# Resumen Teórica 15 : Desarrollo seguro - Parte 2

Tomás F. Melli

October 2025

# Índice

# 1 Principios

## 1.1 Comenzar haciendo preguntas

- Sobre lo que nos preocupa:
  - ¿Qué puede ir mal?
  - ¿Qué estamos tratando de proteger?
  - ¿Qué es lo que pensamos que va a afectar a nuestra seguridad?
  - ¿Cuál es el punto débil de nuestras defensas?
- Sobre nuestros recursos:
  - ¿Tenemos una arquitectura de seguridad? ¿Se usa?
  - ¿Tenemos acceso a librerías de código reusable?
  - ¿Qué estándares y guías tenemos disponibles?
  - ¿Qué buenos ejemplos nos pueden ayudar?
- Sobre el software:
  - ¿En qué parte de la cadena de confianza se encuentra?
  - ¿Quiénes son los usuarios legítimos?
  - ¿Quién accederá al software (exec/source)?
  - ¿El uso y el número de usuarios cambia en el tiempo?
  - ¿En qué entorno se ejecuta?
- Sobre nuestras metas u objetivos:
  - ¿Qué impacto tendrá un problema de seguridad?
  - ¿A quién vamos a molestar con las medidas de seguridad?
  - ¿Tenemos el apoyo de los altos niveles de la organización?
  - Si los usuarios eluden nuestras medidas de seguridad, ¿cómo lo vamos a saber y cómo lo vamos a manejar?

## 1.2 Elegir un destino antes de comenzar

Antes de tomar decisiones se debe entender bien lo que se necesita hacer. Ejemplo: analogía con la construcción de casas. Los arquitectos antes de dibujar los planos primero deben saber:

- Tipo de edificación.
- Lugar de construcción (ej. ¿hay terremotos?).
- Regulación sobre construcciones del lugar.
- Necesidades del cliente.

## 1.3 Decidir cuánta seguridad es suficiente

El nivel de seguridad depende de:

- Tamaño y naturaleza de los riesgos.
- Costo de las medidas de seguridad.

Ejemplo: no hacer aplicaciones tan seguras como sea posible, sino suficientemente seguras. El nivel aceptable de seguridad se debe determinar objetivamente, considerando standards cuando existan (sector financiero, etc.).

## 1.4 Emplear técnicas standard de ingeniería

- Buena seguridad requiere buen diseño de software y buenas técnicas de ingeniería.
- Factores principales para ataques: falta de diseño, debilidad humana, prácticas de codificación pobres.
- Una buena arquitectura de seguridad no elimina el último punto.

### 1.5 Identificar lo que se asume

Ejemplos:

- Paquetes TCP con SYN flag activado: se asume que no hay mala intención.
- Usuarios humanos: puede que sea un script ejecutado muchas veces.
- Solución común: CAPTCHA.

### 1.6 Incluir la seguridad desde el primer día

Evitar agregar controles de seguridad tarde. Ejemplo: agregar cifrado de datos a posteriori podría no ser suficiente.

### 1.7 Diseñar con el enemigo en la mente

Anticipar cómo un atacante podría resolver el rompecabezas de nuestra infraestructura.

### 1.8 Comprender y respetar la cadena de confianza

- No invocar programas no confiables desde otros confiables.
- No delegar autoridad para ejecutar acciones sin delegar responsabilidad de chequearlas.

### 1.9 Ser tacaño con los privilegios

Principio del menor privilegio: un programa debe operar solo con los privilegios necesarios. Ejemplo: leer un archivo sin abrirlo con permisos de escritura innecesarios.

### 1.10 Testear todas las acciones propuestas contra la política

- Asegurar que las decisiones del software cumplen políticas de seguridad.
- Ejemplo: agregar productos al carro de compras solo si pertenecen al usuario y verificar sesión activa.

### 1.11 Construir niveles apropiados de tolerancia a fallas

Identificar funcionalidades críticas y aplicar las tres Rs:

- Resistencia: capacidad de disuadir ataques.
- Reconocimiento: capacidad de reconocer ataques y daños.
- Recuperación: mantener servicios esenciales durante ataques y restaurar tras el incidente.

