

Unidad 6

Desarrollo seguro – parte 2

- **Definición:**

Conjunto de principios de diseño de alto nivel y de decisiones que permiten que un programador diga “sí” con confianza y “no” con certeza.

- Ej. saber que al agregar un componente no se comprometerá la seguridad del sistema

- **Propiedades:**

- Sirve como marco para el diseño seguro
- Debe ajustarse a un requerimiento de seguridad dado
- Se corresponde con un conjunto de documentos de diseño.

- **NO es una política**

Posibles contenidos del documento

- **Organización de la aplicación**
- **Estrategia de cambios**
- **Decisiones sobre que se compra y que no.**
- **Estructuras de datos principales**
- **Algoritmos**
- **Objetos principales**
- **Funcionalidad generica**
- **Métodos para procesar los errores (corrección o detección)**
- **Tolerancia a fallas**
- **...**

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 standards y guías tenemos disponibles ?
- ¿ Qué buenos ejemplos nos pueden ayudar ?

1. Comenzar haciendo preguntas

- **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 que entorno se ejecuta ?

- **Sobre nuestros 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 ?

2. Elegir un destino antes de comenzar

- **Antes de comenzar a tomar decisiones se debe entender bien lo que se necesita hacer.**
- **Ej. 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.
 - ...

3. **Decidir cuanta seguridad es suficiente**
 - **El nivel de seguridad de una aplicación depende fuertemente de:**
 - Tamaño y naturaleza de los riesgos.
 - Costo de las medidas de seguridad a programar.
 - **Ej. NO hacer aplicaciones tan seguras como sea posible SINO hacerlas suficientemente seguras.**
 - El nivel aceptable de seguridad se debe determinar objetivamente, y no porque se me acaba el tiempo pactado.
 - En ciertas áreas de negocio existen standards. (ej. sector financiero)

- 4. Emplear técnicas standard de ingeniería**
 - **Una buena seguridad requiere un buen diseño de software y buenas técnicas de diseño.**
 - **Principales factores para los ataques de seguridad**
 - Falta de diseño.
 - Debilidad humana.
 - Prácticas de codificación pobres.
 - **Una buena arquitectura de seguridad NO elimina el último punto.**

5. Identificar lo que se asume

- **Ej. Si recibimos un paquete TCP con el SYN flag en ON significa que el emisor quiere iniciar una sesión con nosotros.**
 - Asumimos que no hay mala intención.
 - Asumimos que el emisor es quien dice ser.
- **Ej. Los usuarios de este programa son humanos.**
 - Puede que no sea una persona y se ejecute un script cientos de veces...
 - ¿Cómo solucionaron esto yahoo/hotmail/gmail en sus sistemas de signup ? (Captcha)

6. **Incluir la seguridad desde el primer día**
 - **Evitar aplicar agregar los controles de seguridad más tarde.**
 - Ej. agregar cifrado de datos podría no ser suficiente.

- 7. Diseñar con el enemigo en la mente**
 - Se debe tratar de anticipar como un atacante resolverá el rompecabezas de nuestra infraestructura de seguridad.**

- 8. **Comprender y respetar la cadena de confianza**
 - **NO invocar a programas no confiables desde otro que si lo es.**
 - Como regla general: no delegar la autoridad para ejecutar una acción sin haber delegado la responsabilidad de chequear si dicha acción es apropiada.

- 9. **Ser tacaño con los privilegios**
 - **Principio del menor privilegio**
 - Un programa debe operar con los privilegios suficientes para realizar su tarea.
 - **Ej. Si se necesita leer un valor de un archivo no abrirlo con permisos de read/write.**

- 10. Testear todas las acciones propuestas contra la política.**
- **Asegurarse que las decisiones tomadas en todo momento por el software cumplen con las políticas de seguridad.**
 - **Ej. Antes de agregar un producto al carro de compras asegurarse de que pertenezca al usuario.**
 - No se debe reautenticar permanentemente a un usuario.
 - Deberíamos tratar de verificar que su sesión no ha expirado, que su conexión aun se encuentra activa, que no hay reglas adicionales sobre el manejo del carro de compras, ...
 - **Ej. Acceso a archivos en UNIX.**

11. Construir niveles apropiados de tolerancia a fallas.

- **Primero se debe identificar la funcionalidad de la misión crítica de la organización. Y luego utilizar las tres Rs:**
 - Resistencia (la capacidad de disuadir ataques)
 - Reconocimiento (la capacidad de reconocer los ataques y la extensión de los daños que producen)
 - Recuperación (la capacidad de proveer servicios y activos esenciales durante el ataque y recuperar la totalidad de los servicios luego del mismo.)

12. Tratar los temas de manejo de errores de manera apropiada.

- **Es muy común que un manejo de errores no apropiado genere una vulnerabilidad.**

¿ Qué hacer ?

- Arquitecto: decide sobre un plan general de manejo de errores.
Ej. parar solo ante errores no previstos y loguear el resto.
- Diseñador: determina como la aplicación va a detectar los errores, como los va a discriminar y como va a responder a ellos.
- Programador: captura las condiciones de error y responde a las mismas según el diseño.
- Operador: controla los procesos (para ver si terminado en forma anormal) y revisa los logs para ver los posibles reportes de errores.

13. “Degrade Gracefully”

- **Cuando ocurre un problema, la aplicación pasa a operar de manera restringida o degrada su funcionalidad.**
- **Ej. SYN flood attack: no existe un control sobre el numero de conexiones abiertas ... (una solución es poner un límite)**
- **Ej. zonas de arrugado en los automóviles, cuando el automóvil choca se arruga para absorber el impacto.**

14. Fallar de manera segura

- **Ej. el firewall de la empresa falla: detener el trafico de red.**
- **Ej. luces de tráfico fallan: ¿ quedan en verde ?**
- **Ej. el software que determina si un respirador artificial debe detenerse porque el paciente murió falla: ¿ cómo queda el respirador apagado o prendido ?**

- 15. Elegir acciones y valores por defecto seguros.**
- **Ej. si se quiere saber si un usuario tiene autorizacion: asumir primero que NO tiene autorización, luego buscar.**

16. Mantener las cosas simples.

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.

