Semana 1: Seguridad y Vulnerabilidad en Software

Seguridad y Safety:

- **Safety:** Enfocado en prevenir accidentes.
- **Security:** Relacionado con la protección contra ataques intencionales.

Concepto de Vulnerabilidad:
Una vulnerabilidad es una falla en el diseño, implementación u operación de un sistema que puede explotarse para comprometer la seguridad. Existen tres tipos de vulnerabilidades:

1. **Diseño o especificación:** La seguridad debe ser un proceso desde la fase de requerimientos.
2. **Implementación:** Área de mayor enfoque en seguridad.
3. **Operación y gestión:** Relacionado con el manejo adecuado del sistema en funcionamiento.

Éxito de un Ataque:
El éxito depende del grado de vulnerabilidad, la fuerza del ataque y la efectividad de las contramedidas.

Principios de Seguridad en Software:
Los componentes fundamentales de la seguridad son:

- **C:** Confidencialidad: Si yo no tengo autorización, no tendré permisos de lectura.
- **I:** Integridad: No escribir información
- **A:** Disponibilidad: Siempre disponible
- **No repudio:** Registro y responsabilidad de acciones.

La seguridad busca crear software que funcione bajo ataque, balanceando funcionalidad y protección. Para asegurar la seguridad, deben considerarse los participantes, activos, amenazas y el modelo del atacante.

Software de Seguridad vs. Seguridad en el Desarrollo de Software:

- **Software de seguridad:** Herramientas como firewalls, EDR, antivirus, etc.
- **Seguridad en el desarrollo:** Minimizar vulnerabilidades durante la programación.

Práctica en Seguridad:
Usualmente, la seguridad es secundaria frente a la funcionalidad y el tiempo. La seguridad requiere considerar los posibles comportamientos no deseados del software.

Ejemplos en Funcionalidad vs. Seguridad:

- **Sistemas Operativos:** Windows es funcional; OpenBSD es altamente seguro.
- **Lenguajes de Programación:** C es funcional pero menos seguro; Rust y Go priorizan la seguridad.
- **Navegadores:** Chrome enfocado en funcionalidad, Lynx en seguridad.

Importancia de CIAR en Diferentes Contextos:

- **Cuentas bancarias:** Confidencialidad es clave para el usuario; integridad y disponibilidad son críticas para el banco.
- **Expedientes médicos:** Confidencialidad, integridad y disponibilidad son altamente importantes.
- **Planta eléctrica:** La disponibilidad es el factor más crítico.

Historia de la Seguridad en Software:
Desde el gusano Creeper en 1971 hasta los ataques modernos, el malware ha evolucionado de un reto técnico a una herramienta de lucro. En los 2000, la seguridad se estableció como disciplina clave, influenciada por amenazas como el gusano Nimda y Code Red. Hoy, la seguridad en software es esencial debido a su omnipresencia en la vida cotidiana.

Resumen Semana 2: Problemas de Seguridad en el Software y Taxonomías de Debilidades

Atributos que Generan Problemas de Seguridad en el Software:

1. **Complejidad:** Aumenta la probabilidad de errores con más líneas de código y dificultad de comprensión.
2. **Extensibilidad:** Los programas que admiten extensiones o plugins pueden introducir vulnerabilidades.
3. **Conectividad:** La conexión a redes expone el software a una mayor cantidad de amenazas externas.

Problemas Comunes de Seguridad:

- Falta de conciencia y conocimiento en seguridad.
- Enfoque en funcionalidad sobre seguridad, aumentando el riesgo de vulnerabilidades.

Common Weakness Enumeration (CWE): El CWE es una taxonomía que categoriza y clasifica las debilidades comunes en software, permitiendo a la comunidad de seguridad compartir un vocabulario común para identificar, mitigar y prevenir debilidades en el software. Está estructurado en tres niveles:

1. **Category:** Nivel más general, agrupando múltiples bases.
2. **Base:** Debilidades independientes de un recurso específico.
3. **Variant:** Debilidades específicas de un producto particular.

Concepto de Debilidad: Una debilidad es una condición en el software, firmware, hardware o servicios que, bajo ciertas circunstancias, podría contribuir a introducir vulnerabilidades. No todas las debilidades son vulnerabilidades, pero podrían volverse explotables bajo ciertos factores.

Problemas comunes en programación:

Off by one: Este error ocurre cuando se realiza un cálculo con un índice o límite que está desviado en una unidad.

División entre cero: Esta operación matemática no está definida y puede causar una excepción o un comportamiento inesperado en el programa..