### 1.12 Tratar los temas de manejo de errores de manera apropiada

- Arquitecto: define plan general de manejo de errores.
- Diseñador: define detección, discriminación y respuesta a errores.
- Programador: captura condiciones de error según diseño.
- Operador: controla procesos y revisa logs.

### 1.13 "Degrade Gracefully"

La aplicación debe operar de manera restringida o degradar funcionalidad ante problemas. Ejemplos: límite de conexiones para SYN flood, zonas de absorción de impacto en automóviles.

### 1.14 Fallar de manera segura

Ejemplos: firewall que detiene tráfico ante fallo, semáforos o respiradores que fallan de manera segura.

### 1.15 Elegir acciones y valores por defecto seguros

Ejemplo: asumir que un usuario no tiene autorización hasta comprobarlo.

### 1.16 Mantener las cosas simples y modularizar

- Definir interfaces claras entre módulos.
- Limitar privilegios y recursos a lo estrictamente necesario.

### 1.17 Modularizar

Para tener éxito hay que:

- Definir de manera adecuada las interfaces entre los módulos.
- Limitar privilegios y recursos a los módulos que realmente los necesitan.

### 1.18 No confiar en la ofuscación

La seguridad por oscuridad no funciona, aunque engañar a atacantes puede ser útil (ej. honeypots).

### 1.19 Mantener mínima información de estado

Ejemplo: TCP SYN flood attack. Minimizar estado dificulta exploits.

### 1.20 Adoptar medidas prácticas con las que los usuarios puedan vivir

- Interfaz de usuario que facilite buenas prácticas.
- Modelos mentales consistentes con la realidad.
- Usuarios pueden eludir medidas excesivas; ser realista.

### 1.21 Asegurarse que un individuo es responsable

- No crear cuentas grupales.
- Asignar claramente la responsabilidad de seguridad.

### 1.22 Los programas deben autolimitar su consumo de recursos

Imponer límites de procesos, memoria, etc., para evitar ataques de agotamiento.

### 1.23 Asegurarse de que sea posible reconstruir los eventos

- Loguear operaciones y cambios de datos.

### 1.24 Eliminar puntos débiles

- Mantener un nivel consistente de seguridad.
- Evitar que medidas poco razonables fomenten back doors.

### 1.25 Construir varios niveles de defensa

No poner todos los huevos en la misma canasta.

### 1.26 Tratar a la aplicación como un todo

### 1.27 Reusar código seguro

### 1.28 No basar la seguridad solamente en paquetes de software

### 1.29 No dejar que las necesidades de seguridad sobrepasen los principios democráticos

### 1.30 Recordar preguntarse "¿Me olvidé de algo?"

## 2 Caso de ejemplo: Postfix MTA

Postfix es un **Mail Transfer Agent (MTA)** desarrollado por **Wietse Venema** como reemplazo seguro y eficiente de Sendmail. Su primera versión contenía más de **30.000 líneas de código** y fue diseñada bajo premisas de seguridad, facilidad de administración, rapidez y compatibilidad con Sendmail.

**Premisas de diseño principales:**

- **Rápido:** capaz de procesar grandes volúmenes de correo sin degradación significativa del rendimiento.
- **Fácil de administrar:** configuración clara y modular, con herramientas para monitoreo y mantenimiento.
- **Seguro:** incorpora principios de diseño seguro, como separación de privilegios y minimización de componentes con acceso privilegiado.
- **Compatible con Sendmail:** permitía a los administradores migrar desde Sendmail sin grandes cambios en los sistemas existentes.

Postfix fue lanzado **en 1990** y continúa con **mantenimiento activo**, siendo ampliamente utilizado en sistemas de correo modernos.

### 2.1 Arquitectura

Postfix está compuesto por un **conjunto de daemons cooperativos**:

- No tienen relación jerárquica.
- Cada daemon realiza una tarea específica.

**Múltiples niveles de defensa:**

- Casi todos los daemons pueden ejecutarse en `chroot`.
- Los módulos sensibles no son accesibles directamente desde la red.
- No utiliza `setuid`.

**Implementación modular:**

- Existe un módulo principal que ejecuta los daemons según la demanda.
- La cantidad de procesos es configurable.

### 2.2 Diseño Seguro

**Principio de menor privilegio:**

- Los daemons se ejecutan con bajos privilegios.
- Los módulos con acceso a la red funcionan dentro de un entorno `chroot`.
- Incluye componentes como `SMTP server` y `SMTP client`.