18. No confiar en la ofuscación

- **La seguridad por oscuridad no funciona**

- Aún así el engañar a los atacantes es útil. ej: honeypots

- 19. Mantener mínima información de estado.**
 - **Ej: TCP SYN flood attack**
 - Mantener el estado hace posible el exploit.
 - **La no existencia de información de estado hace que los atacantes tengan más dificultades para cambiarlo.**

- 20. Adoptar medidas prácticas con las que los usuarios puedan vivir.**
- **En teoría no debería haber diferencia entre la teoría y la práctica, pero en realidad la hay.**
 - Hay que ser realista.
 - **Elegir la interfaz de usuario que facilite hacer las cosas bien.**
 - **Usar modelos mentales del mundo real.**
 - **Los usuarios pueden saltar las medidas de seguridad por ser demasiado duras.**
 - ¿Cada cuanto se cambian las contraseñas ?

21. Asegurarse que un individuo es responsable

- **No crear cuentas grupales.**
- **Debe ser difícil para una persona pasar por otra.**
- **La responsabilidad por los temas de seguridad debe ser claramente asignada.**
 - ¿ Quién es responsable de instalar los parches de seguridad ?.

22. Los programas deben autolimitar su consumo de recursos.

- **Método de ataque muy común.**
 - Agotar los recursos del sistema para de esta manera colgarlo.
- **Imponer limites en el consumo de recursos**
 - Ej. número de procesos, cantidad de memoria.

23. Asegurarse de que sea posible reconstruir los eventos

- **Loguear las operaciones**
 - Ej registrar los cambios en los datos.

24. Eliminar puntos débiles.

- **Asegurar un nivel de medidas de seguridad consistente a través de todo el programa.**
- **También significa que las medidas de seguridad son lo suficientemente razonables para no alentar a los usuarios a introducir “back doors”.**

25. Construir varios niveles de defensa.

- No poner todos los huevos en la misma canasta !

26. Tratar a la aplicación como a un todo.

27. Reusar código seguro.

28. No basar la seguridad solamente en paquetes de software.

29. No dejar que las necesidades de seguridad sobrepasen los principios democráticos.

30. Recordar preguntarse “¿ Me olvide de algo ?”

Caso de ejemplo: Postfix MTA

- **Creado por Wietse Venema para reemplazar a Sendmail.**
- **Más de 30.000 líneas de código en su primera versión.**
- **Premisas de diseño:**
 - rápido
 - fácil de administrar
 - seguro
 - compatible con Sendmail.
- **Primera versión en 1990, actualmente versión 2.10.**

<http://www.postfix.org/>

- **Conjunto de daemons cooperativos.**
 - No tienen relación jerárquica.
 - Cada uno realiza una tarea.
- **Utilización de múltiples niveles de defensa:**
 - Casi todos los daemons pueden ser ejecutados en entorno chroot.
 - No se puede acceder de manera directa desde la red a los módulos sensibles.
 - No utiliza setuid.
- **Implementado como un módulo principal que ejecuta daemons según la demanda.**
 - La cantidad de procesos es configurable.

- **Menor privilegio**
 - Los daemons se pueden ejecutar:
 - con bajos privilegios
 - en un entorno chroot
 - En particular los módulos que tienen acceso a la red pueden funcionar de la manera descripta.
 - SMTP server.
 - SMTP client.

- **Aislamiento**

- Utilización de procesos separados para aislar las actividades entre ellos.
- No se puede acceder de manera directa desde la red a los módulos sensibles de delivery local.
- Algunos procesos internos son multi thread.
- Los procesos 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 de un proceso de usuario.
 - Los procesos de Postfix se ejecutan:
 - bajo el control de un master daemon
 - en un entorno controlado
 - sin relación de jerarquía con procesos de usuario
 - Esto elimina exploits que involucren:
 - señales entre procesos
 - archivos abiertos
 - variables de entorno, etc.

- **Uso de perfiles y permisos**
 - No se utiliza el setuid.
 - Inicialmente el directorio en el que se encuentran las colas de mail (maildrop) podía ser escrito por todos.
 - No accesible via red.
 - Se requiere permisos para acceder al mismo.
 - Problema: usuario malicioso puede llenar las colas.
 - Actualmente utiliza un programa (postdrop) que ejecuta con setgid para escribir en las colas de mail.

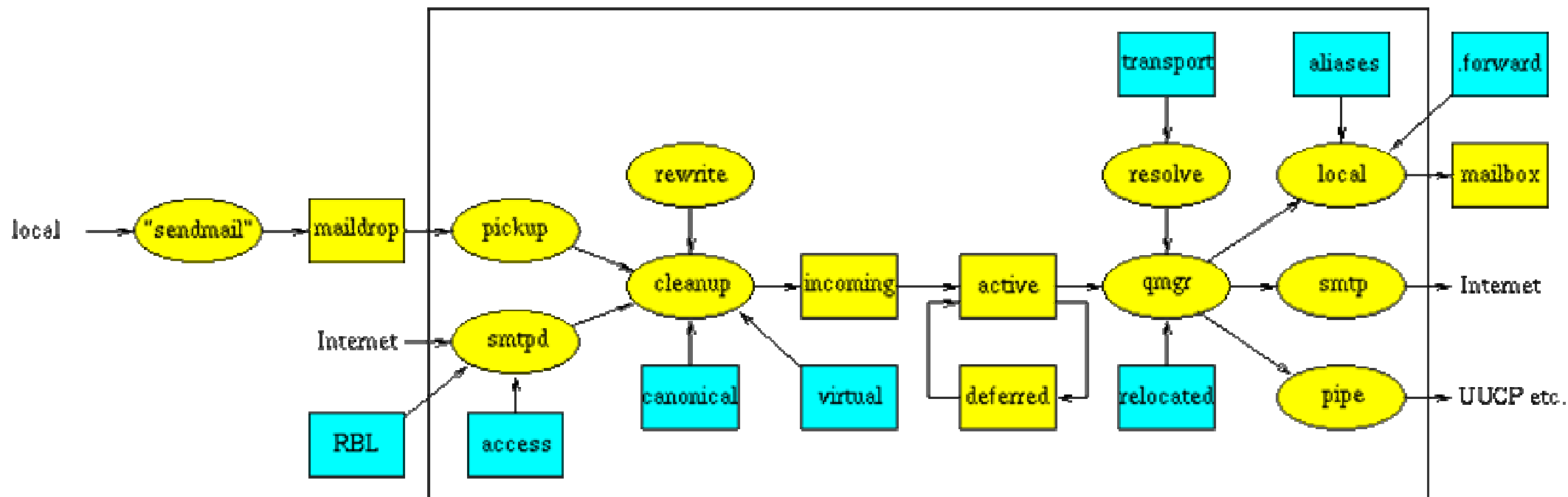
- **Confianza**
 - Los daemons no confían en:
 - los contenidos de las colas
 - los mensajes internos entre procesos
 - los datos recibidos de la red
 - Los daemons realizan chequeos cada vez que procesan los casos anteriores.

