcionalidad y el análisis de sus resultados. Estos componentes corresponden al servicio web que fue desarrollado como plataforma para el proyecto, el cual se debe de levantar antes de realizar el experimento. El otro componente corresponde al archivo XML que realizará el experimento en sí, ya que este es el archivo que contiene la entrada maliciosa que corresponde al *XML Injection* y que se espera deba realizar un ataque DoS (*Denial of Service*) en el servicio web. Se espera entonces que el servicio web se vea interrumpido de forma continua hasta que el servicio sea levantado nuevamente, o que el servicio levante una excepción durante la traducción del archivo XML y que por lo tanto se vea interrumpido.

A. Arquitectura de servicio web

Para la arquitectura del servicio web, se opta por diseñar una arquitectura muy básica de cualquier servicio web que se utilice en una aplicación web de propósito general. Esta consiste en un servicio RESTful, el cual es una arquitectura cliente-servidor simple que realiza una serie de operaciones simples mediante el consumo de un recurso enviado por el usuario, bajo un protocolo conocido, en este caso HTTP. El diagrama de componentes de este servicio se encuentra en la figura 1.

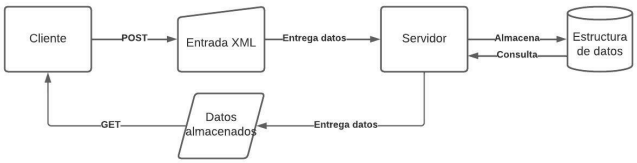

Fig. 1. Diagrama de componentes del servicio web

Como se puede observar en la figura, el servicio web utiliza el protocolo HTTP y sus operaciones estandarizadas, las cuales acceden a los recursos del servidor mediante rutas URI. Así, el cliente realiza un POST del XML, enviando su información al servidor. El servidor parsea esa información y la separa en los datos que desea consultar mediante las etiquetas del XML. Se guarda en la estructura de datos la información y se consulta por el servidor. La operación GET del protocolo HTTP realiza entonces la consulta al servidor para recuperar la información actualizada de la estructura de datos, por lo que el servidor retorna dicha información al cliente y este la despliega en su correspondiente página web.

El funcionamiento de esta arquitectura es simplista ya que es una arquitectura de propósito específico, diseñada tomando en cuenta las especificaciones del experimento realizado. Tomando en cuenta este propósito, se utilizaron los siguientes elementos para la implementación de la arquitectura:

- Servidor: El servicio que procesa las consultas y realiza operaciones dadas. Toma el tiempo que dura procesando un request de la misma forma. Escrito en Python mediante el framework Flask para desarrollo web.
- Cliente: El servicio que realiza las consultas al servidor y despliega la información en pantalla, así como el tiempo que dura procesando el request. Escrito en HTML y CSS.
- Parser: Es el que traduce el XML en su *markup language* a datos que puede utilizar el cliente-servidor para

realizar sus funciones. Escrito en Python con la biblioteca MiniDOM.

- Estructura de Datos: Consiste en un diccionario de diccionarios que guarda todos los datos provenientes de la entrada del usuario. Escrito en Python.

A nivel de página web, esta consiste simplemente en una tabla de *Markup Language* que despliega los datos del XML. Para este experimento específico se tiene un servicio web que clasifica canciones de acuerdo a su título, artista y grupo, por lo que la estructura de datos como el XML manejan esta información mediante sus etiquetas. De esta forma, tanto la página web como el XML y la estructura, despliegan la información de la canción bajo los atributos de Título, Autor y Grupo. En el caso de la página, se despliega en pantalla como se ve en la figura 2.

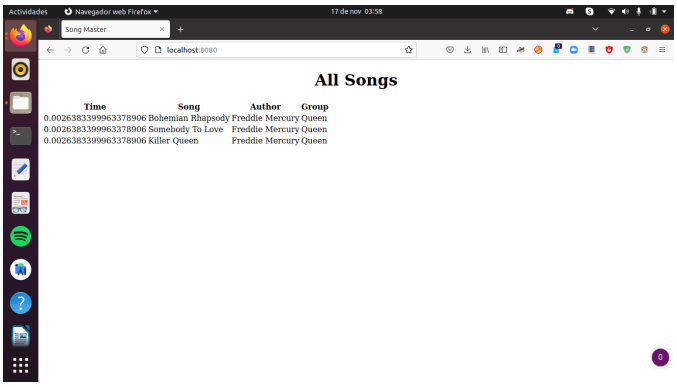

Fig. 2. Cliente utilizado en la arquitectura.