**Aislamiento:**

- Uso de procesos separados para aislar actividades.
- Los módulos sensibles de *delivery local* no son accesibles directamente desde la red.
- Algunos procesos internos son multithread, mientras que los que interactúan con el exterior son single-thread.

- Evita el uso de direcciones compartidas de memoria.

**Entorno controlado:**

- Ningún proceso se ejecuta bajo el control directo de un usuario.
- Todos los procesos son controlados por un *master daemon*.
- Esto previene exploits relacionados con señales, archivos abiertos y variables de entorno.

**Uso de perfiles y permisos:**

- No se utiliza `setuid`.
- Inicialmente, el directorio de colas de mail podía ser escrito por todos, aunque no accesible vía red.
- Actualmente se usa `postdrop` con `setgid` para escribir en las colas de mail.

**Confianza y validación:**

- Los daemons no confían en los contenidos de las colas, mensajes internos ni datos recibidos de la red.
- Se realizan chequeos exhaustivos cada vez que se procesan estos datos.

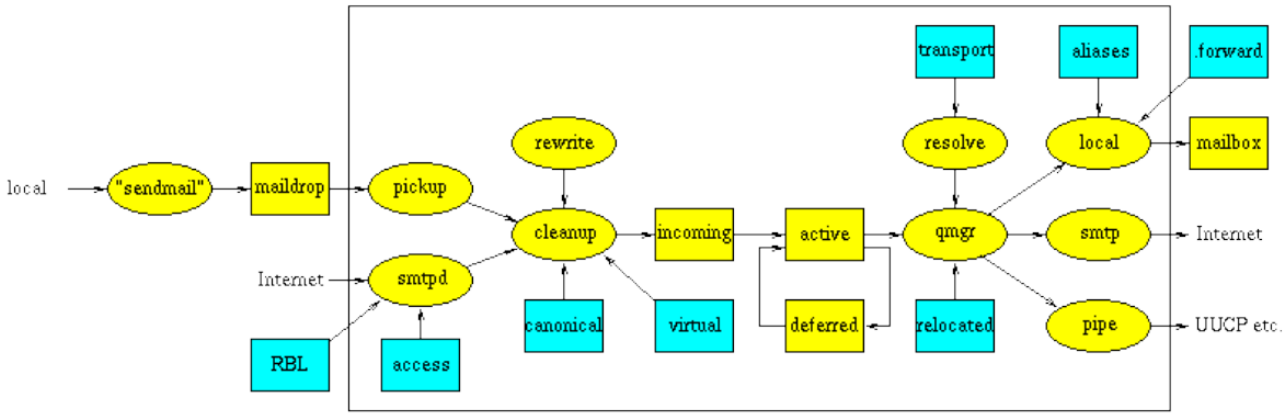
**Manejo de datos de entrada:**

- Se utiliza **asignación dinámica de memoria** para prevenir desbordamientos de buffer.
- Líneas demasiado largas en mensajes se dividen y reconstruyen al entregar.
- Mensajes de diagnóstico (*debug, info, error*) se truncan antes de enviarse a `syslog`.
- No se prevé defensa contra argumentos de línea de comando excesivamente largos.

**Otras defensas:**

- Se limita el número de instancias en memoria de cada objeto.
- Postfix pausa la ejecución en caso de problemas antes de enviar errores al cliente o reiniciar procesos fallidos.

## 2.3 Esquema de funcionamiento




- Óvalos amarillos: programas de mail.
- Cajas amarillas: colas de mail o archivos.
- Cajas azules: tablas de búsqueda.
- Los programas dentro del recuadro se ejecutan bajo control del *master daemon*.
- Los datos dentro del recuadro son propiedad del sistema Postfix.

