Seguridad en el desarrollo de aplicaciones

Escuela Politécnica Superior
UCLM - Albacete

26 noviembre 2004

Sergio Talens-Oliag
sto@uv.es

Introducción

- En esta charla hablaremos de la seguridad desde el punto de vista del programador, es decir, de aquello que tenemos que tener en cuenta en las etapas de diseño y codificación de los programas.
- Describiremos algunos errores de programación habituales que tienen implicaciones desde el punto de vista de la seguridad, daremos ejemplos de como se han usado para romper la seguridad de aplicaciones reales y comentaremos técnicas para detectar y corregir estos errores.

Principios de seguridad

- En general se suele decir que los tres objetivos fundamentales de la seguridad informática son:
  - *Confidencialidad*; el acceso a los activos del sistema está limitado a usuarios autorizados.
  - *Integridad*; los activos del sistema sólo pueden ser borrados o modificados por usuarios autorizados.
  - *Disponibilidad*; el acceso a los activos en un tiempo razonable está garantizado para usuarios autorizados.
- Hay que identificar los objetivos de seguridad de una aplicación para saber si un diseño o implementación los satisfacen.

¿Por qué se escriben programas inseguros?

- Hay poca bibliografía y la formación específica de los programadores es escasa.
- Es difícil programar de manera segura; no se suelen usar metodos de verificación formal.
- La seguridad no es un requisito a la hora de elegir un programa, por lo que se suele obviar.
- La seguridad incrementa los costes económicos y requiere más tiempo y esfuerzo en el desarrollo e implantación de aplicaciones.

Identificación de requisitos de seguridad

- *Common Criteria* o *CC* (ISO/IEC 15408:1999): estándar internacional para identificar y definir requisitos de seguridad. Se suele emplear para redactar dos tipos de documentos:
  - *Perfil de protección* (*Protection Profile* o *PP*): es un documento que define las propiedades de seguridad que se desea que tenga un producto; básicamente se trata de un listado de requisitos de seguridad.
  - *Objetivo de seguridad* (*Security Target* o *ST*): es un documento que describe lo que hace un producto que es relevante desde el punto de vista de la seguridad.

Entorno y objetivos de seguridad

- El primer paso para redactar un *PP* o un *ST* es identificar el entorno de seguridad: ¿En qué entorno vamos a trabajar? ¿Qué activos debemos proteger? ¿Para que se va a usar el producto?
- A partir de esta identificación obtenemos una serie de *supuestos* sobre el entorno (tipos de usuarios, tipo de red, etc.), una lista de posibles *amenazas* y una descripción de las políticas de seguridad de la organización.
- Por último se define un conjunto de objetivos de seguridad, demostrando que con ellos se combaten las amenazas y se cumplen las políticas.

Requisitos funcionales (1)

- Los *CC* definen un conjunto de requisitos funcionales de seguridad que puede necesitar una aplicación:
  - **Auditoría de Seguridad**: permitir el registro de eventos (hay que identificar cuales pueden ser interesantes desde el punto de vista de la seguridad).
  - **No rechazo** (*Non-repudiation*): uso de técnicas para verificar la identidad del emisor y/o el receptor de un mensaje.
  - **Soporte criptográfico**: si se usa criptografía ¿qué operaciones la usan? ¿que algoritmos y tamaños de clave se utilizan? ¿cómo se gestionan las claves?

Requisitos funcionales (2)

- **Protección de datos de usuario**: especificar una política para la gestión de datos de usuario (control de acceso y reglas de flujo de información).
- **Identificación y autenticación**: uso de técnicas de validación de identidad.
- **Gestión de seguridad**: definición de perfiles de usuario y niveles de acceso asociados.
- **Privacidad**: soporte del anonimato de los usuarios.
- **Autodefensa**: la aplicación debe incluir sistemas de validación de su funcionamiento y fallar de manera segura si esa validación no se cumple.

Requisitos funcionales (3)

- **Utilización de recursos**: soporte a la asignación de recursos, tolerancia a fallos, ...
- **Control de acceso**: soporte de sistemas que limiten el número y tipo de sesiones, el nivel de concurrencia y que proporcionen información sobre sesiones anteriores al usuario para ayudar a la detección de intrusos.
- **Rutas o canales fiables**: existencia de mecanismos que permitan al usuario identificar que accede a la aplicación real (p. ej. certificados digitales) evitando ataques del tipo *hombre en el medio*.