- **Datos de entrada**

- Se realiza una asignación dinámica de memoria para prevenir problemas de buffers.
- Las líneas demasiado largas en los mensajes se dividen en partes mas pequeñas, y se reconstruyen cuando el mensaje se entrega.
- Los mensajes de diagnostico (debug, info ,error, etc) se truncan antes de pasarlos a la interfaz de syslog.
- No se prevé defensa contra argumentos de línea de comando demasiado largos.

- **Otras defensas:**
 - El número de instancias en memoria de cada objeto esta limitado.
 - En caso de problemas Postfix deja de procesar (queda en pausa):
 - antes de enviar un error al cliente
 - antes de terminar con un error fatal
 - antes de intentar reiniciar un programa que fallo
- esto evita que aparezcan mas problemas.

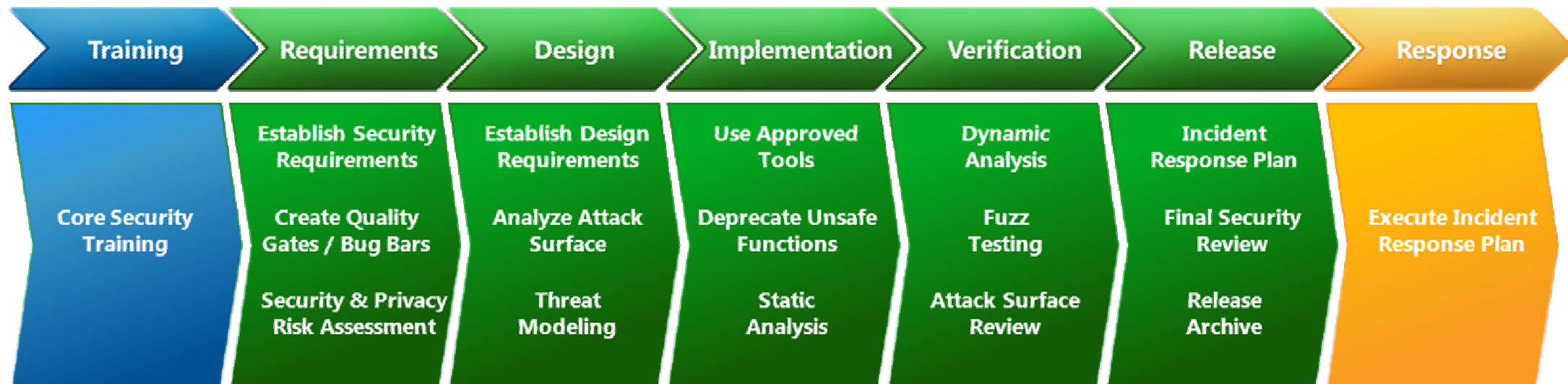
Esquema de funcionamiento



- Los óvalos amarillos son programas de mail.
- Las cajas amarillas con colas de mail o archivos.
- Las cajas azules son tablas de búsqueda.
- Los programas en el recuadro ejecutan bajo el control del master daemon de Postfix.
- Los datos en el recuadro son propiedad del sistema Postfix.

MS SDL: Secure Development Lifecycle

- Es un proceso de desarrollo de software propuesto por Microsoft para mejorar desde el punto de vista de seguridad el software desarrollado.



Más info: <http://www.microsoft.com/security/sdl/>

- **Superficie de Ataque:** Puntos de acceso al sistema/aplicación que podrían ser aprovechados por un atacante.
 - Principio de seguridad: reducir la superficie de ataque (todo código tiene vulnerabilidades, ergo 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 de la identidad del usuario
 - Tampering
 - Repudiation
 - Information Disclosure
 - Denial of Service
 - Elevation of Privilege