# 3 MS SDL: Secure Development Lifecycle

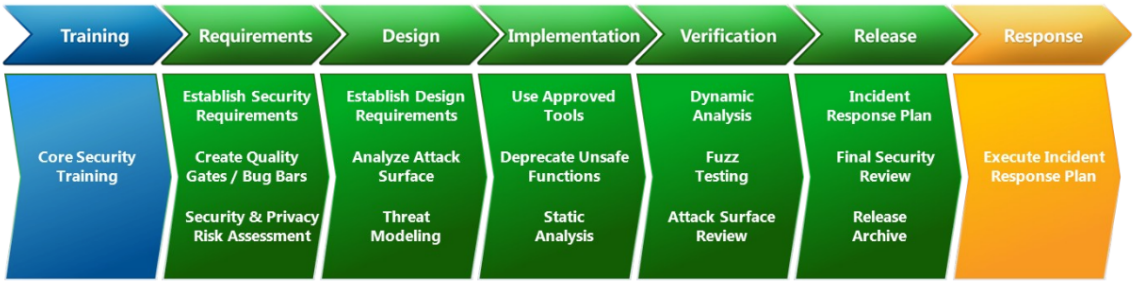
El **Secure Development Lifecycle (SDL)** es un proceso de desarrollo de software propuesto por Microsoft para mejorar desde el punto de vista de seguridad el software desarrollado.

## 3.1 Conceptos Clave del SDL

- **Superficie de Ataque:** Puntos de acceso al sistema o aplicación que podrían ser aprovechados por un atacante. Principio de seguridad: reducir la superficie de ataque, dado que todo código tiene vulnerabilidades; si reducimos la exposición, reducimos el riesgo.
- **Threat Modeling:** Conjunto de posibles ataques a considerar contra el software. SDL clasifica los riesgos según STRIDE:
  - **Spoofing:** Suplantación de la identidad del usuario.
  - **Tampering:** Modificación no autorizada de datos o código.
  - **Repudiation:** Negación de acciones realizadas por un usuario.
  - **Information Disclosure:** Exposición no autorizada de información.
  - **Denial of Service (DoS):** Ataques que impiden el funcionamiento del sistema.
  - **Elevation of Privilege:** Escalada no autorizada de privilegios.

## 3.2 Fases del SDL

El proceso de SDL incluye varias fases clave para integrar la seguridad en todo el ciclo de vida del software:




## 3.3 Fases del SDL

El proceso de **Secure Development Lifecycle (SDL)** incluye varias fases clave para integrar la seguridad en todo el ciclo de vida del software:

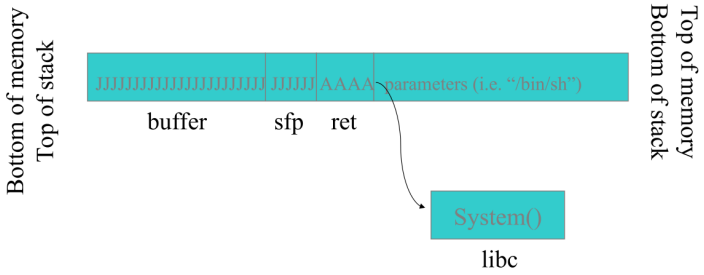
- **Training:** Capacitación de los desarrolladores y personal involucrado en buenas prácticas de seguridad, concienciación sobre amenazas y técnicas de mitigación.
- **Requirements:** Definición de requerimientos de seguridad junto con los funcionales, considerando la reducción de la superficie de ataque y cumplimiento de políticas.
- **Design:** Diseño seguro de la arquitectura del software, inclusión de controles de seguridad, modelado de amenazas (*Threat Modeling*) y análisis de riesgos.
- **Implementation:** Desarrollo seguro del código, siguiendo prácticas de codificación segura, revisiones de código y pruebas unitarias enfocadas en seguridad.
- **Verification:** Validación y pruebas de seguridad, incluyendo análisis de código, pruebas de penetración, revisión de cumplimiento de políticas y aseguramiento de que los controles implementados funcionan correctamente.
- **Release:** Preparación para la entrega del software, incluyendo documentación de seguridad, escaneo final de vulnerabilidades y verificación de cumplimiento de políticas de seguridad.
- **Response:** Monitoreo y gestión de vulnerabilidades post-lanzamiento, emisión de parches, gestión de incidentes y retroalimentación para futuras versiones.

