Una solución estática al problema de Buffer Overflow

Javier Echaiz* Rafael B. García Jorge R. Ardenghi

Laboratorio de Investigacion de Sistemas Distribuidos (LISiDi)
Departamento de Ciencias e Ingenier a de la Computacion
Universidad Nacional del Sur
Bah a Blanca - Buenos Aires - Argentina
e-mail: {je,rbg,jra}@cs.uns.edu.ar

Resumen

El Bu er Over ow es desde hace mas de una decada el mayor de los problemas que amenaza la seguridad en sistemas. La clave del exito para tratar de mitigar sus efectos parece encontrarse en el analisis estatico. Es nuestra intencion desarrollar un sistema estatico (que extiende a LCLint) de deteccion de posibles problemas de esta naturaleza analizando el codigo fuente de los programas de aplicacion. Una de las mayores ventajas de un sistema de este tipo es que se pueden evitar problemas de seguridad antes de que los programas se encuentren en funcionamiento.

Palabras Clave: bu er over ow, seguridad en sistemas, sistemas operativos.

1 Introduccion

Durante el año 2000 dos estudios [1, 2] indicaron que la mayoría de los ataques no provienen de problemas de seguridad recientemente descubiertos sino de viejos y conocidos problemas. El CERT y la NSA señalaron que los ataques por explotación de buffer overflow son hoy en día el mayor problema de seguidad [3] y que seguirán siendo un problema importante durante los próximos 20 años [4].

Los programas escritos en lenguaje C son particularmente susceptibles a ataques por buffer overflow, pues en las consideraciones de diseño de dicho lenguaje se priorizó espacio y performance sobre seguridad. Como consecuencia de este diseño, C permite manipular punteros sin ningún tipo de chequeo de límites, característica peligrosa por naturaleza debido a que muchas librerías estándard de C incluyen muchas funciones que se tornan "inseguras" si no son utilizadas cuidadosamente. La importancia del problema reside en el hecho de que la mayor parte de los programas críticos de seguridad están escritos en este lenguaje.

Una posible solución al problema es emplear mecanismos en tiempo de ejecución para mitigar los riesgos asociados al persistente buffer overflow. Sin embargo, estas técnicas no están siendo suficientemente empleadas como veremos en la próxima sección. Nuestra propuesta es la de detectar posibles vulnerabilidades en buffers mediante el análisis del código fuente de la aplicación.

*Becario de la Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, Argentina.

Este trabajo involucra una serie de ideas provenientes de diversas áreas de la computación, incluyendo análisis de programas, teoría, compiladores y seguridad en sistemas. Esta nueva herramienta tendrá en cuenta los siguientes tres principios: (1) explotar comentarios semánticos agregados al código fuente para controlar propiedades interprocedurales; (2) emplear técnicas estáticas que logren buena performance y escalabilidad; (3) emplear heurísticas sobre *loops*, de forma tal que sea posible analizar loops presentes en programas típicos.

2 Tecnicas Dinamicas de Defensa y Ataque

El ataque más simple y conocido de buffer overflow es el *stack smashing* [5]. Consiste en sobrescribir un buffer en el stack con el objetivo de reemplazar la dirección de retorno. Cuando la función ejecuta el `return`, en lugar de saltar a la dirección de retorno original, el control pasará a la dirección que fue colocada en el stack por el atacante. Esto concede al atacante la capacidad de ejecutar código arbitrario. Como mencionamos en la sección previa, C brinda acceso directo a memoria y aritmética de punteros sin ningún tipo de control de límites. Más grave aún, C incluye funciones inseguras en su librería estándar, como `gets`, que permiten copiar una cantidad ilimitada de información introducida por el usuario en un buffer de tamaño fijo sin controlar los límites de dicho buffer. Muchas veces estas funciones reciben buffers pasados como parámetro mediante el stack, facilitando la tarea del atacante, quien unicamente debe ingresar una entrada de longitud mayor a la del buffer y codificar un programa binario en dicha entrada. El famoso Internet Worm de 1988 [7] aprovechaba esta vulnerabilidad presente en el `fingerd`. Existe otro tipo de ataque de buffer overflow, donde se emplea el heap en lugar del stack. Este ataque es más sofisticado que el anterior, pues la mayoría de los sistemas operativos no permiten saltar a direcciones cargadas desde el heap o hacia código almacenado en el heap.