B. XML Injection

Esta sección corresponde a la porción de código que se cataloga como malicioso, ya que es este con el que se lleva a cabo el ataque. La estructura utilizada corresponde a una referencia enciclada de la famosa canción "Bohemian Rhapsody" de la banda Queen como se muestra en la figura 3.

```
<?xml version="1.0"?>
<!DOCTYPE Musica [
  <!ENTITY song "Bohemian Rhapsody">
  <!ENTITY song1 "&song;&song;&song;&song;&song;&song;">
  <!ENTITY song2 "&song1;&song1;&song1;&song1;&song1;&song1;&song1;&song1;&song1;&song1;">
  <!ENTITY song3 "&song2;&song2;&song2;&song2;&song2;&song2;&song2;&song2;&song2;&song2;">
  <!ENTITY song4 "&song3;&song3;&song3;&song3;&song3;&song3;&song3;&song3;&song3;&song3;">
  <!ENTITY song5 "&song4;&song4;&song4;&song4;&song4;&song4;&song4;&song4;&song4;&song4;">
  <!ENTITY song6 "&song5;&song5;&song5;&song5;&song5;&song5;&song5;&song5;&song5;&song5;">
  <!ENTITY song7 "&song6;&song6;">
  <!ENTITY autor "Freddie Mercury">
  <!ENTITY autor1 "&autor;&autor;&autor;&autor;&autor;&autor;">
  <!ENTITY autor2 "&autor1;&autor1;&autor1;&autor1;&autor1;&autor1;&autor1;&autor1;&autor1;&autor1;">
  <!ENTITY autor3 "&autor2;&autor2;&autor2;&autor2;&autor2;&autor2;&autor2;&autor2;&autor2;&autor2;">
  <!ENTITY autor4 "&autor3;&autor3;&autor3;&autor3;&autor3;&autor3;&autor3;&autor3;&autor3;&autor3;">
  <!ENTITY autor5 "&autor4;&autor4;&autor4;&autor4;&autor4;&autor4;&autor4;&autor4;&autor4;&autor4;">
  <!ENTITY autor6 "&autor5;&autor5;&autor5;&autor5;&autor5;&autor5;&autor5;&autor5;&autor5;&autor5;">
  <!ENTITY autor7 "&autor6;&autor6;">
  <!ENTITY group "Queen">
  <!ENTITY group1 "&group;&group;&group;&group;&group;&group;">
  <!ENTITY group2 "&group1;&group1;&group1;&group1;&group1;&group1;&group1;&group1;&group1;&group1;">
  <!ENTITY group3 "&group2;&group2;&group2;&group2;&group2;&group2;&group2;&group2;&group2;&group2;">
  <!ENTITY group4 "&group3;&group3;&group3;&group3;&group3;&group3;&group3;&group3;&group3;&group3;">
  <!ENTITY group5 "&group4;&group4;&group4;&group4;&group4;&group4;&group4;&group4;&group4;&group4;">
  <!ENTITY group6 "&group5;&group5;&group5;&group5;&group5;&group5;&group5;&group5;&group5;&group5;">
  <!ENTITY group7 "&group6;&group6;">
]>
<Musica>
  <CancionFavorita>
    <Titulo>&song7;</Titulo>
    <Autor>&autor7;</Autor>
    <Grupo>&group7;</Grupo>
  </CancionFavorita>
</Musica>
```

Fig. 3. Código utilizado para el XML Injection.

Como se explicó anteriormente el servicio web recibe un XML como entrada del usuario y lo parsea para usar sus datos. Esto implica que el servicio debe traducir el XML línea por línea. La idea principal de utilizar la *XML Bomb* como instancia de *XML Injection* es la de realizar innumerables referencias anidadas dentro del XML, por lo que el *parser* deberá recorrerlo de forma recursiva en el servicio, como se ve en la figura 3. En esta misma línea, el servicio REST que consume el XML deberá realizar sus operaciones mientras este es traducido. Por lo tanto, el *XML Injection* tiene como intención denegar el servicio de esta arquitectura web aumentando su latencia mediante el procesamiento de este *XML Bomb*. Esto puede llevar a dos posibles escenarios como se observa en general en el diagrama de flujo en la figura 4.