# 4 Buffer Overflow

Los ataques por desbordamiento de búfer (buffer overflow) permiten, en el peor de los casos, ejecutar código propio dentro del contexto del proceso víctima. A continuación se enumeran y describen las principales líneas de defensa para mitigar este tipo de ataques:

- **Validación de parámetros:** Escribir código que valide correctamente todos los parámetros de entrada (listas blancas, comprobación de longitudes, normalización), evitando confiar en tamaños asumidos.
- **Buffers no ejecutables (NX / DEP):** Marcar las regiones de datos y pila como no ejecutables para prevenir que un shellcode colocado en esas zonas pueda ser ejecutado.
- **Chequeos de integridad antes de restaurar registros importantes:** Realizar comprobaciones (por ejemplo, verificación de canarios) antes de que el flujo de control restaure el `EIP/RIP` u otros registros críticos.
- **ASLR (Address Space Layout Randomization):** Aleatorizar el layout del espacio de direcciones (librerías, heap, stack, PIE) para dificultar la localización fiable de direcciones útiles a explotar.
- **Limitación de instancias y recursos:** Controlar la cantidad de instancias en memoria de cada objeto, y los límites de recursos para reducir la superficie y el impacto de fallos.

## 4.1 Return-to-libc




Return-to-libc es una técnica de explotación de vulnerabilidades de desbordamiento de buffer (*buffer overflow*) que permite ejecutar código sin inyectar código nuevo en memoria. En lugar de eso, reutiliza funciones existentes en la biblioteca estándar de C (`libc`), como `system()`, para ejecutar comandos arbitrarios.

### 4.1.1 Idea

- Los exploits tradicionales intentan inyectar código en la pila y ejecutar un *shellcode*.
- Con protecciones modernas como NX (no executable stack), la pila no es ejecutable.
- Return-to-libc evita esto reutilizando funciones legítimas ya presentes en memoria.
- Por ejemplo, se puede hacer que el programa llame a `system("/bin/sh")` sobrescribiendo la dirección de retorno.

**Cómo funciona**

1. Desbordamiento del buffer: se sobreescribe la dirección de retorno de la función vulnerable.
2. Redirección a una función `libc`: la dirección de retorno apunta a `system()`.
3. Paso de argumentos: se manipula la pila para que `system()` reciba como argumento un comando como `"/bin/sh"`.
4. Resultado: se abre un shell sin necesidad de ejecutar código inyectado.

**Requisitos y limitaciones**

- Se debe conocer la dirección exacta de la función en `libc`.
- Puede ser mitigado usando *Address Space Layout Randomization* (ASLR).

## 4.2 Return-oriented programming (ROP)

**Return-oriented programming (ROP)** es una técnica que permite eludir protecciones como NX/DEP usando fragmentos de código legítimo (gadgets) ya presentes en memoria y encadenándolos para construir una conducta maliciosa.

- ROP no inyecta código nuevo en regiones ejecutables; reutiliza trozos ya existentes en librerías o el binario.
- Es una de las técnicas más usadas actualmente para saltar distintos mecanismos de protección cuando existen gadgets suficientes y la defensa (ASLR, RELRO, CFI) es insuficiente o deshabilitada.
- Las defensas contra ROP pasan por: ASLR fuerte, RELRO/PIE, Control Flow Integrity (CFI) y reducciones de la presencia de gadgets (por ejemplo, compilación con opciones que minimicen gadgets).

## 4.3 Chequeos de integridad — Canarios de pila

Los **canarios** son valores conocidos colocados entre los buffers locales y los datos de control (saved frame pointer, return address) para detectar si un desbordamiento ha corrompido la zona intermedia. Si el canario cambia, el proceso detecta la corrupción y aborta o toma medidas seguras.

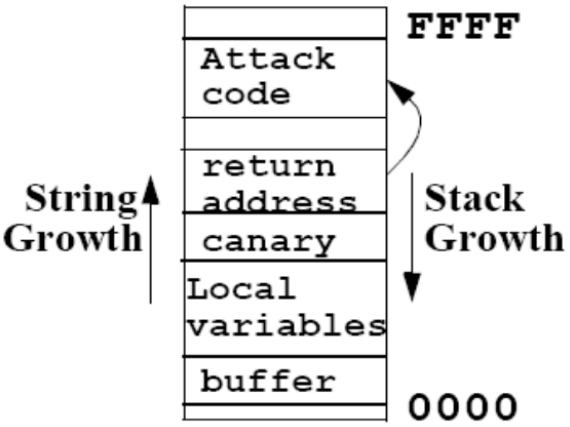
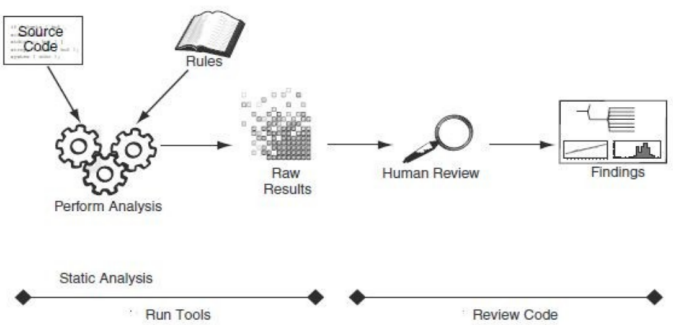