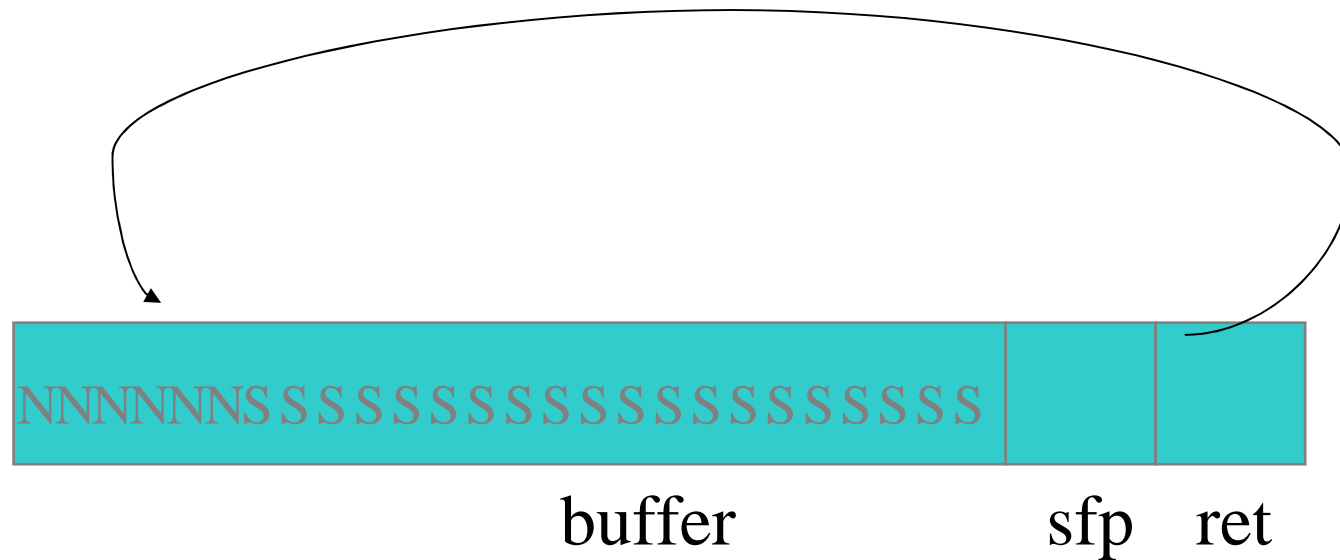


Demo Buffer overflows

Mecanismos de protección

Ejecutar código Propio

Bottom of memory
Top of stack



Top of memory
Bottom of stack

Uso NOP (0x90) para aumentar la posibilidad de éxito.

Mecanismos de defensa para evitar ataques de desbordamiento de buffers

- **Escribir código que valide correctamente todos los parámetros**
- **Buffers no ejecutables – prevenir la ejecución de instrucciones en los segmentos de datos y pila de la víctima (NX Bit)**
- **Realizar chequeos de integridad antes de restaurar registros importantes, como el EIP**
- **Address space layout randomization (ASLR)**

Return to libc

Bottom of memory
Top of stack



buffer

sfp

ret

parameters (i.e. "/bin/sh")

Top of memory
Bottom of stack



libc

- **Return-oriented programming**
- **Es la forma más usada actualmente para saltar los distintos mecanismos de protección.**

Chequeos de integridad - Canarios

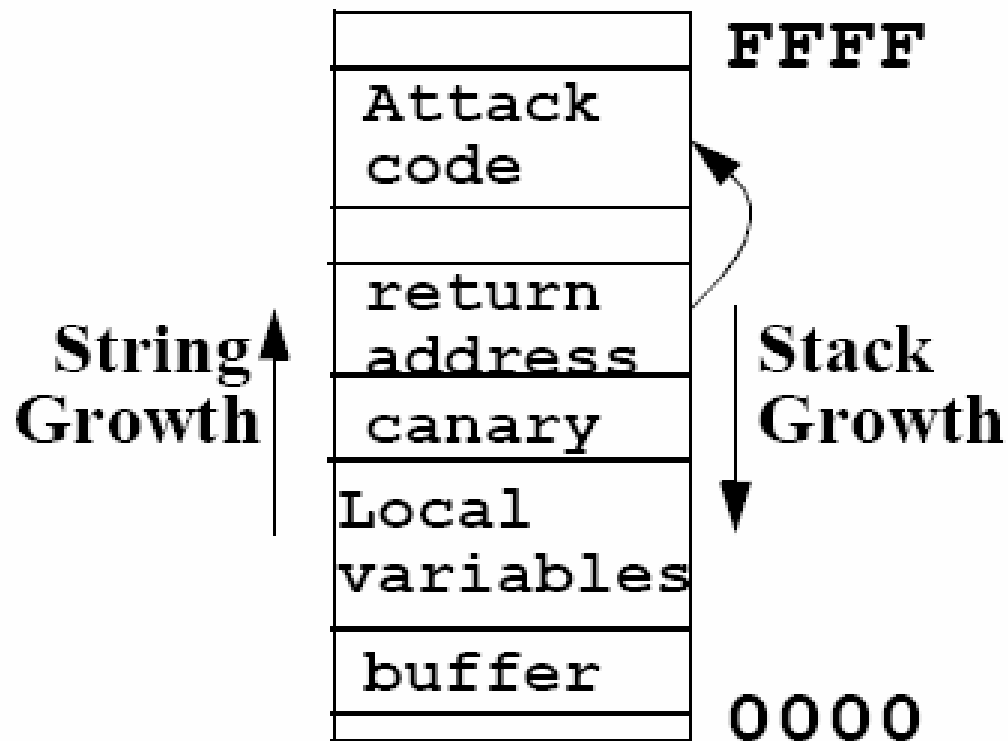


Figure 2: StackGuard Defense Against Stack Smashing Attack

Los canarios son valores conocidos que se ponen entre un buffer y datos de control para monitorear desbordamiento del búfer. Cuando se produce el desbordamiento, se sobrescribe el canario, quedando en evidencia el problema.

- **Hay distintas implementaciones.**
- **Las más conocidas son :**
 - GCC Stack-Smashing Protector (ProPolice). Standard en OpenBSD. Versión reducida en GCC 4.1
 - /GS en compiladores de Microsoft

“StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.” Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 7th USENIX Security Symposium, January 1998, San Antonio, TX.

“Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.” Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. SANS 2000, Orlando FL, March 2000.