Desarrollo de aplicaciones seguras

- Para desarrollar una aplicación segura deberemos tener en cuenta los siguientes aspectos:
  1. Control de la entrada: validar todas las entradas
  2. Gestión de memoria: desbordamiento de buffers
  3. Estructura interna y diseño del programa.
  4. Llamadas a recursos externos: bibliotecas, scripts, ...
  5. Control de la salida: formato, restricciones, ...
  6. Problemas de los lenguajes de programación
  7. Otros: algoritmos criptográficos, de autentificación, ...

Control de la entrada

- Hay que validar *todas* las entradas que vienen de fuentes no fiables.
- Se debe determinar qué es legal y rechazar lo que no lo sea; siempre se debe verificar que algo es legal, la aproximación contraria (detección de entradas erroneas) puede fallar. El sistema se debe verificar generando entradas erróneas desconocidas.
- Hay que limitar la longitud máxima de la entrada

Validación de cadenas y números

- *Vigilar caracteres especiales*:
  - Caracteres de control
  - Caracteres especiales para el Shell, SQL, etc.
  - Delimitadores (p. ej. tabuladores, comas, :, etc.).
  - Verificar la códificación y decodificación de URLs y la validez de los juegos de caracteres.
  - Minimizar las *decodificaciones*; no decodificar más de una vez de modo inndecesario.
- *Números*: verificar máximos, mínimos y ceros.

Validación de otros tipos de datos

- *Direcciones de correo*: ver RFC 2822 y 822.
- *Nombres de fichero*: Omitir caracteres especiales (/, '\n', "", ...), omitir '../' de los patrones legales y no expandir expresiones regulares.
- *Cookies*: comprobar dominios.
- *HTML/XML*: prevenir *cross-posting*.
- *URI/URL*: validar antes de procesar.
- *Locales*: validar nombre del locale (`[A-Za-z][A-Za-z0-9_,+@\-\.=]*`) y rutas antes de usarlos.

Fuentes de datos (1)

- *Línea de órdenes*: hay que verificar valores de `argv`.
- *Variables de entorno*: se heredan y no sabemos el proceso que las creó, se pueden definir todo tipo de variables e incluso redefinir más de una vez. *Solución*: extraer y eliminar en el momento que las necesitemos.
- *Descriptores de fichero*: no se debe asumir que `stdin`, `stdin` o `stderr` están abiertos.

Fuentes de datos (2)

- *Contenido de ficheros*: nunca es fiable, hay que validar los formatos (p.ej. para ficheros de configuración).
- *Cookies y campos de formularios HTML*: pueden tomar valores arbitrarios, hay que validarlos e incluso pedir autentificación al emisor.
- *Otras fuentes*: el directorio de trabajo, las señales, la memoria, las llamadas a procedimiento remoto (System V IPC), el valor de umask, etc.

Errores de validación de entrada
Control de mínimos en sendmail (1)

- `sendmail -d`*`flag,value`* ajusta el flag número *flag* al valor *value*.
- El nombre de config. (`/etc/sendmail.cf`) se guarda en el segmento de datos anterior al vector de *flags*; en ese fichero se define `/bin/mail`.
- `sendmail` verificaba los valores máximos de los *flags*, pero no los mínimos (el formato de entrada no permitía valores < 0), pero para C en un micro de 32 bits un valor de tipo int >= 2^{31} es negativo.

Errores de validación de entrada
Control de mínimos en sendmail (2)

- La llamada `sendmail -d4294967269,117 -d4294967270,110 -d4294967271,113` cambia `etc` por `tmp` en el nombre del fichero de configuración y ahora `/etc/sendmail.cf` pasa a ser `/tmp/sendmail.cf`.
- Si se crea un fichero `/tmp/sendmail.cf` que dice que el programa de envío local es `/bin/sh` en lugar de /`bin/mail` y se realiza la llamada anterior se obtiene un shell de root en la máquina.

Errores de validación de entrada
Bug validando UNICODE en el IIS

- Accediendo a un URL como <`http://servidor.ms.iss/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir+c:\`> servido por un *Internet Information Server* 4.0 ó 5.0 sin parchear se ejecuta la orden `dir c:` y se devuelve el resultado vía http.
- El problema es simplemente que al parecer el IIS decodifica el UNICODE después de validar las rutas, en lugar hacerlo antes.

Desbordamientos de buffer

- Se producen cuando un atacante puede conseguir escribir datos fuera de los límites de un buffer (generalmente más allá del final), con lo que sobreescribe valores preexistentes.
- Si el *buffer* está en la pila el ataque se llama *stack overflow* y puede permitir cambiar direcciones de retorno e incluir código para ser ejecutado, que es lo que suelen hacer los *exploits* de este tipo de errores.
- Es posible porque C, C++ o el ensamblador no validan el acceso a los límites de los *buffers*.