Existen varias implementaciones que atacan el problema en tiempo de ejecución. StackGuard [8] es un compilador que genera binararios que incluyen código diseñado para prevenir ataques del tipo stack smashing. Este compilador ubica un valor especial a continuación de la dirección de retorno y chequea antes de saltar que ésta no haya sido modificada. Baratloo, Singh y Tsai describen dos métodos que actúan en tiempo de ejecución: uno reemplaza las funciones inseguras de la librería estándard de C por versiones seguras; el otro modifica los ejecutables con la intención de chequear la dirección de destino alojada en el stack antes de saltar.

Otra técnica dinámica es la denominada *software fault isolation* (SFI), mediante la cual se insertan instrucciones antes de las operaciones de memoria para prevenir el acceso a regiones fuera de rango [10]. Esta técnica ofrece escasa protección contra ataquetes típicos de buffer overflow dado que no evita que un programa escriba en la dirección del stack donde se almacena la dirección de retorno. Existen versiones de SFI que agregan controles más extensivos alrededor de las instrucciones potencialmente peligrosas para restringir de forma más general el comportamiento de los programas y así lograr una mayor protección. Algunos ejemplos de sistemas que utilizan esta técnica son Janus [11], Naccio [12] y Generic Software Wrappers [13].

El sistema operativo puede prevenir algunos tipos de ataque por buffer overflow haciendo que el código y los datos se encuentren en segmentos de memoria separados, donde el segmento de código es *read only* y no se pueden ejecutar instrucciones almacenadas en el segmento de datos. Esto no elimina totalmente el problema, pues el atacante todavía puede sobrescribir una dirección almacenada en el stack y hacer que el programa salte hacia cualquier dirección del segmento de código. Peor aún, si el programa utiliza librerías compartidas, el atacante puede saltar a una dirección (en el segmento de código) que puede ser utilizada maliciosamente (e.g.,

una llamada a `system`). Es claro entonces que esta técnica no solo no soluciona el problema real sino que también evita la ejecución de código automodificable legítimo.

3 Prototipo de una Solucion Estatica

Sin considerar la disponibilidad de las técnicas vistas que actúan en tiempo de ejecución, es evidente que no son actualmente una solución viable al problema de buffer overflow. Probablemente esto se debe a la falta de conciencia acerca de la severidad del problema y a la falta de soluciones prácticas. Por otra parte es cierto que en algunos ambientes de trabajo, las soluciones en tiempo de ejecución son inaceptables. Por ejemplo StackGuard reporta un overhead en la performance del 40% [6]. El otro problema real que presentan este tipo de soluciones es que convierten un ataque de buffer overflow en uno de *denial of service*, pues generalmente no pueden hacer más que abortar la ejecución del programa frente a un intento de explotación de buffer overflow.

El control estático soluciona estos inconvenientes detectando posibles vulnerabilidades antes de que el programa se ejecute. Detectar vulnerabilidades de buffer overflow analizando código fuente es evidentemente un problema no decidible[1]. La intención de éstos métodos no es la de tratar de verificar que un programa no tenga vulnerabilidades debido a buffer overflow, sino la de detectar la mayoría de los posibles problemas de este tipo antes de que el programa se encuentre en funcionamiento. Podemos tratar de conseguir entonces buenos resultados a pesar de que esta técnica sea incompleta y no sensata. Esto significa que nuestra herramienta ocasionalmente generará falsas alarmas (falsos positivos) y fallará esporádicamente en detectar algunos problemas reales (falsos negativos). A pesar de esto, nuestro objetivo es desarrollar una herramienta capaz de producir resultados útiles sobre programas reales utilizando un esfuerzo razonable, dejando a cargo del programador el chequeo "a mano" de un número reducido de posibles vulnerabilidades.

