GRUPO DE SEGURIDAD INFORMÁTICA

Fundamentos de la Seguridad Informática

Seguridad en Aplicaciones

Introducción

- En una máquina *stand-alone*, estamos en control de los componentes de software que hacen entrada al sistema
- El software es seguro si puede manejar entradas intencionalmente malformadas
- Analizaremos las causas generales de las vulnerabilidades en el software
- Propondremos pautas para mitigar los problemas

Agenda

- Algunas causas básicas comunes
- Defensas
  - Prevención
  - Detección
  - Testing
- Atenuación
  - Mínimo privilegio
  - Mantenerse al día

Causas básicas

Presentar las causas básicas que llevan a fallas de seguridad del software

- Números y caracteres
- Representaciones canónicas
- Manejo de memoria
- Datos y código
- Condiciones de carrera (*Race Conditions*)

Seguridad y confiabilidad

- La seguridad del software está relacionada a la calidad y confiabilidad del software
- La confiabilidad está vinculada a las fallas accidentales
- Para hacer el soft más confiable, se testea con patrones de uso típicos
- → En el contexto de seguridad, el atacante selecciona la distribución de las entradas

Taxonomía de un *Malware*

- *Malware == Malicious software*
- Hay diferentes tipos:
  - Virus
    - es un pedazo de código que se replica automáticamente, anexado a otro
    - se dice que *infecta* un programa insertándose a si mismo en el código de un programa
  - Gusano (*Worm*)
    - es un programa que se replica, pero no infecta

Taxonomía de un *Malware* (2)

- Caballo de troya (Trojan horse)
  - es un programa con efectos laterales ocultos no especificados en su documentación y no previstos por el usuario
- Bombas lógicas
  - es ejecutado únicamente cuando una condición específica se cumple
- Backdoor
  - permite acceso no autorizado a computadoras comprometidas

Taxonomía de un *Malware* (3)

- Exploit
  - Explota una vulnerabilidad de software para ganar acceso no autorizado
- Rootkit
  - Software que se oculta de forma activa

Taxonomía de un *Malware* (3)

- HackTool
  - Herramientas de explotación, ataque y escaneo
- Spyware
  - Software que invade la privacidad del usuario

Codificaciones: Caracteres y números

- Muchos defectos del soft se deben a malas abstracciones
- Problemas de este tipo relevantes para la seguridad pueden encontrarse con conceptos elementales como caracteres y enteros
- Las descripciones de esos problemas se refieren a la forma en que dichos caracteres y enteros están representados en memoria

Codificaciones

- Hay muchas, dependen del contexto
  - URL Encoding
    - http://tools.ietf.org/html/rfc3986
  - UTF-8 encoding
  - ISO8859-1, Windows-1252, etc.
- Cada una expresa una abstracción

Caracteres (codif. UTF-8)

- Escriben una aplicación que debería dar a los usuarios acceso al subdirectorio `A/B/C`
- Los usuarios ingresan el archivo como entrada
- El path del archivo se construye como `A/B/C/entrada`
- Podrían escalar en el árbol de directorios usando `../`
- Acceso a los passwords: `../../../../etc/passwd`

Caracteres (2)

- Como contramedida, el desarrollador hace alguna validación de la entrada
- Filtra la combinación '. . /'?
- La codificación UTF-8 se define para usar Unicode en sistemas que fueron designados para ASCII (RFC 2259)
- Los chars ASCII son representados por los bytes ASCII (0x00 - 0x7F)

Caracteres (3)

- P.ej, el signo © es U00A9, en UTF-8 se escribe como 0xC2 0xA9
- Por ese motivo, un único caracter puede tener distinta representación: en el caso de '/':

  1 byte – 2F

  2 byte – C0 AF

  3 byte – E0 80 AF

URL encoding

- Usa codificación “porcentaje”
  - pct-encoded = "%" HEXDIG HEXDIG
- {IP}/scripts/..%25%32%66../winnt/system32/

