— José Rodgo Monro López.

8,75

Fundamentos y Técnicas de Seguridad para Aplicaciones

Evaluación de fin de cursado

1. *Gestión de Dependencias como Problema de Seguridad*

2 puntos

Analiza por qué la gestión de dependencias de software (ej. en NPM, PyPI, apt) se ha convertido en un vector crítico de ataques a la cadena de suministro. Propón dos prácticas, una técnica y una organizacional, para concientizar o reducir este riesgo.

2. *Antipatrones de seguridad*

1 punto

¿Cuáles de los siguientes son *antipatrones de seguridad* reconocidos? Indica cuáles son, e indica qué correspondería hacer si encuentras código con este *antipatrón* en un sistema que estés trabajando.

a) "Security by Obscurity" (Seguridad por Oscuridad).
b) Almacenar contraseñas en texto plano en la base de datos.
c) Implementar el principio de mínimo privilegio para los servicios.
d) Realizar validación de entrada únicamente del lado del cliente.

3. *Desbordamientos*

2 puntos

Si bien los desbordamientos más famosos (¡y por buenas razones!) son los desbordamientos de pila (*stack overflows*), la cantidad de amenazas que pueden caracterizarse de esta manera es mucho mayor. Compara la explotación de un *desbordamiento de pila* (*stack overflow*) con un *desbordamiento de enteros* (*integer overflow*).

4. *Enumeración de debilidades, enumeración de vulnerabilidades*

2 puntos

Dos de los índices mas importantes para la comprensión y el seguimiento de la seguridad en aplicaciones son el **CWE** (*Common Weakness Enumeration*) y el **CVE** (*Common Vulnerabilities Enumeration*). ¿Cuál es la relación entre ambos? ¿Existe algún *camino* que lleve a algún *problema* que descubramos en determinado código a *transitar* de clasificarse como CWE a clasificarse como CVE (o a la inversa)?

5. *Ataques de inyección*

2 puntos

Elige un ataque de inyección específico (SQLi, des-serialización, SSRF, etc.) y propón una estrategia de "defensa en profundidad", ubicando controles de seguridad distintos en diferentes capas (ej. aplicación, red, SO) para mitigarlo.

6. *Tipos de inyección*

1 punto

Relaciona cada concepto (*tipo de inyección*) con su descripción correspondiente.

Tipo de inyección	**Descripción**
a) Inyección de comandos	a) La aplicación des-serializa datos no confiables, instanciando objetos o invocando métodos arbitrarios
b) Inyección de objetos	b) Datos no validados se envían a un intérprete de comandos del sistema operativo
c) Server-Side Request Forgery (SSRF)	c) Entradas de usuario se incorporan directamente en una consulta de base de datos, alterando su lógica
d) Inyección SQL	d) La aplicación es engañada para realizar consultas a un destino interno o restringido

José Rocky Manco López

Problema 1

El motivo es que la mayoría de dependencias existentes y de uso masivo hoy en día como NPM, PyPI etc sus paquetes dependen de muchísimos paquetes terceros, lo cual hace que las dependencias exploten en términos de # de paquetes de 3rd's usados en un solo paquete. Eso lo hace altamente vulnerable por el simple hecho de que es muy poco probable que todos los paquetes de bibliotecas sigan las mejores prácticas → si cae una es muy probable que caigan todos. Dos propuestas son: Técnica: analizar y dejar mínimas dependencias así como adoptar herramientas tipo SonarQube.

¿En qué medidas técnicas podrías asegurar mínimas pérdidas?

Organizacional: tener una metodología Octave para reconocer riesgos de accesos de software.

La conciencia es fundamental como medida organizacional pero... para ver es suficiente. ¡Ocupamos medidas fuertes!

Problema 2 (a) Anti patrón de asumir que un atacante no conoce tu sistema y que es imposible por ejemplo que repita que encriptador usar. Mejorar por ejemplo en vez de usar algs. de encriptado poco seguros a usar un SHA.