Nuestra herramienta estática ampliará las capacidades de LCLint [9], una herramienta de chequeo estático que emplea anotaciones en el código fuente. LCLint es capaz de detectar entre otras, violaciones a ocultamiento de información, referencias a punteros nulos, problemas en la instanciación de parámetros y modificación inconsistente de variables globales.

El usuario agregará comentarios semánticos (de aquí en adelante llamados *anotaciones*) al código fuente tanto de la aplicación como de las librerías estándard. Las anotaciones describirán las intenciones y asunciones del programador, las cuales pasarán desapercibidas por el compilador de C pero serán reconocidas como entidades sintácticas por nuestra herramienta. Utilizaremos entonces la siguiente notación (a grandes rasgos) para especificar una anotación dentro del código fuente:

$$anotación := \quad *\,\$\; requerimiento \;\$\,*$$

Podemos por ejemplo utilizar $*\,\$\; notnull \;\$\,*$ para indicar que el valor de una variable debe ser distinto de `NULL`.

No es difícil imaginar que el *requerimiento* debe ser suficientemente expresivo como para poder realizar una detección efectiva de vulnerabilidades de buffer overflow. LCLint contempla un número pequeño de casos posibles, y será necesario por ejemplo saber cuanta memoria fue asignada a un determinado buffer. Por lo tanto deberemos extender LCLint para que soporte un

[1] Trivialmente podemos reducir el problema de la detención al problema de detección de buffer overflow agregando instrucciones que causen un buffer overflow antes de todas las instrucciones `halt`.

lenguaje de anotaciones más expresivo mientras mantenemos la simplicidad de sus comentarios semánticos.

El programador podrá explicitar el estado de una función mediante precondiciones y poscondiciones utilizando cláusulas[2]. Podemos utilizar dichas cláusulas para describir las asunciones acerca de los buffers que se pasan como parámetros a funciones y controlar que se cumplan las restricciones cuando la función retorna. Nuestra herramienta utilizará (inicialmente) cinco tipos de asunciones y restricciones: *minSet*, *maxSet*, *minRead*, *maxRead* y *nullterminated*.

Cuando *minSet* y *maxSet* se encuentran presentes en una cláusula como requerimiento, indicarán los límites iniciales y finales (respectivamente) que puede soportar un buffer de forma segura (e.g., del lado izquierdo de una asignación (`lvalue`)). Por ejemplo, consideremos una función que recibe dos parámetros: un arreglo `a` y un entero `i` y que tiene la precondición *maxSet*(`a`) >=`i`. El análisis asume que al principio del cuerpo de la función `a[i]` puede ser utilizado como `lvalue`. Si `a[i+1]` fuese usado antes de cualquier modificación a `a` o a `i`, LCLint generaría un *warning* ya que las precondiciones de la función no son suficientes para garantizar que `a[i+1]` puede utilizarse de forma segura como `lvalue`. Los arreglos en C comienzan en 0 por lo tanto la declaración `char buffer[MAXSIZE]` genera las restricciones *maxSet*(`buffer`) =`MAXSIZE`−1 y *minSet*(`buffer`) = 0.

Análogamente, las restricciones *minRead* y *maxRead* aplicadas a un buffer indican los límites de lectura seguros, mínimo y máximo respectivamente. Obviamente *maxRead*(`buffer`) siempre será menor o igual que *maxSet*(`buffer`). En aquellos casos donde existan algunos elementos de un buffer todavía no inicializados *maxRead*(`buffer`) puede ser menor que *maxSet* (`buffer`).

Cuando LCLint encuentra una llamada a función, controla que las precondiciones implicadas por la cláusula de requerimientos sean satisfechas antes de realizar dicha llamada. Para el ejemplo de *maxSet*(`a`) >=`i` devolvería un warning si no puede determinar con certeza si el arreglo pasado en `a` puede alojar al menos tantos elementos como el valor pasado en `i`.

Si tenemos *nullterminated* en una anotación, podremos monitorear que el buffer de almacenamiento termine con el caracter `NULL`. Muchas funciones de la librería estándard de C requieren la presencia del terminador nulo en los *strings* que reciben como parámetros, si este requerimiento no se satisface pueden apararecer vulnerabilidades por explotación de buffer overflow.