Como evitar desbordamientos de buffer en C/C++

- Evitar o usar con mucho cuidado funciones peligrosas como `gets`, `strcpy`, `strcat`, `sprintf`, `scanf`, ...
- Trabajar de un modo consistente con funciones que trabajan con *buffers* de tamaño fijo o dinámicos:
  - C estándar con tamaño fijo: `strncpy`, `strncat`, ...
  - `strlcpy` y `strlcat` (*OpenBSD*, más fáciles y consistentes que `strncpy` y `strncat`).
  - C estándar con gestión dinámica: `malloc`, ...
  - C++ estándar: `std::string`.

Ejemplo de *buffer overflow*
wu-ftpd y la función realpath (1)

- La función `realpath()` convierte una ruta relativa en absoluta (se eliminan referencias a `../`).
- La implementación de la función usaba un *buffer* de tamaño fijo y no controlaba el acceso a posiciones más allá del tamaño del mismo.
- Un atacante con acceso al ftp en modo escritura podía crear una ruta arbitrariamente larga (p.ej. `mkdir AAA...; cd AAA...;` repetido una y otra vez, ...).

Ejemplo de *buffer overflow*
wu-ftpd y la función realpath (2)

- Al final de la ruta (calculando el tamaño a partir del tamaño del *buffer* empleado por `realpath`) el atacante creaba un nombre de fichero con una dirección de retorno y código máquina para ejecutar (p.ej. instrucciones para lanzar un shell).
- Cuando el ftpd llamaba a `realpath` para obtener la ruta absoluta en lugar de retornar la función ejecutaba el código proporcionado por el atacante.

Estructura y diseño de programas (1)

- Seguir principios de la ingeniería del software:
  - Siempre se debe trabajar con privilegios mínimos.
  - Simplicidad en el diseño: *KISS*.
  - Diseño abierto: no hay que depender de la ocultación.
  - Mediación completa: todos los accesos se controlan.
  - Valores por defecto seguros: acceso cerrado.
  - Separación de privilegios: control acceso multi-nivel.
  - Mínimo uso de recursos compartidos (p.ej. `/tmp`).
  - Facilidad de uso: los usuarios colaboran más.

Estructura y diseño de programas (2)

- *Interfaz segura*: debe ser mínima (simple), dar acceso a las funciones justas y no ser evitable. Siempre se asume que la confianza es mínima.
- *Separción de control y datos*: no se debe soportar el uso de macros almacenadas en los documentos.
- *Minimización de privilegios*: Concesión de privilegios mínimos, se abandonan en cuanto no son necesarios (ej. apertura puertos TCP), se controla su tiempo de validez y se conceden al mínimo número de módulos posible. También se puede limitar el acceso a distintos recursos: `FSUID`, `chroot`, `ulimit`, ...

Estructura y diseño de programas (3)

- *Valores por defecto seguros*:
  - instalación por defecto con muchas restricciones, el usuario es el que las relaja si es necesario,
  - nunca se instala nada con claves por defecto,
  - los programas se deben instalar de manera que no sean modificables por los usuarios (permisos).
- *Carga de valores iniciales segura*: desde `/etc/`.
- *Fallo seguro*: Se cancela el proceso de una petición si hay errores de proceso de la entrada o inesperados.

Estructura y diseño de programas (4)

- *Evitar condiciones de carrera*: Se producen cuando varios procesos se interfieren unos a otros; se pueden producir entre procesos en los que no confiamos (*problemas de secuencia*) o entre procesos en los que si confiamos (*interbloqueo*).
- *Problemas de secuencia*: se dan cuando se pueden producir cambios entre dos operaciones (la secuencia no es atómica); comunes al acceder a sistemas de archivos, sobre todo si se usan directorios compartidos (`/tmp/`).
- *Interbloqueos*: se pueden evitar reservando los recursos siempre en el mismo orden (desde todos los procesos).

Estructura y diseño de programas (5)

- *Confiar sólo en canales fiables*: las direcciones origen IP o de correo pueden ser falseadas, el DNS no es un sistema seguro, ...
- *Prevenir contenido malicioso cruzado* mediante filtrado, codificación y validación de los datos de entrada.
- *Ataques semánticos* (el usuario piensa que hace una cosa y en realidad está haciendo otra), como por ejemplo el uso de URLs erroneos: se minimiza su efecto educando al usuario y dándole pistas de donde está en cada caso (ej: http://foo.org@ham.net).