  %25%32%66 == %2F

  %2F = ?
- A veces el hecho de realizar una lectura e interpretación de los caracteres cambia el significado

Overflow de enteros

- Son representados como cadenas binarias de largo fijo (precisión)
- Los lenguajes de programación tienen enteros con signo o sin signo, cortos, largos, etc.
- Comparación entre una var `size` con signo y `size_t` sin signo

```
if (size < sizeof(buf))
```

Overflow de enteros (2)

- Si `size` es negativo y el compilador lo "castea" con signo y `size_t` sin signo puede dar overflow
- También se puede acortar un valor: *Integer truncation*

Lección:

- Declarar todos los enteros como sin signo a no ser que se necesiten enteros negativos
- Tomar en cuenta *warnings* del compilador

Arreglos

- Los índices de los arreglos utilizan aritmética de enteros
- Si no se chequea que el resultado esté entre los valores permitidos, se va a asignar memoria fuera del arreglo
- Por lo tanto, se deberían verificar los índices del arreglo siempre

Representaciones canónicas

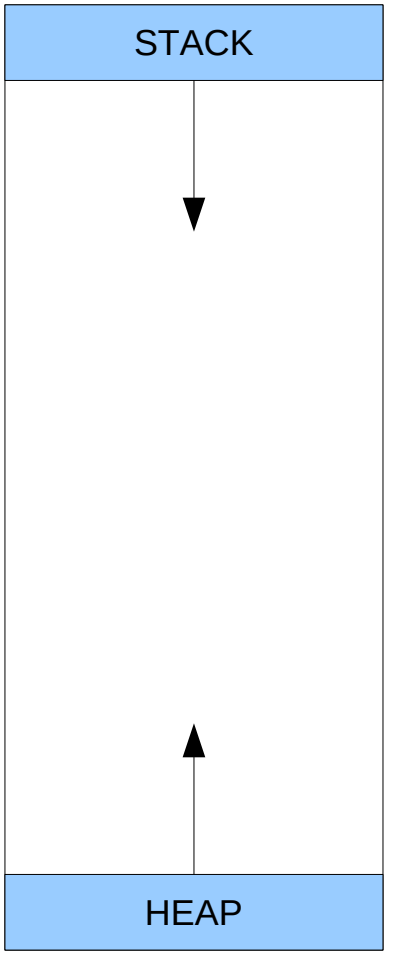
- Los nombres (identidades, identificadores) se usan ampliamente como abstracción
  - Cuando una entidad tiene más de un nombre o un nombre con varias representaciones hay problemas

Lección:

- No confiar en los nombres recibidos del usuario, convertirlos a la representación estándar usada
- No hacer decisiones basado en nombres a nivel de la aplicación sino utilizar al SO para ello

Manejo de Memoria

- El runtime stack usando las direcciones de memoria más alta
- Contiene los *stack frames* del proceso que se está ejecutando en el *call stack*
- Un stack frame contiene info como direcciones de retorno, variables locales y argumentos
- El *heap* comienza desde las direcciones más bajas