(b) Básicamente si se logra acceder a tu BD se tiene conocimiento de todas las PWD por eso es un antipatrón. Lo mejor es encriptar la PWD en la BD con algún algoritmo y matchear en una app con ese algoritmo de encriptado. (Anti patrón)

¡No se dice "encriptar" se dice "cifrar"! Y... No guardes contraseñas cifradas. Guarda el hash de las contraseñas (¡No es lo mismo!) :)

(c) Esto es un patrón de buenas prácticas. Básicamente consiste en hundir acceso a los usuarios mínimo que necesitan para su rol.

(d) Esto es un patrón de buenas prácticas. centralizar y no permitir llegar a tu dominio es una buena práctica.

Pero hay que realizar las validaciones del lado del servidor. Del lado del cliente pueden fallar (o ser omitidas intencionalmente por un atacante), y recibes datos malos :(

Problema 3 La diferencia principal consiste en que el stack overflow → controlar o forzar → permite ~~predecir~~? o ~~entender~~? cual es la siguiente instrucción a ejecutar lo cual permite la ejecución de código malicioso.

Un integer overflow le da la vuelta a un valor de modo que $2^{31}+1 = -2^{31}$, al otro mientras que el integer Overflow permite alterar o ingresar ~~código malicioso en el heap~~. De esta forma uno complementa

Mientras que el integer Overflow permite alterar o ingresar código malicioso (int overflow) el otro puede ejecutarlo) (stack Overflow)

~~Problema 4~~ Aparte de que uno se da al atacar el stack de ejecución como un ciclo while infinito (que si no hay algo que lo detenga satura la memoria de Acceso rápido) el otro lo guarda en

Me parece que comprendes al buffer overflow en stack. El integer overflow es un mal bicho muy diferente!

Problema 4

CWE :: es un repositorio (index) de prácticas que se consideran como "muy riesgosas" como SQLi en este evaluare encontramos patrones genéricos.

CVE :: es un repositorio (index) de fallas o problemas de seguridad existentes en productos actualmente usados como "riesgo en la librería ~~x~~ de la distribución y de Ubuntu". ¡biblioteca!

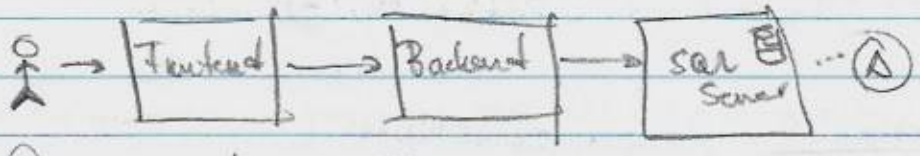
Relación sii { CVE nos ayuda a poder detectar patrones para generar CWE, también CWE puede ser útil para hacer CVE en productos.

El párrafo anterior también responde a la tercer pregunta ya que siento las bases para pasar de CVE a CWE y viceversa.

Cada vez que le dices "librería" a una "biblioteca", Dios mata a un gatito.
Por favor... ¡Piensa en los gatitos! =^.^=

José Rodrigo Moreno López.

Problema 5 SQLi. Recordemos que la defensa en profundidad consiste en detectar todas las capas y añadir a cada una de ellas todas las posibles validaciones para evitar en este caso un SQLi. Supongamos una arquitectura como la rig:



Queremos evitar un SQLi:

CAPA 1

① En el Frontend se valida por ejemplo en el servicio de autenticación que
username = "< una sola palabra>"
o validar con regex la estructura de email.
pwd = dominio validar la estructura propia de pwd.

CAPA 2

② En el Backend en vez de tener algo como:
s = "select * from user where user='@user' and pwd='@pwd'"
Esto podría atacar un SQLi pues si user = admin" or "1=1" --
=> se hypasa la seguridad y ∴ libre acceso, la pregunta es user ① un encode del pwd → evita espacios pwd intentales en la BD.

② user prepared statements.
s = PreparedStatement("select .. user=? and pwd=?")
s.setString(1, user)
s.setString(2, pwd)