Muchos problemas de buffer overflow aparecen en funciones de librería, como por ejemplo `strcpy`. Parte de nuestra tarea será entonces agregar anotaciones a librerías estándard para así poder detectar inconvenientes incluso antes de agregar anotaciones en el programa de aplicación.

3.1 Implementación y Trabajos Futuros

Nuestra herramienta será implementada en C bajo Solaris combinando técnicas tradicionales de construcción de compiladores con generación y resolución de restricciones. Los programas serán analizados a nivel de función y los análisis interprocedurales se efectuarán mediante la información contenida en las anotaciones. En el caso de que se obtengan resultados promisorios implementaremos una versión de esta herramienta que haga uso de un sistema subyacente de DSM[3] con la intención de lograr una mayor performance, especialmente en el procesamiento de código fuente de aplicaciones de gran tamaño (decenas o centenas de miles de líneas de código).

[2]LCL, predecesor de LCLint, incluía una notación general de pre y poscondiciones mediante cláusulas.

[3]Nuestro grupo de investigación se encuentra desarrollando un sistema de DSM que cuenta con un modelo de consistencia scope y un protocolo de coherencia basado en home.

4 Conclusiones

Los mecanismos estáticos de detección de problemas de buffer overflow desarrollados previamente indican que éste es un buen curso de acción frente al persistente problema de buffer overflow que atenta contra la seguridad en sistemas. Es claro que las técnicas estáticas no son ni sensatas ni completas (no es una solución óptima) pero somos optimistas con respecto a los posibles resultados que obtendremos mediante la aplicación de la herramienta descripta brevemente en este trabajo. Este sistema será capaz de evitar muchos de los problemas de seguridad antes de que los programas se encuentren en funcionamiento, evitando así que el usuario tenga que aplicar un *patch* a un programa luego de que el atacante aprovechó una vulnerabilidad.

El principal objetivo de esta herramienta es mejorar la seguridad de los sistemas en general y el problema de buffer overflow en particular, pero además, agregar anotaciones al código fuente redundará en un mejor entendimiento del funcionamiento de los programas y en una mejor capacidad para su mantenimiento.

Referencias

[1] C. Cowan, C. Pu. Bu er Over ows: Attacks and Defenses for the Vulnerability of the Decade. *DARPA Information Survivability Conference*, enero 2000.

[2] D. Wagner, J. Foster. A First Step Towards Automated Detection of Bu er Overrun Vulnerabilities. *Network and Distributed Security Symposium*, febrero 2000.

[3] R. Pethia. Bugs in Programs. *SIGSOF Symposium on the Fundations of Software Engineering*, noviembre 2000.

[4] B. Snow. Future of Security. *IEEE Security and Privacy*, mayo 1999.

[5] Aleph One. Smashing the Stack for Fun and Pro t. *BugTraq archives*, 1996.

[6] C. Cowan, S. Beattie. Protecting Systems from Stack Smashing Attacks with StackGuard. *Linux Expo.*, mayo 1999.

[7] E. Spa ord. The Internet Worm Program: An Analysis. *Purdue Tech Report 832*, 1988.

[8] C. Cowan, D. Maier. Automatic Detection and Prevention of Bu er-Over ow Attacks. 7^{th} *USENIX Security Symposium*, enero 1998.

[9] D. Evans, J. Guttag. LCLint: A Tool for Using Speci cations to Check Code. *SIGSOF Symposium on the Fundations of Software Engineering*, diciembre 1994.

[10] R. Wahbe, S. Lucco. E cient Software-Based Fault Isolation. 14^{th} *ACM Symposium on Operating Systems Principles*, 1993.

[11] I. Goldberg, D. Wagner. A Secure Environment for Untrusted Helper Applications: Con ning the Wily Hacker. 6^{th} *USENIX Security Symposium*, julio 1996.

[12] D. Evans, A. Twyman. Flexible Policy-Directed Code Safety. *IEEE Symposium on Security and Privacy*, mayo 1999.

[13] T. Fraser, L. Badger. Hardening COTS Software with Generic Software Wrappers. *IEEE Symposium on Security and Privacy*, mayo 1999.