Uso de recursos externos (1)

- Llamar sólo a rutinas de biblioteca fiables: si no lo son escribimos nuestras propias versiones.
- Limitar los parámetros de las llamadas sólo a los valores aceptables.
- Escapar o prohibir los metacaracteres de *shell* antes de invocarla (si es qué está permitido, en general no se debería poder hacer).
- Escapar o prohibir los metacaracteres de otras herramientas como el SQL.

Uso de recursos externos (2)

- Llamar sólo a interfaces pensadas para programas.
- Evitar llamar a programas como `mail`, `mailx`, `ed`, `vi` o `emacs`; todos tienen caracteres de escape. Si se usan hay que aprender a prevenir esos mecanísmos de escape y evitarlos.
- Compruebar los retornos de todas las llamadas a bibliotecas y al sistema.
- Cifrar la información sensible (p.ej. usa SSL/TLS para transmitir datos privados vía Internet).
- Cifrar los datos en el disco si son críticos.

Control de la salida

- Minimizar respuesta al usuario:
  - Se registran los fallos pero no se explican a usuarios desconocidos.
  - No se deve enviar el número de versión del programa.
- Gestionar el agotamiento de recursos (disco lleno, usuario deconocido, ...)
- Control del formato de los datos (cadenas de formato).

Errores en el control de la salida
Cadenas de formato de salida (PHP < 4.0.3)

- Si se habilita el registro de errores en PHP se llama a la función `php_syslog` con información proporcionada por el usuario.
- `php_syslog` llama a `printf` usando esa información como cadena de formato.
- Un atacante puede forzar que un proceso sobreescriba las variables de su pila con datos arbitrarios, permitiendo a un atacante tomar el control del PHP, que es un proceso con los permisos del servidor web.

Lenguajes de programación (1)

- *Perl*: uso de opciones -w y -T, uso de open con tres parámetros (ver `perltut`), `use strict;`.
- *Python*: controlar el uso de `exec`, `eval`, `execfile` y `compile`; la función de entrada es peligrosa, no usar `rexec` ni `bastion`.
- *Shell*: Evitar usarla para programas seguros salvo que esté muy protegida; hay demasiados métodos para explotarla (espacios en blanco, caracteres de control o nombres que empiezan por -).

Lenguajes de programación (2)

- *PHP*: `register_globals="off"`, PHP 4.1.0+, usa `$_REQUEST` para acceder a datos externos y filtra datos usados por `fopen`.
- *C/C++*: control de tipos, uso de *enum* y *unsigned* cuando corresponda, control de datos de tipo *char* (y su signo), compilar los programas con todas las opciones de *Warnings* y resolver los fallos que aparezcan, controlar los *desbordamientos de buffer*.

Otros aspectos (1)

- Usar `/dev/(u?)random` para generar números aleatorios
- No enviar claves en claro a través de Internet
- Autentificación de usuarios vía web:
  - Para intranets, usar sistemas internos (p.ej. LDAP).
  - La autentificación web básica es en claro, evitarla.
  - Los certificados de cliente no están bien soportados, alternativamente se pueden usar claves (sobre un enlace cifrado) e intercambiar *cookies*.

Otros aspectos (2)

- Proteger los secretos (*contraseñas*, *claves*) en la memoria del usuario.
- Deshabilitar volcados de memoria con `ulimit`; controlar `mmap` para evitar *swapping*, no usar cadenas inmutables.
- Utilizar algorimos de cifrado y criptográficos ya existentes, no hay que inventar nada nuevo: *SSL/TLS*, *SSH*, *IPSec*, *OpenPGP* (*GnuPG*), *Kerberos AES* o *Triple-DES*, *RSA*, *SHA-1*, ...

Conclusiones

- Se puede resolver más del 95% de las vulnerabilidades evitando problemas conocidos:
  - Hay que validar las entradas, evitar desbordamientos de *buffer y* controlar contenidos y formatos de salida.
  - Estructura el programa: minimizar privilegios, evitar condiciones de carrera.
  - Cuidar las invocaciones (metacaracteres shell/SQL) y verificar los valores retorno de las funciones.
  - Hay que ser paranoico: realmente te persiguen.

Referencias

- David A. Wheeler, *Secure Programming for Linux and Unix HOWTO*, v3.010, 3 March 2003. http://www.dwheeler.com/secure-programs
- ISO/IEC 15408:1999, *The Common Criteria for Information Technology Security Evaluation (CC)*. http://www.commoncriteriaportal.org/
- OWASP, *Guide to Building Secure Web Applications*. http://www.owasp.org/documentation/guide.html