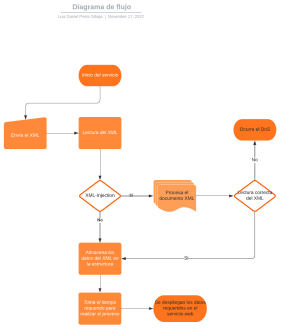

Fig. 4. Diagrama de flujo de la detección de XML-Injection.

El primer escenario es que el ataque *XML-Injection* sea exitoso en su cometido, por lo que se sobrecargará el *endpoint* del servicio web que recibe las solicitudes, realizando las operaciones necesarias para parsear el *XML Bomb* pero sufriendo una latencia importante, por lo que durante los segundos que esté inactivo realizando este proceso, efectivamente se denegará el servicio. El segundo escenario es que ocurra una excepción frente al ataque DoS, considerando que la biblioteca o servicio utilizado para la traducción del XML a un conjunto de datos que pueda entender la aplicación de software, tenga integrado un manejador de excepciones para justamente evitar el ataque DoS que realiza el *XML Bomb*. De esta forma ocurre una interrupción del servicio en lo que se levanta la excepción, el procesamiento del POST es denegado y se resume el servicio con normalidad.

De esta forma, mediante el cliente se carga un XML que contiene la referencia anidada destinada a denegar el servicio, el *parser* recibe este archivo y lo traduce normalmente, pero debe lidiar con cada una de las referencias anidadas dentro de las etiquetas del XML. Cabe destacar que se debe utilizar un archivo XML con DTD (*Document Type Definition*, o DOCTYPE) ya que se desea declarar N cantidad de entidades distintas en el encabezado de dicho documento. Estas entidades son referencias a un dato con un valor específico, el cual es incluido dentro de una etiqueta del XML. Estas entidades se pueden referenciar una a la otra, lo cual constituye la cantidad de referencias anidadas dadas y el cuerpo de la

bomba, como se observa en la figura 3. Siguiendo con el tema de musicalidad del proyecto, si busca una referencia a la entidad ”Canción 6” dentro del XML, el *parser* deberá resolver las referencias dentro de dicha entidad a las entidades anteriores, aumentando enormemente la latencia del servicio y efectivamente provocando un DoS.

Para prevenir este tipo de ataques según [4], se pueden tener 2 soluciones. En la primera se recurre a configurar el analizador XML para deshabilitar la expansión en línea de entidades. Sin la expansión en línea, el aumento del tamaño geométrico no está disponible para el atacante y estos ataques se volverán inofensivos. Cuando la aplicación requiere la expansión de la entidad, o si el analizador XML no proporciona esta opción de configuración, configure el analizador para aplicar un límite en el tamaño de las entidades expandidas.

La idea es generar una contención en el archivo de entrada para que no se expanda sin límites, y que llene la memoria del servidor. Otro método que se puede utilizar es la validación de los datos dentro del XML mediante la revisión del parseo. Así, si se tiene un archivo XML que presenta DTD (se revisa si hay un DOCTYPE), se puede mitigar completamente los efectos del *XML Bomb* simplemente denegando la solicitud de procesamiento de cualquier archivo que presente el formato DTD. A nivel de este proyecto, se considera que ambas soluciones son completamente válidas para su implementación y posibles dentro de la completitud del mismo.

IV. Resultados y Análisis

Antes de mostrar los resultados obtenidos de la simulación del ataque, en esta sección resulta importante incluir algunos datos obtenidos durante el desarrollo y prueba del experimento que, si bien no llegaron a formar parte del código como tal que se utilizó para el experimento principal, se decidió dejar como un experimento secundario debido a un comportamiento inesperado, pero que a fin de cuentas de alguna manera aportó información interesante a la investigación a pesar de conllevar con ello un rediseño del sistema.