Predicción de números aleatorios: Aunque los generadores de números aleatorios están diseñados para producir secuencias impredecibles, algunos algoritmos pueden ser predecibles bajo ciertas condiciones.

CWSS (common weakness scoring system)

Los tres grupos de criterios del CWSS son:

Base finding: Evalúa las características intrínsecas de la vulnerabilidad, como la complejidad de la explotación y los requisitos de acceso.

Attack surface: Considera factores temporales, como la disponibilidad de exploits y la existencia de parches.

Environmental: Tiene en cuenta el contexto específico de la organización afectada, como la importancia de los activos y las medidas de seguridad existentes.

CWE

- Es un lenguaje común para el progreso de la disciplina del software
- No todas las vulnerabilidades son explotables
- Priorizar las debilidades mediante CWSS

Common Attack Pattern Enumerations and Classifications (CAPEC)

CAPEC brinda una clasificación detallada de los patrones de ataque utilizados. Al comprender estos patrones, los defensores pueden desarrollar estrategias más efectivas para prevenir y detectar ataques.

Un patrón de ataque es una forma común de explotar una debilidad del software, hardware entre otros.

CAPEC tiene una serie de pasos de cómo realizar un ataque para explotar la debilidad.

Execution flow: Es como una serie de pasos y técnicas que se pueden hacer para realizar este ataque.

Esta página tiene skill level, prerequisites, mitigciones, ejemplos y las debilidades relacionadas.

Está organizado en una jerarquía

- **Category**
- **Meta**: Un abstracto detallado de un método específico
- **Standard**: Es enfocado ya en una metodología específica o técnica para realizar el ataque.
- **Details**: Se da con gran detalle que se necesita para completar.

Una taxonomía de patrones de ataque es útil porque nos permite ver la perspectiva y los enfoques de los atacantes

Cyber threat intelligence (CTI): Tiene un papel fundamental en la protección contra las ciberamenazas. Al analizar la información sobre grupos de atacantes. Basado en cierta información como:

- Grupo y patrones de actividad
- Reconocer la actividad criminal
- Identificar e implementar las estrategias de mitigación.

Las debilidades y patrones de ataque se combinan.

Tríada mágica de la seguridad del software es

- **CWE:** Todas las debilidades de software
- **CAPEC**: Patrones de ataque en base a las vulnerabilidades
- **CVE**: Vulnerabilidades que se generan cuando el patrón de ataque se vuelve exitoso (es el único que no es una taxonomía)

VBD vulnerability database.

CVSS es el framework donde viene la severidad de las vulnerabilidades pero no hay una definición clara y formal sobre qué tan peligroso sea. Esta métrica también tiene categorías y fórmulas que combinan todo.

Resumen Semana 5 Buffer Overflow y Programación a Bajo Nivel

¿Qué es un Buffer Overflow?

Un *buffer overflow* ocurre cuando un programa escribe más datos en un buffer de lo que este puede almacenar, lo que puede sobrescribir áreas incorrectas de la memoria, introduciendo vulnerabilidades de seguridad. En lenguajes como C y C++, esta falta de verificación de límites permite que se escriba fuera de los límites del buffer, corrompiendo potencialmente datos importantes en la memoria.

Lenguajes modernos como Java y Python verifican los límites, a diferencia de C/C++, que priorizan la eficiencia y permiten estos accesos peligrosos.

Problemas Comunes en el Manejo de Memoria en C

- **malloc y free**: El manejo manual de la memoria es propenso a errores como *use-after-free*.
- **Desbordamiento de enteros**: Exceder el tamaño asignado a un número.
- **Ataques de formato**: Debido al tipado débil de C, permite manipulaciones riesgosas de la memoria.
- **Punteros salvajes y variables no inicializadas**: Apuntan a ubicaciones de memoria indefinidas, causando errores.
- **Fugas de memoria**: Ocurren cuando la memoria asignada no se libera.
- **Exhaustión del stack y heap**: Al llenar la pila o agotar la memoria heap.
- **Double free**: Intentar liberar una misma porción de memoria más de una vez.

Enlazado de Programas

- **Enlazado estático**: Copia el código de la biblioteca en el ejecutable, haciéndolo autocontenido. Tiene la ventaja de independencia, rendimiento y consistencia.
- **Enlazado dinámico**: Incluye solo las referencias a las funciones de la biblioteca, lo que ahorra espacio pero aumenta la fragilidad. Ventaja de tamaño de archivo, Actualización sencilla y eficiencia de memoria al reducir los recursos utilizados.