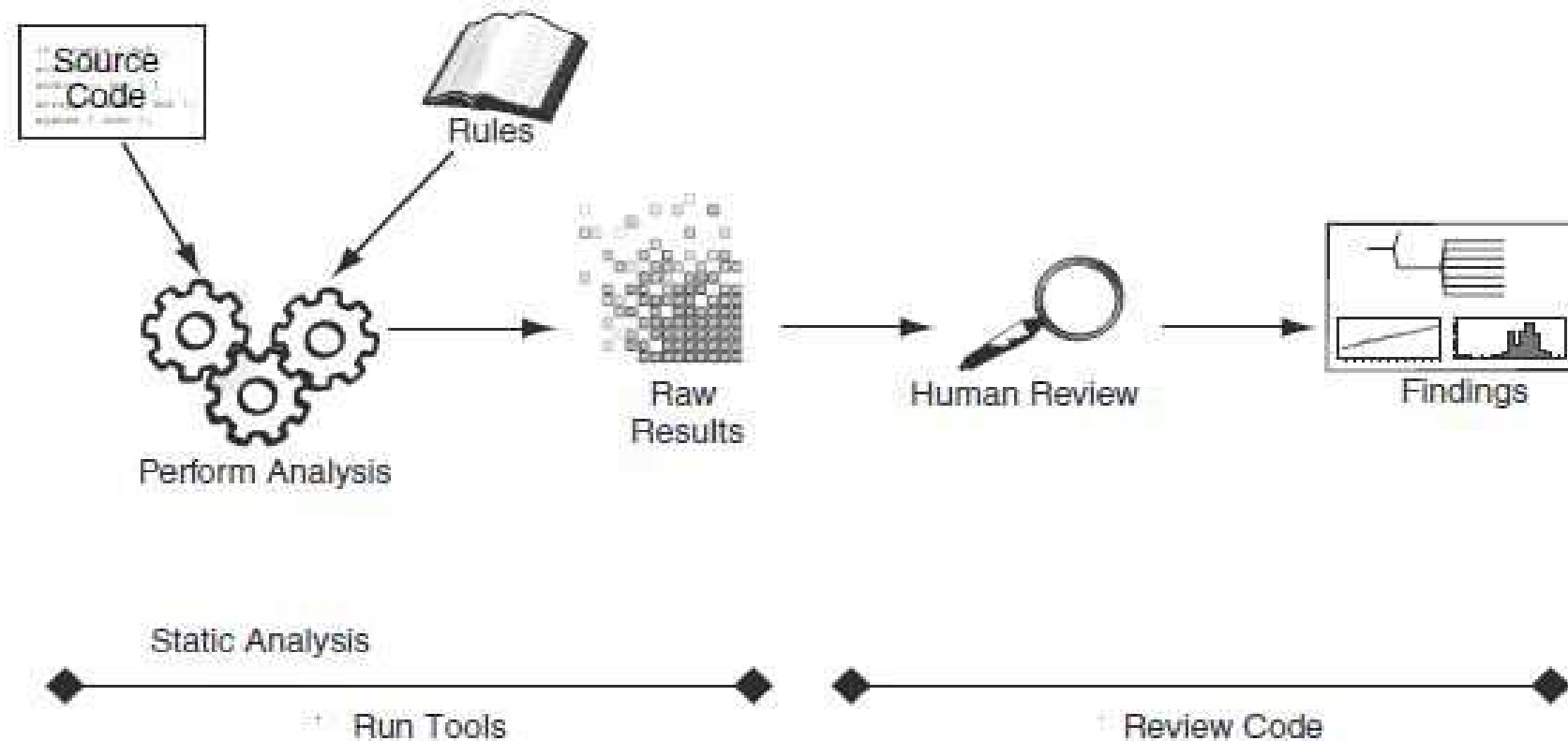
“Type-Assisted Dynamic Buffer Overflow Detection.” K.S.Hlee and J.S.Chapin. 11th Annual USENIX Security Symposium 2002.

Las herramientas de análisis estático de código examinan un software en busca de fallas, sin ejecutarlo. Algunas analizan el código fuente, otras el binario o el byte-code.

Utilizan distintas técnicas de análisis, y pueden presentar falsos positivos o falsos negativos. Estos últimos son más peligrosos, porque pueden llevar a una falsa sensación de seguridad.

Existen analizadores estáticos específicos para la problemática de seguridad.