Buffer Overruns

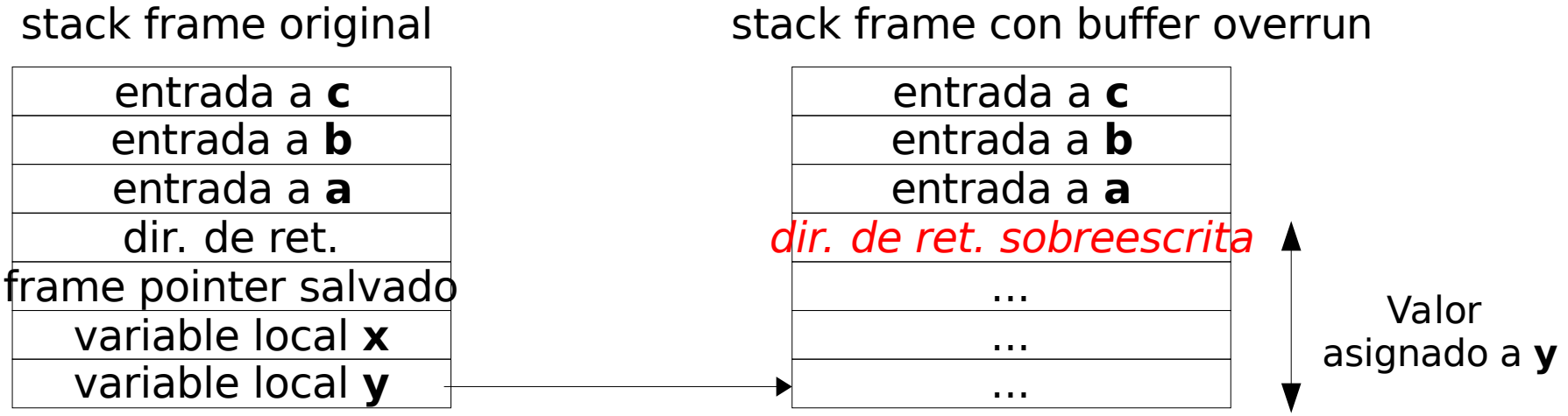
- Si el valor asignado a una variable excede el tamaño del buffer asignado, ocurre un *buffer overrun (overflow)*
- Los lugares de memoria no asignados a esta variable son sobreescritos
- Así pueden escribirse los valores de otras variables
- Han sido el origen de vulnerabilidades de seguridad desde hace un tiempo

Buffer Overrun (Ejemplo)

- Bug del finger (worm de 1988)
  - buffer overrun en el demonio de finger
- Heap buffer overflow in the TFTP protocol handler in cURL 7.19.4 to 7.65.3. (CVE-2019-5482)
- TRENDnet ProView Wireless camera TV-IP512WN 1.0R 1.0.4 is vulnerable to an unauthenticated stack-based buffer overflow in handling RTSP packets. (CVE-2020-12763)

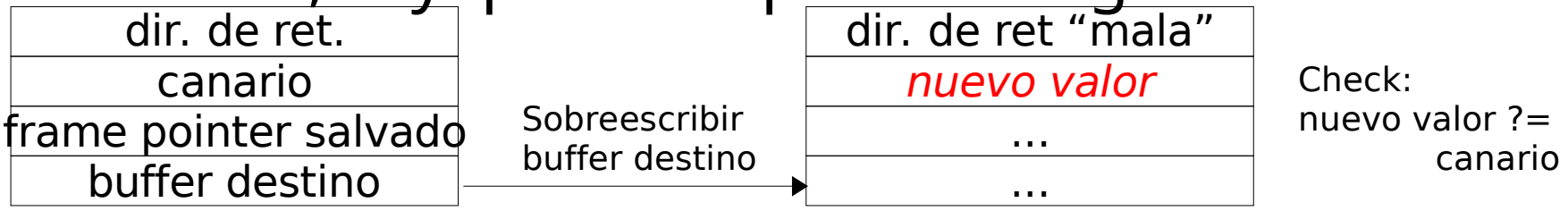
Stack Overruns

- Los ataques de buffer overrun en el call stack se conocen como stack overruns.



Stacks no ejecutables y "canarios"

- Hay arquitecturas con stack no ejecutable
- El software que lo necesite, no funcionará más
- Intentos de cambiar la dirección de retorno pueden detectarse, mediante el uso de "canarios" justo antes de dicha dirección
- En ese caso, hay que recompilar el código



Heap Overruns

- Otros ataques de overrun son más difundidos
- El heap es el área de memoria asignada dinámicamente por la aplicación
- Para tomar control de la ejecución, un atacante debe sobreescribir algún parámetro que influencie en el control o el flujo de datos
- Ataca los punteros a funciones

Confusión de tipos

- Los programas escritos en un lenguaje *type safe* no pueden acceder a memoria de forma inapropiada
- Un ataque de confusión de tipos manipula la estructura de punteros para que un tipo con un tag de clase distinta acceda al mismo lugar
- Son tipos de ataques raros, pero ocurren