Ingeniería Reversa

La **ingeniería inversa** consiste en descomponer un programa para entender su funcionamiento sin tener acceso al código fuente, útil en análisis de malware como el caso de **WannaCry**, donde el malware se desactivaba en entornos virtuales. Existen dos enfoques principales:

1. **Análisis Dinámico:** Ejecuta el programa para observar su comportamiento en tiempo real, útil cuando el código está cifrado. Herramientas comunes:
    - **GDB + peda** para observar registros y pila.
    - **Fuzzing** envía datos aleatorios para detectar fallos.
    - **Angr** para ejecución simbólica.
    - **IDA Pro** (versión gratuita limitada) como herramienta de análisis.
2. **Análisis Estático:** Se analiza el binario sin ejecutarlo, desensamblando o decompilando para entender su lógica. Herramientas útiles:
    - **Ghidra** y **Radare2** para decompilar o entender el binario.

Diferencias entre descompilación y desensamblado: Desensamblar convierte el código binario a bajo nivel (ensamblador), mientras que descompilar lo convierte a un lenguaje de alto nivel como C.

Java Modeling Language (JML) y **OpenJML:** JML permite definir especificaciones en código Java como precondiciones, postcondiciones e invariantes. Herramientas como OpenJML verifican automáticamente si el código cumple con estas condiciones.

- **Precondiciones** y **postcondiciones** aseguran que los métodos cumplen ciertos criterios al ejecutarse. Requires y ensures
- **Assertions** son verificaciones de valores específicos en ciertos puntos del código. assert
- **Invariantes** en el contexto de valores (ej. centavos entre -99 y 99).

Para usar OpenJML, se recomienda empezar con los constructores y luego descomentar gradualmente. Las especificaciones deben estar documentadas en el encabezado de cada método.

Semana 12 Compiladores

Compiladores y sus Componentes:

Un compilador tiene cuatro partes principales:

1. **Frontend:** Realiza el análisis léxico y sintáctico para interpretar las instrucciones del código fuente. Es útil para encontrar errores en el código y prepara el código de entrada para el optimizador.
2. **Intermediate Representation (IR):** Genera una representación intermedia que es más fácil de optimizar y transformar.
3. **Optimizer:** Optimiza el código para mejorar su eficiencia, tanto en velocidad como en uso de memoria.
4. **Backend:** Genera el código específico para el procesador de destino, adaptándolo a la arquitectura específica del CPU. Tiene las siguientes partes
    a. Selección de instrucciones: Elige instrucciones para transformar el código al modelo específico.
    b. Register allocation: Compilador decide cuáles datos se guardaran en los registros del CPU optimizando los recursos disponibles.
    c. Instruction scheduling: Ordena instrucciones para ejecución paralela.

Optimización:

La optimización del código puede transformar el programa para hacerlo más eficiente, aunque esto puede complicar su reversión. Ejemplos:

- **Desenrollado de bucles:** Transforma un bucle `while` en múltiples repeticiones de su contenido para eliminar saltos y variables, mejorando la velocidad a costa del tamaño del código.
- **Eliminación de redundancias:** Si un valor como `foo = "hola"` no cambia en un bucle, el optimizador lo coloca fuera del bucle para evitar asignaciones repetitivas.

Ejecución en Entornos de Máquina Virtual (VM):

Las máquinas virtuales permiten la portabilidad al proporcionar una interfaz constante para diferentes infraestructuras, facilitando la ejecución en distintas plataformas. Además, ofrecen beneficios como **garbage collection, portabilidad** y **type safety**. Por otro lado tiene las desventaja de reducir el rendimiento

Ejecución Simbólica y Concolica:

La **ejecución simbólica** es una técnica de análisis dinámico que explora múltiples entradas en el programa siguiendo distintas rutas. No genera falsos positivos en la detección de errores, lo cual es ventajoso. Herramientas como **Angr** permiten esta ejecución, modelando el flujo del programa mediante diagramas de decisiones.

- **Ejemplo de Ejecución Simbólica:** En un código con varias condiciones, la ejecución simbólica rastrea todas las posibles rutas para encontrar ramas que activen funciones o banderas especiales sin hacer fuerza bruta.
- **Ejecución Concolica:** Cuando el programa tiene estructuras complejas como bucles o condiciones anidadas, se utiliza una combinación de ejecución simbólica con fuerza bruta. Esto permite gestionar la complejidad y encontrar rutas específicas en códigos muy complejos.