Análisis estático de código



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 */  
    strcpy(buf2, argv[0]); /* uso inseguro de strcpy */  
    ...  
}
```


- **Analizadores Léxicos:**
 - ITS4 (2000), Rats, Flawfinder.
 - Reglas simples: “uso de strcpy() debe ser evitado”.
 - Muy rápidos, análisis poco profundo, muchos falsos positivos.

Ejemplo Rats

Rough Auditing Tool for Security. Soporta C, C++, Perl, PHP, Python y Ruby.

336 reglas para lenguaje C

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 occurred.

The following line(s) contain uses that may match up with this check: 767 (open)

Total lines analyzed: 331990

Total time 1.233556 seconds

269132 lines per second

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.

Se utilizan técnicas de compiladores.

Se incorpora análisis sintáctico y semántico, seguimiento de flujo de control y de datos, propagación de “tainted data”, entre otras técnicas.

Trade-off entre precisión, profundidad y escalabilidad.

Ejemplo: Fortify SCA

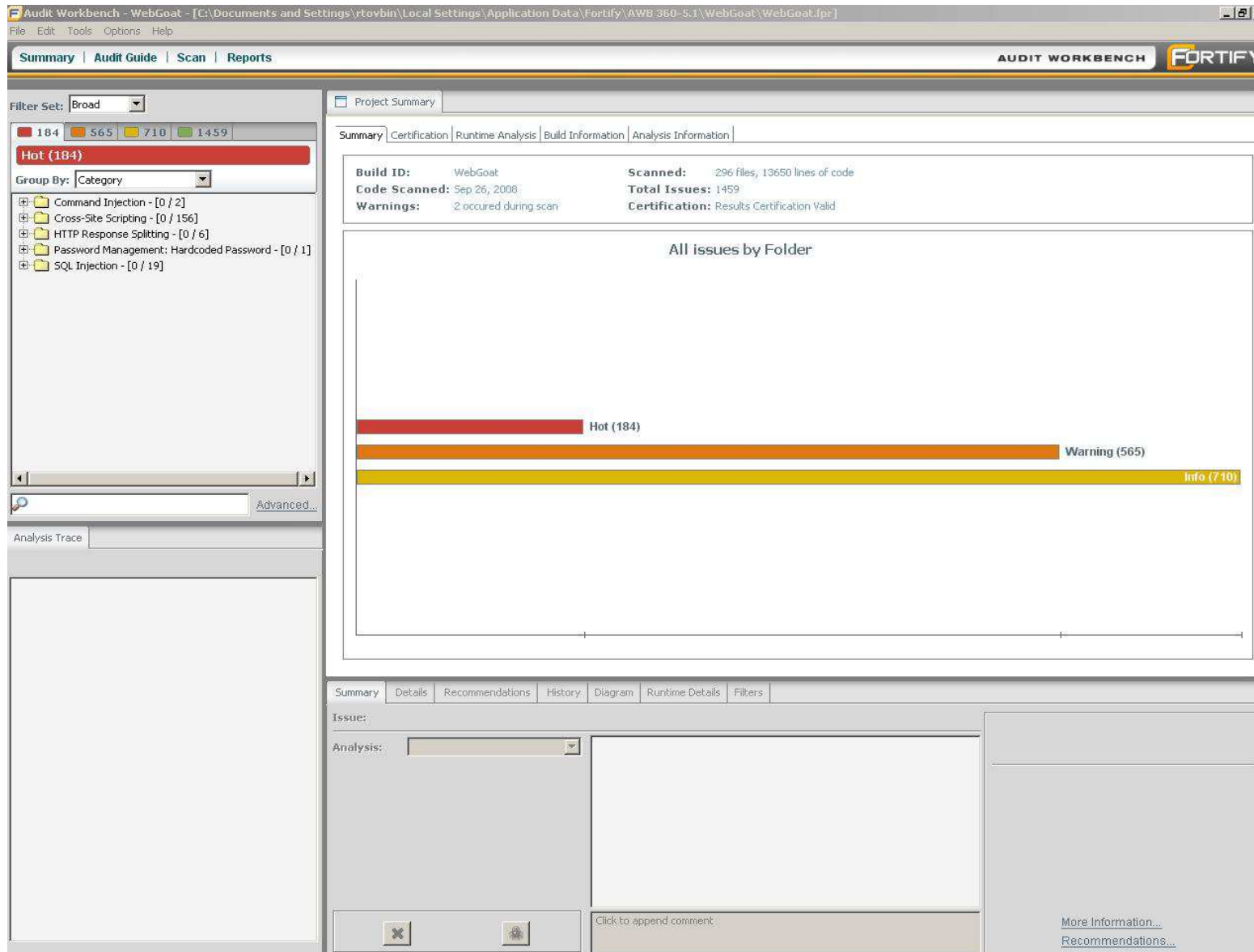
Lenguajes: ASP.NET, C/C++, C#, Cobol, ASP clásico, VB6, ColdFusion, javascript, vbscript, Java, JSP, PL/SQL, T-SQL, PHP, VB.NET y otros lenguajes .NET

Plataformas: Windows, Solaris, Linux, Mac OS X, HP-UX, AIX

Frameworks: J2EE/EJB, Struts, Hibernate

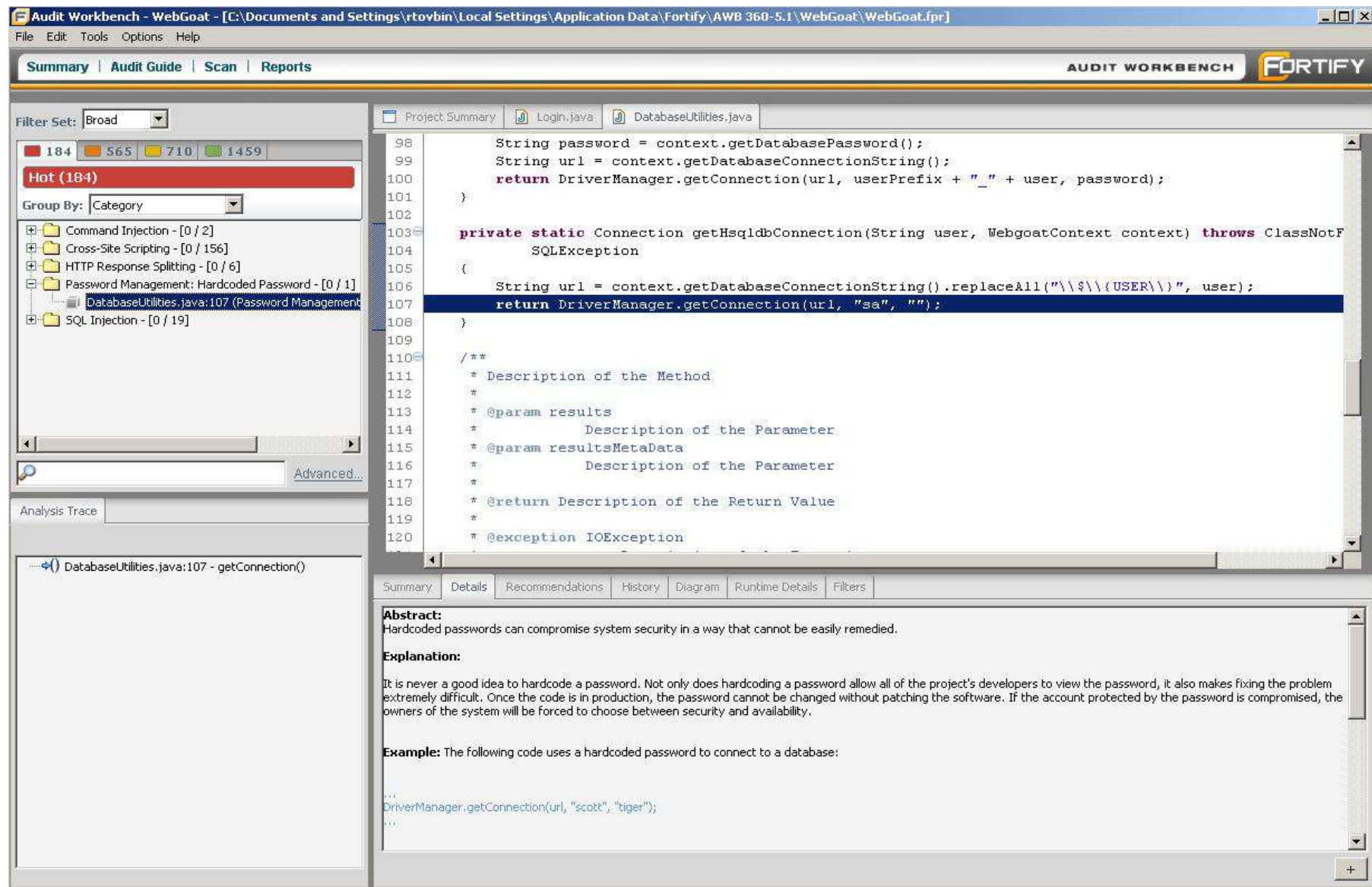
IDEs: Microsoft Visual Studio, Eclipse, Web Sphere Application Developer, IBM Rational Application Developer

Fortify y webgoat - I



The screenshot displays the Fortify Audit Workbench interface for a project named 'WebGoat'. The window title is 'Audit Workbench - WebGoat - [C:\Documents and Settings\rtovbin\Local Settings\Application Data\Fortify\AWB\360-5.1\WebGoat\WebGoat.fpr]'. The interface includes a menu bar (File, Edit, Tools, Options, Help) and a toolbar with 'Summary', 'Audit Guide', 'Scan', and 'Reports' buttons. A 'Filter Set' dropdown is set to 'Broad'. On the left, a 'Group By' dropdown is set to 'Category', showing a tree view of issue categories: Command Injection (0/2), Cross-Site Scripting (0/156), HTTP Response Splitting (0/6), Password Management: Hardcoded Password (0/1), and SQL Injection (0/19). The main area shows a 'Project Summary' tab with a 'Summary' sub-tab. It displays the following information: Build ID: WebGoat, Scanned: 296 files, 13650 lines of code, Code Scanned: Sep 26, 2008, Total Issues: 1459, Warnings: 2 occurred during scan, and Certification: Results Certification Valid. Below this, a horizontal bar chart titled 'All issues by Folder' shows the distribution of issues: Hot (184) in red, Warning (565) in orange, and Info (710) in yellow. The bottom section has tabs for 'Summary', 'Details', 'Recommendations', 'History', 'Diagram', 'Runtime Details', and 'Filters'. The 'Issue:' field is empty, and the 'Analysis:' dropdown is set to 'All'. At the bottom right, there are links for 'More Information...' and 'Recommendations...'. The bottom status bar includes a close button, a help button, and the text 'Click to append comment'.

Fortify y Webgoat - II



Audit Workbench - WebGoat - [C:\Documents and Settings\rtovbin\Local Settings\Application Data\Fortify\AWB 360-5.1\WebGoat\WebGoat.fpr]

File Edit Tools Options Help

Summary | Audit Guide | Scan | Reports

AUDIT WORKBENCH FORTIFY

Filter Set: Broad

184 565 710 1459

Hot (184)

Group By: Category

- Command Injection - [0 / 2]
- Cross-Site Scripting - [0 / 156]
- HTTP Response Splitting - [0 / 6]
- Password Management: Hardcoded Password - [0 / 1]
- DatabaseUtilities.java:107 (Password Management)
- SQL Injection - [0 / 19]

Advanced...

Analysis Trace

DatabaseUtilities.java:107 - getConnection()