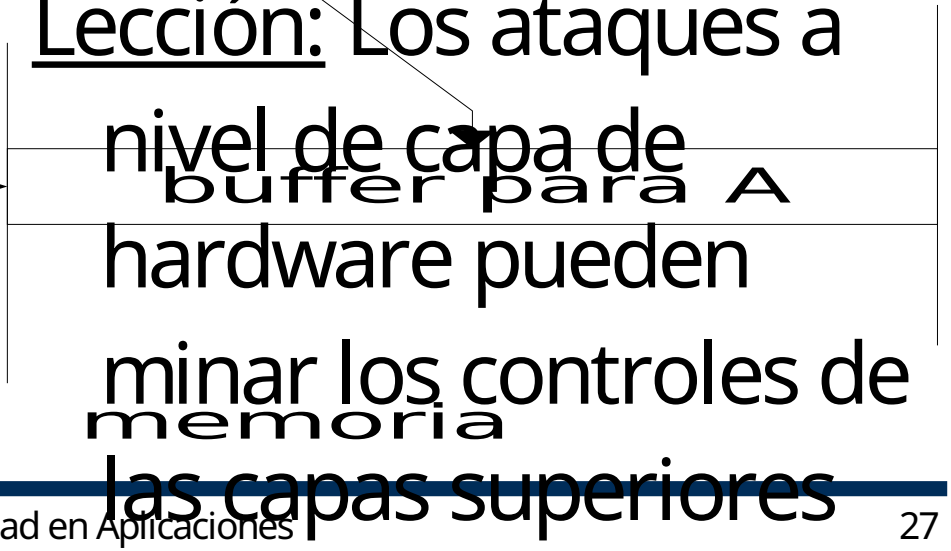
Confusión de tipos (2)

Tabla de referencias

objeto	tipo
A	Tr
X	Un

- Objeto confiable A vs. no confiable *X*
- Ataque en Java para celulares

<u>Lección:</u> Los ataques a nivel de capa de hardware pueden minar los controles de las capas superiores



Datos y código

- Pueden ser importantes abstracciones al valorar la seguridad de un sistema
- Cuando la entrada se toma, y los datos se ejecutan, se rompe la abstracción y los ataques se hacen posibles

Ejemplo:

El bug de rlogin: `rlogin [-l user] [-f] <máquina>`

La opción '-f' fuerza el ingreso

`rlogin -l froot máquina`

Inserción de SQL

- SQL es sumamente usado para hacer consultas a BD
- Las aplicaciones pueden procesar consultas SQL para acceder una BD desde una página web

<u>Lección:</u>

No tratar de adivinar cuáles entradas están mal. Definir solamente las que son aceptadas, p.ej. usando regexp

Condiciones de carrera

- Pueden ocurrir cuando múltiples procesos acceden datos compartidos (archivos, memoria)
- El resultado depende de la secuencia de acceso a dichos datos
- Este problema se conoce en la literatura como TOCTTOU (*time of check to time of use*)

Condiciones de carrera (2)

Pueden dividirse en 2 categorías dependiendo de quién causa la interferencia:

- Por procesos no confiables, se denominan problemas de secuencia
  - Son condiciones causadas por procesos que se meten dentro de la ejecución del nuestro
- Por procesos "confiables", denominados problemas de *locking*
  - Son condiciones causadas por procesos ejecutando el "mismo" programa

Problemas de secuencia

- En general, debe verificarse el código por cualquier par de operaciones que puedan fallar si se ejecuta código arbitrario entre ellas
  - Notar que cargar, modificar y salvar una variable compartida no es un proceso atómico
- Problemas característicos: archivos temporales

Problemas de locking

- Hay situaciones en las cuales un programa debe asegurar que tiene derechos exclusivos sobre algún recurso
- Cualquier sistema que haga locking debe lidiar con los problemas estándar de los locks (deadlocks, livelocks, etc.)
- Típicamente en Unix se utiliza la existencia de un archivo como lock, por ser portable

Condiciones de carrera

Lección:

Una transacción atómica es una abstracción de una operación que debería ejecutarse como una unidad.