**Figure 2: StackGuard Defense Against Stack Smashing Attack**

- Implementaciones conocidas:
  - **Stack-Smashing Protector / ProPolice:** implementado en GCC y estándar en OpenBSD (y en compiladores con variantes).
  - **/GS:** mecanismo de Microsoft Visual C++.
- Puntos a considerar:
  - Los canarios pueden ser estáticos, aleatorios por proceso o derivados dinámicamente; los mejores son aleatorios e impredecibles.
  - No son infalibles; con suficiente información y escritura parcial un atacante avanzado puede intentar derrotarlos, pero aumentan significativamente la dificultad de explotación.

## 4.4 Análisis estático de código



Las herramientas de análisis estático examinan software sin ejecutarlo para detectar fallas. Pueden analizar código fuente, binarios o bytecode y usan técnicas diversas: análisis léxico, sintáctico, semántico, seguimiento de flujo de control y propagación de *tainted data*.

**Primeras herramientas (análisis léxico):**

- ITS4, RATS, Flawfinder — reglas simples (por ejemplo, avisos sobre `strcpy`).
- Rápidas pero con muchos falsos positivos; análisis superficial.

**Evolución hacia análisis más profundo:**

- Uso de técnicas de compiladores para modelar el programa (ASTs, CFG, análisis de datos).
- Detección de flujos de datos tainted, seguimiento interprocedural y uso de reglas expresivas.
- Trade-off: mayor precisión implica mayor coste computacional y retos de escalabilidad.

**Ejemplo de aplicación C con una vulnerabilidad**

```
int main(int argc, char* argv[]) {
    char buf1[1024];
    char buf2[1024];
    char* shortString = "a short string";
    strcpy(buf1, shortString); /* uso seguro de strcpy (tamaño conocido) */
    strcpy(buf2, argv[0]);     /* uso inseguro de strcpy (argv[0] puede ser largo) */
    ...
}
```

Este ejemplo ilustra un patrón típico: usar `strcpy` con datos de longitud no controlada (`argv[0]` u otros inputs) puede provocar desbordamientos si la entrada excede la capacidad del buffer.

**Herramientas y metodología**

- **Herramientas léxicas y SAST tempranas:** ITS4, RATS, Flawfinder — útiles como primera barrera.
- **Herramientas modernas:** analizadores más sofisticados integrados en cadenas de CI, que realizan análisis estático interprocedural, detección de flujos *tainted*, y reglas personalizadas.
- **Fuzzing y DAST:** pruebas dinámicas que envían entradas aleatorias o especialmente construidas para encontrar fallos de parsing y bordes no contemplados.
- **Guías y recursos educativos (defensivos):**
  - Documentación y guías sobre mitigaciones y desarrollo seguro.
  - Repositorios y proyectos que muestran ejemplos seguros y pruebas.

**Ejemplo de salida de una herramienta (RATS, ilustración)**

Rough Auditing Tool for Security (RATS) produce reportes indicando posibles fallos, por ejemplo:

```
www/source/core.c:53: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy more data
than can be handled, resulting in a buffer overflow.

magick/delegate.c:761: Medium: stat
A potential TOCTOU (Time Of Check, Time Of Use) vulnerability exists. This is the
first line where a check has occured.
The following line(s) contain uses that may match up with this check: 767 (open)
```

- `www/source/core.c:53: High: strcpy`
  Indica que en la ruta `www/source/core.c` en la línea 53 se detectó una invocación de `strcpy` considerada de severidad **High**. El comentario aclara el riesgo: si el segundo argumento (fuente) contiene más datos que el tamaño del buffer destino, puede producirse un *buffer overflow*. RATS alerta por la presencia de la función sin analizar si el tamaño está controlado.