```
98 String password = context.getDatabasePassword();
99 String url = context.getDatabaseConnectionString();
100 return DriverManager.getConnection(url, userPrefix + "_" + user, password);
101 }
102
103 private static Connection getHsqldbConnection(String user, WebgoatContext context) throws ClassNotFoundException,
104     SQLException
105 {
106     String url = context.getDatabaseConnectionString().replaceAll("\\$\\{\\{USER\\}\\}", user);
107     return DriverManager.getConnection(url, "sa", "");
108 }
109
110 /**
111  * Description of the Method
112  *
113  * @param results
114  *     Description of the Parameter
115  * @param resultsMetaData
116  *     Description of the Parameter
117  *
118  * @return Description of the Return Value
119  *
120  * @exception IOException
```

Summary Details Recommendations History Diagram Runtime Details Filters

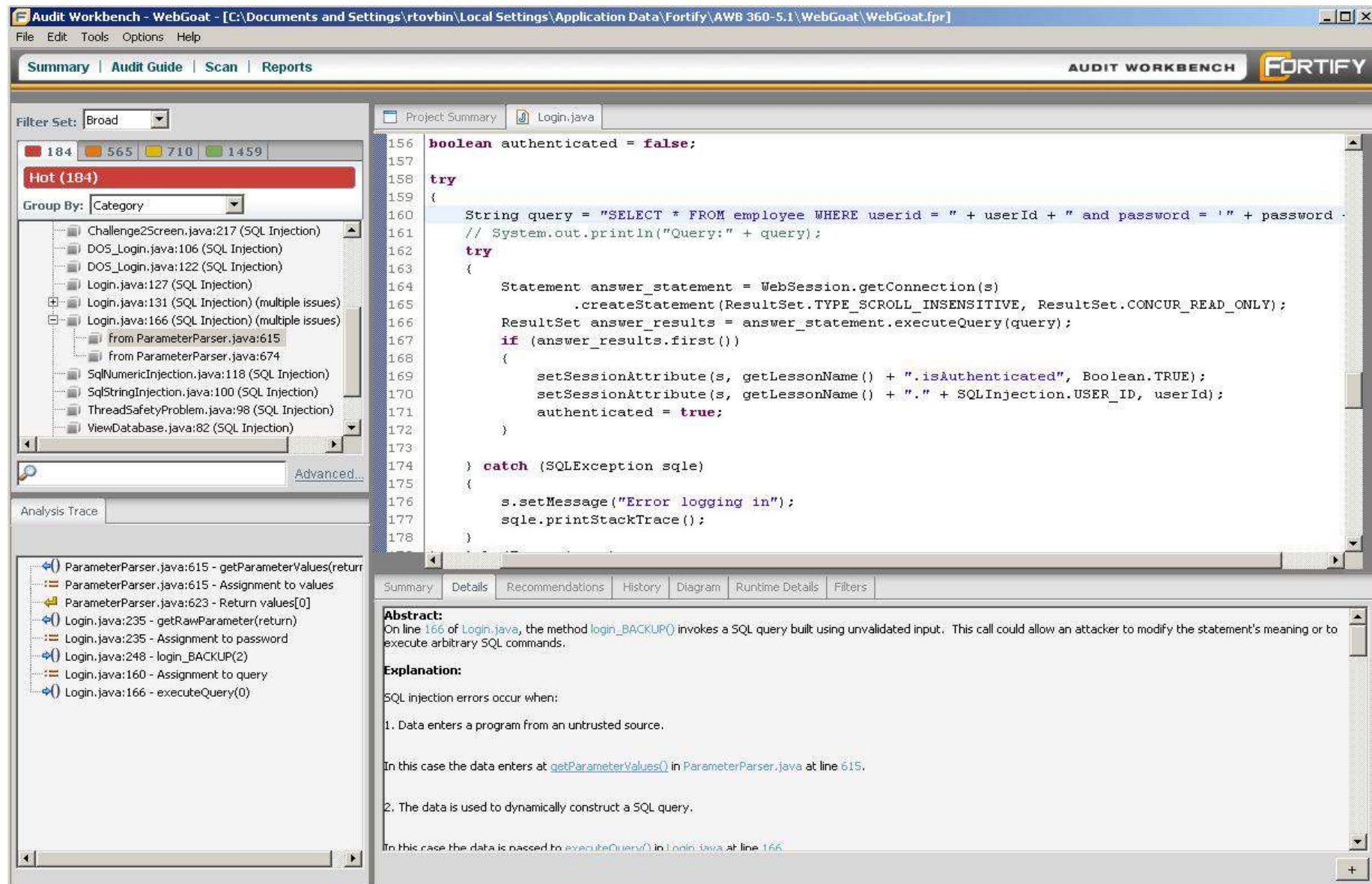
Abstract:
Hardcoded passwords can compromise system security in a way that cannot be easily remedied.

Explanation:
It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.

Example: The following code uses a hardcoded password to connect to a database:

```
...
DriverManager.getConnection(url, "scott", "tiger");
...
```


Fortify y Webgoat - III



The screenshot displays the Fortify Audit Workbench interface for a project named 'WebGoat'. The top menu bar includes 'File', 'Edit', 'Tools', 'Options', and 'Help'. The main toolbar shows 'Summary', 'Audit Guide', 'Scan', and 'Reports'. The 'Filter Set' is set to 'Broad', and the 'Group By' is set to 'Category'. The left sidebar shows a tree view of the project structure, with 'Login.java:166 (SQL Injection) (multiple issues)' selected. The main window displays the source code for 'Login.java', showing a SQL query constructed from user input. The code is as follows:

```
156 boolean authenticated = false;
157
158 try
159 {
160     String query = "SELECT * FROM employee WHERE userid = " + userId + " and password = '" + password + "'";
161     // System.out.println("Query: " + query);
162     try
163     {
164         Statement answer_statement = WebSession.getConnection(s)
165             .createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
166         ResultSet answer_results = answer_statement.executeQuery(query);
167         if (answer_results.first())
168         {
169             setSessionAttribute(s, getLessonName() + ".isAuthenticated", Boolean.TRUE);
170             setSessionAttribute(s, getLessonName() + ". " + SQLInjection.USER_ID, userId);
171             authenticated = true;
172         }
173     }
174     catch (SQLException sqle)
175     {
176         s.setMessage("Error logging in");
177         sqle.printStackTrace();
178     }
179 }
```

The bottom pane shows the 'Abstract' and 'Explanation' for the vulnerability. The 'Abstract' states: 'On line 166 of Login.java, the method login_BACKUP() invokes a SQL query built using unvalidated input. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.' The 'Explanation' states: 'SQL injection errors occur when: 1. Data enters a program from an untrusted source. In this case the data enters at getParameterValues() in ParameterParser.java at line 615. 2. The data is used to dynamically construct a SQL query. In this case the data is passed to executeQuery() in Login.java at line 166.'

Otros analizadores estáticos

- **Coverity Prevent.** Comercial, pero analiza en forma gratuita aplicaciones open-source. Desarrollado por investigadores de la Universidad de Stanford.
- **Ounce Core (IBM).** Comercial. Al igual que Fortify SCA, orientado a la seguridad.
- **Microsoft FXCop.** Gratuito. Analiza aplicaciones .NET. Se puede integrar a MS Visual Studio.
- **Findbugs.** GPL. Solo soporta Lenguaje Java. Desarrollado por Universidad de Maryland. No está orientado únicamente a la seguridad.

¿ Por qué la gente codifica mal ?

- **Factores educativos**
 - Falta capacitación en desarrollo seguro
- **Factores técnicos**
 - Muchas veces las vulnerabilidades no son el resultado de un error de codificación o de diseño en una aplicación, sino que resultan de problemas derivados de la interacción entre elementos del sistema que por sí solos no son inseguros. (Ej. Sun tarball)
- **Factores psicológicos**
 - Decisiones influenciadas por experiencias personales.
 - Modelos mentales sobre lo que un programa hace.
 - Formas de pensar acerca del software. (Ej. driver de mouse)
- **Factores del mundo real**
 - Presión por lanzar una aplicación.
 - Tan seguro como se pueda.

Bibliografía