Defensas

- Debemos tratar de identificar patrones generales
- En el caso de software inseguro, el patrón que se repite es el de abstracciones de programación como *variable, array, integer*
- Los problemas de la seguridad en el software pueden ser direccionados en la arquitectura del procesador, lenguajes de programación, etc.

Prevención: Hardware

- Muchos ataques de BO sobreescriben información de control
- Estos ataques pueden ser prevenidos mediante el uso de características del hardware para proteger la información de control
- El procesador de Intel Itanium tiene un registro separado para la dirección de retorno

Prevención: *Type safety*

- Podemos prevenir bugs en el software mediante el uso de un lenguaje de programación que nos prevenga de hacer errores
- La seguridad en los tipos garantiza la ausencia de errores no atrapados
- Ejemplos son Java y C#
- Se puede garantizar mediante el chequeo estático y en tiempo de ejecución

Prevención: Funciones más seguras

- Siendo un desarrollador programando en C/C++ se debe evitar escribir código susceptible a BO
- C es tristemente célebre por las funciones de manejo de strings no seguras, como `strcpy, sprintf, gets`
- Tomemos por ejemplo la especificación de strcpy:

```
char *strcpy (char *strDest, const char *strOrig)
```

Usando funciones más seguras

- Hay una excepción si el buffer de origen o destino son null
- El resultado es indefinido si los strings no terminan en '\0'
- Es recomendable reemplazar las funciones por otras en las cuales se pueda especificar la cantidad de chars, ej: `strncpy, snprintf, fgets`
- No erradican BO por sí solas !

Detección: *Code Inspection*

- La inspección de código manual es tediosa, y tendiente a los errores por lo tanto es deseable alguna forma de automatización
- Herramientas de este tipo exploran el código en búsqueda de errores potenciales
- Es bueno para detectar tipos de problemas conocidos pero no garantiza que no hay vulnerabilidades

Algunas herramientas

- Estas son algunas de las herramientas más usadas:
  - BOON – Buffer Overrun Detection
  - CodeWizard
  - FlawFinder
  - Illuma
  - ITS4
  - MOPS

Detección: *Testing*

- El testeo es parte integral del desarrollo de software, normalmente será usada para demostrar la correctitud de la funcionalidad
- En el testing de seguridad, la situación difiere algo dado que tenemos que mostrar la correctitud de la funcionalidad de seguridad
- Esto implica tener alguna idea de las amenazas potenciales

Testing

- El testeo no puede probar la ausencia de errores

Lección:

No se necesita el código fuente para saber cómo se utiliza la memoria o para chequear si las entradas se verifican correctamente.

Testing (2)

- Una técnica usada es la de *data mutation*
- Esto envía entradas mal formadas a las interfaces de los programas. Se testea como se manejan las entradas inesperadas
- El software puede caerse si se intenta crear un recurso que ya existe
- Para los datos, los casos de test deben incluir los bordes, los tipos distintos, tamaños, etc.

Atenuación: Mínimo privilegio

- El principio de mínimo privilegio aplica dos veces
- Al escribir código, ahorrar el requerir privilegios para ejecutar el código
- Si éste código se ve comprometido, el daño es limitado

Mínimo privilegio

- También debe usarse nuevamente al instalar nuevos sistemas
- No dar a los usuarios mas derechos de acceso de los necesarios y no activar las opciones que no se necesitan