Esta situación se presentó al instalar e incluir una biblioteca de Python llamada lxml [6], la cual se utiliza para parsear XML. Al momento de cargar el archivo XML con el código malicioso, el parser detectó las intenciones del código, por lo que bloqueó dicha ejecución. Esto implicó utilizar otra herramienta diferente para parsear XML, puesto que lxml contempla estos ataques y toma medidas de protección para evitar que tengan éxito, lo cual resulta interesante y aportó información inesperada, pero útil para efectos de la investigación. Esto confirma la segunda hipótesis sobre los resultados de la experimentación, ya que, como se pudo observar, una gran cantidad de bibliotecas de traducción y generación de XML están equipadas con un manejador de excepciones para evitar ataques DoS por *XML-Injection*. Como se esperaba mediante el análisis de la teoría, la primera biblioteca contemplada para el proyecto, lxml, cumple con este requisito, puesto que al detectar las referencias anidadas del archivo ”Queen Bomb”, este no permite que dicho archivo sea procesado, levantando una excepción e interrumpiendo el servicio. De esta foma, el segundo escenario contemplado se prueba correcto, sin embargo, para efectos del experimento, se debe cambiar de biblioteca a XML MiniDOM para obtener resultados contundentes con su respectivo análisis.

Una vez realizado el experimento, los resultados se pueden visualizar en la tabla I, los cuales muestran un aumento exponencial en el tiempo de respuesta del servidor, de manera que el servidor en algún momento debe detener el servicio dado que se quedó atorado con una sola solicitud.

Por entidades se entienda cada linea de la manera:
<!ENTITY song1 ”&song;&song;&song;&song;&song;&song;”>
Lo que a su vez define que cada *song1*, va a estar compuesta de 6 entidades *song*, este despliegue que hace XML de manera recursiva sobre las entidades es lo que geneara en el archivo una carga inusualmente grande de información.

Para 6 líneas de entidades, por solicitud el servidor consume 0.0052 segundos, para un total de 8.70 segundos, para 7 cada solicitud 0.0025 para un total de 27.85 segundos y 8 líneas de entidades para 0.0067 segundos para un total de 93.90 segundos en responder.

De estos valores lo que se puede evidenciar es que un solo archivo, puede generar una carga realmente significativa, en comparación con los estándares normales de 300 ms de respuesta en servidores, y aún tomando en cuenta que la aplicación no es relativamente complicada.

Tomando como base estos puntos recién mencionados, es que se puede observar que el panorama presenciado durante la ejecución del ataque y posterior análisis de los resultados obtenidos coincide con lo planteado dentro de la primera hipótesis, donde el servicio va a ser interrumpido durante el lapso que se procesan las *requests* introducidas por el código malicioso, el cual toma más tiempo en ejecutarse conforme se aumenta la cantidad de entidades en la bomba, acorde a lo mencionado en el marco teórico con respecto a la interrupción del servicio mientras se trata la solicitud.

Tiempo de ejecución de request (s)	**Tiempo de ejecución de bomba (s)**	**Cantidad de entidades en bomba**
0.005204	8.705336	6
0.002581	27.85432	7
0.006757	93.90468	8

TABLE I
Tiempo de respuesta a la XML Bomb

V. Conclusiones

Como motivo central, basándose en evidencia de casos a través de la historia así como en la investigación realizada y su correspondiente experimento, se puede decir a ciencia cierta que los ataques a sistemas informáticos se dan debido a que estos sistemas lo permiten, pues muestran algún tipo de vulnerabilidad que alguien logra detectar y aprovechar a su favor, para entorpecer el funcionamiento del servicio. Estas vulnerabilidades no se suelen dejar adrede, pues la mayor parte del tiempo estas se descubren unas vez se experimenta un ataque, sin embargo, si que la seguridad recae bastante en los desarrolladores, pues deben intentar prever situaciones de

ataques conocidos e implementar el código de manera que pueda reaccionar ante estos en caso de que se presente alguno. Por otro lado, aunque es difícil saber el enfoque de nuevos ataques o las herramientas que se usen para ello, no se deben dejar portillos abiertos que faciliten la manipulación del código por medio de software malicioso.