- `magick/delegate.c:761: Medium: stat`
  Señala un posible problema de severidad **Medium** relacionado con el uso de `stat`. El mensaje menciona una potencial vulnerabilidad *TOCTOU* (Time Of Check, Time Of Use): se detectó una verificación (check) en la línea 761 y posteriormente un uso (open) en la línea 767 que podría corresponderse con esa verificación. El riesgo ocurre si entre la verificación y el uso un atacante modifica el estado del fichero (por ejemplo, mediante enlaces simbólicos).

**Interpretación práctica:** RATS marca ubicaciones sospechosas y proporciona pistas (líneas de check/uso) para que el auditor humano verifique el contexto y confirme si hay una vulnerabilidad real.

## 4.5 Evolución: análisis estático avanzado

Se busca armar un modelo del programa, con estructuras de datos que representen el código, y un lenguaje para armar reglas mucho más expresivo que encapsule más conocimiento. Para ello se recurren a técnicas clásicas de compiladores y análisis estático avanzado.

### 4.5.1 Técnicas incorporadas

- **Análisis sintáctico y semántico:** construcción del AST (árbol de sintaxis abstracta) y comprobaciones semánticas (tipos, alcance, resoluciones de símbolos).
- **Seguimiento de flujo de control (CFG):** generación del grafo de flujo de control para analizar rutas de ejecución posibles.
- **Análisis de flujo de datos:** propagación de información a través del programa para inferir valores posibles y dependencias.
- **Propagación de *tainted data*:** seguimiento de datos no confiables desde fuentes (inputs) hasta sinks (puntos sensibles) para detectar vulnerabilidades como inyección.
- **Modelado interprocedural:** análisis que cruza fronteras de funciones/módulos para comprender efectos globales.
- **Análisis simbólico y abstracción:** uso de símbolos o abstracciones (abstract interpretation) para representar conjuntos de valores y mantener la escalabilidad.

### 4.5.2 Objetivo y trade-offs

- **Objetivo:** elevar la precisión del análisis (menos falsos positivos/negativos) y capturar patrones complejos de vulnerabilidad mediante reglas más expresivas.
- **Trade-off:** existe un compromiso entre *precisión*, *profundidad* (qué tan profundo/interprocedural es el análisis) y *escalabilidad* (tiempo y memoria para analizar grandes bases de código). A mayor precisión y profundidad, mayor coste computacional.

### 4.5.3 Fortify

- **Descripción:** Fortify es una plataforma comercial de análisis estático enfocada en seguridad que construye modelos del programa y aplica reglas de seguridad avanzadas.
- **Lenguajes soportados (ejemplos):** `ASP.NET`, `C/C++`, `C#`, `COBOL`, `ASP clásico`, `VB6`, `ColdFusion`, `JavaScript`, `VBScript`, `Java`, `JSP`, `PL/SQL`, `T-SQL`, `PHP`, `VB.NET` y otros lenguajes .NET.
- **Plataformas:** Windows, Solaris, Linux, Mac OS X, HP-UX, AIX.
- **Frameworks y IDEs:** J2EE/EJB, Struts, Hibernate. Integraciones con Microsoft Visual Studio, Eclipse, WebSphere Application Developer, IBM RAD, etc.

**Otros analizadores estáticos**

- **Coverity:** comercial; ofrece análisis gratuito para proyectos *open-source*. Nacido de investigación en la Universidad de Stanford. Fuerte en detección de bugs y ciertos problemas de seguridad.
- **CodeQL:** herramienta de Microsoft / GitHub que permite expresar consultas en un lenguaje declarativo (QL) sobre una base de datos del código; potente para buscar patrones de seguridad y errores.
- **FindBugs / SpotBugs:** licencia GPL; históricamente para Java. FindBugs fue desarrollado en la Universidad de Maryland; su sucesor activo es *SpotBugs*. No está orientado exclusivamente a seguridad, pero detecta antipatrónes.

- **SonarQube:** soporta más de 20 lenguajes; muy utilizada en pipelines DevOps/CI para calidad de código, métricas y detección de vulnerabilidades con buena integración en flujos de trabajo.