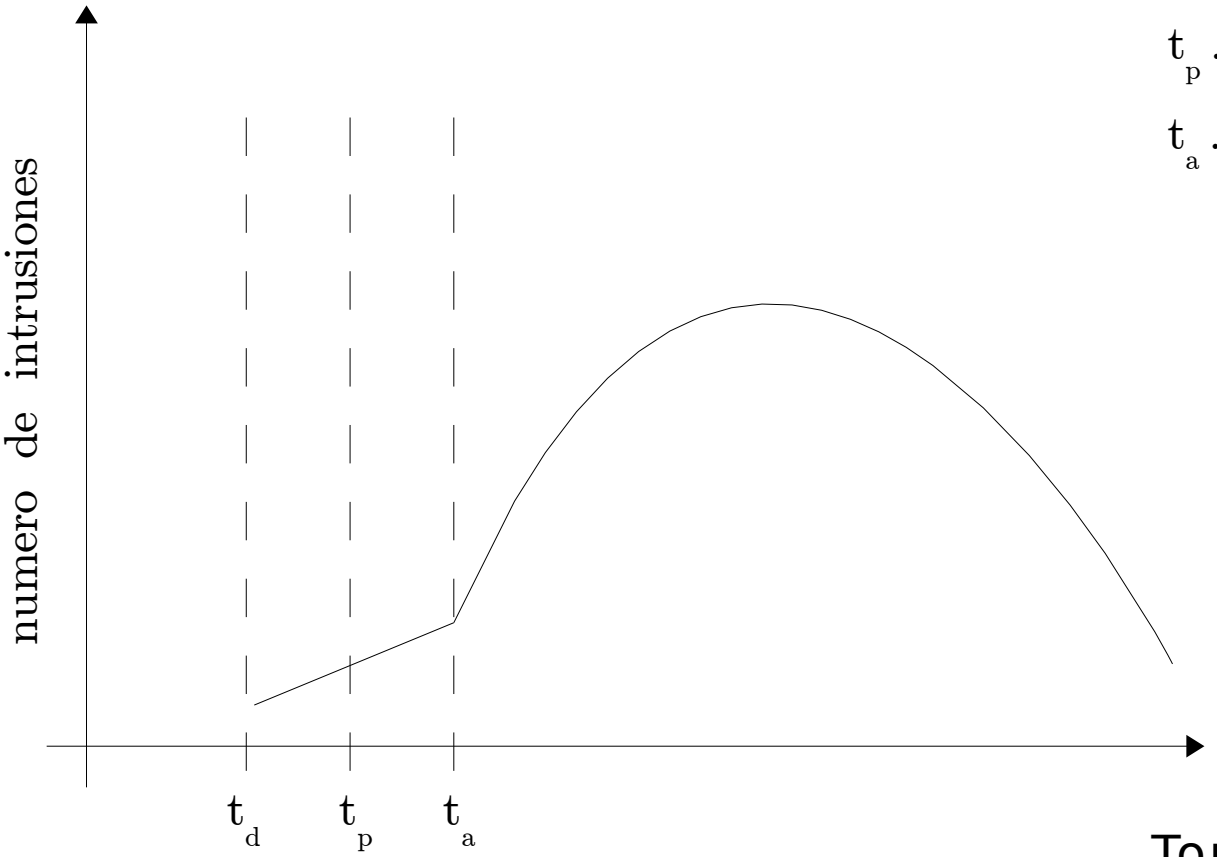
Lección:

El software debe ser instalado de forma que los usuarios activen las características que necesitan, no al revés

Reacción: Manteniéndose al día

- Cuando se descubre una vulnerabilidad, el código afectado debe arreglarse
- La nueva versión debe ser testeada, se debe hacer el patch correspondiente
- ...y por último, los usuarios deben instalarlo
- Por ello, no son sólo los vendedores de software los que deben actuar
- También lo debe hacer la comunidad de usuarios

Ciclo de vida de las intrusiones



t_d ... tiempo de descubrimiento

t_p ... tiempo salida patch

t_a ... tiempo script de ataque

Tomado de Aurbaugh, IEEE

Bibliografía y material de referencia

- D. Gollman, *Computer Security*, Wiley, 2006
- W. Stallings, *Cryptography and Network Security,* Prentice Hall, 2006.
- R. Anderson, *Security Engineering – A Guide to Building Dependable Distributed Systems,* Wiley, 2001
- Jesse Ruderman, *Race conditions in security dialogs*, http://www.squarefree.com/2004/07/01/race-conditions-in-security-dialogs , 2004