También es importante mencionar el uso de casos de prueba en conjunto con automatizaciones de sistema, este método expone de cierta manera en etapas previas a los lanzamientos distintos tipos de vulnerabilidades, desde el punto de vista de un desarrollador se debe considerar escenarios que pueden parecer ficticios, pero que pueden generar en el futuro grandes problemas, casos como cantidad de caracteres en una entrada de una página web, el tamaño de un archivo en la carga de información, agregar o eliminar campos en archivos de llenado etc. No se debe aislar el comportamiento de una aplicación a los casos esperados, prever este tipo de situaciones también es responsabilidad del desarrollador, y no esperar que este tipo de vulnerabilidades sean expuestas por terceros.

En cuanto al análisis de resultados del experimento, el comportamiento obtenido fue el esperado, puesto que al incrementar la cantidad de entidades que se quieren procesar dentro del código malicioso, esto se ve reflejado directamente en el tiempo que el servicio se mantiene caído, aunque si bien su comportamiento se esperaba que fuera lineal, resultó ser exponencial.

VI. Sugerencias y Recomendaciones

- Dadas las situaciones sucedidas que se detallaron en la sección de Resultados y Análisis, se recomienda que al desarrollar sistemas que involucren parseo de archivos XML, se investigue de primera mano cuáles bibliotecas y herramientas cuentan con algún tipo de seguridad o manejo que permita detectar cuándo se quiere introducir código malicioso, esto por supuesto sin dejar de lado las medidas extra que se puedan aportar como desarrollador.
- Se debe tener en cuenta que en las páginas web, los usuarios pueden ingresar información a modo de texto solo en los espacios donde los desarrolladores les permitan hacerlo, por ejemplo, formularios. Es una buena práctica limitar el tipo de información que se puede ingresar en estos espacios, por ejemplo, si lo que se quiere obtener de un formulario es un número telefónico, limitar la entrada de texto para que únicamente reciba caracteres con valores entre 0-9, o si se quiere ingresar un nombre, que se habiliten solo caracteres con valores entre A y Z. Esto evita situaciones en los que se puedan ingresar caracteres erróneos que puedan desencadenar problemas dentro del código.
- Todo proceso de desarrollo de software debe involucrar etapas de evaluación y pruebas, de modo que se asegure el correcto funcionamiento de las diferentes *features*, sin embargo, es importante que en estas etapas también se dedique tiempo y recursos para revisar vulnerabilidades y simular ataques conocidos, como es el caso del ataque analizado en esta investigación, de esta forma, se pueden encontrar fallos o debilidades antes de que se pase el producto a producción, evitando posibles ataques a futuro y con ello no solo asegurar la integridad del sistema, sino también evitar pérdidas económicas, lo cual siempre resulta de gran importancia tener en cuenta.
- Como complemento para el punto anterior, se pueden utilizar herramientas que se encarguen de evaluar el código y automatizar pruebas para detectar vulnerabilidades ante ataques, como es el caso por ejemplo de SonarQube [7].

References

[1] Juvile, J. (2021). *XML Files: What They Are How to Open Them.* Hubspot. https://blog.hubspot.com/website/what-is-xml-file

[2] Jan, S., Panichella, A., Arcuri, A., Briand, L. (2017). "Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications". IEEE Transactions on Software Engineering, https://ieeexplore.ieee.org/document/8125155

[3] Crane C. (Mayo 18, 2022) " XML Injection Attacks: What to Know About XPath, XQuery, XXE More." Hashedout. https://www.thesslstore.com/blog/xml-injection-attacks-what-to-know-about-xpath-xquery-xxe-more/

[4] Kohnfelder L., Heymann E., Miller B. (2022) "Introduction to Software Security" Chapter 3.8.4: XML Injection Attacks. https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Chapters/$3_{84}-XML-Injections.pdf$

[5] "Learn Flask for Python - Full Tutorial". YouTube. https://www.youtube.com/watch?v=Z1RJmh$_{OqeAt} = 903sab_channel = freeCodeCamp.org(accessedNov.13, 2022)$.

[6] "lxml - Processing XML and HTML with Python." lxml. https://lxml.de/

[7] "Static Application Security Testing." SonarQube https://www.sonarqube.org/features/security/sast (accessed Nov. 16, 2022).