TÉCNICAS DE OFENSA Y DEFENSA A LOS FALLOS POR CORRUPCIÓN DE MEMORIA

David Mora Rodríguez*
Mario Muñoz**

Recibido: 18/11/2010
Aceptado: 19/10/2011

RESUMEN

Las técnicas de ataque a aplicaciones por corrupción de memoria aprovechan las debilidades de los programas para obtener ejecución de código arbitrario. Estos fallos de programación han sido utilizados por diferentes ataques desde la década de los ochenta. Este documento presenta las diferentes técnicas que puede utilizar un atacante para lograr su objetivo y las precauciones que debe tener un desarrollador de aplicaciones, para evitar que su programa esté expuesto a vulnerabilidades que permitan ejecutar ataques por corrupción de memoria. Los fabricantes de sistemas operativos y compiladores introdujeron diferentes mecanismos de defensa para proteger las aplicaciones. Estos mecanismos no son excluyentes y cada uno tiene sus propios objetivos de diseño para añadir nuevas capas de seguridad.

Palabras clave: ASLR, Corrupción de memoria, NX/XD, Sistemas operativos, Programación orientada a retornos

* Mg. Seguridad de la Información (Universitat Oberta de Catalunya), Estudiante de Maestría en Ingeniería (Universidad de Antioquia), Dirección Calle 67 Número 53 – 108, Teléfono 2195560, dmora@udea.edu.co
** M. Sc. Ingeniería Eléctrica (Universidade de Sao Paulo, Brasil), aspirante a doctorado (Universidade de Sao Paulo, Brasil), profesor en Ingeniería Electrónica (Universidad de Antioquia), Teléfono 2198565, mamunoz@udea.edu.co

MEMORY CORRUPTION FAILURES ATTACK AND DEFENSE TECHNIQUES

ABSTRACT

Attack techniques against memory corruption applications take advantage of the programs weakness for obtaining execution of arbitrary code. These programming failures have been used for several attacks since the 80's. This document shows several techniques to be used by an attacker in order to achieve his objectives and the precautions an application developer should have for preventing the program to be exposed vulnerable situations which may allow having attacks for memory corruption. Manufacturers of operating systems and compilers introduced several defense mechanisms to protect applications. These are not excluding mechanisms and each one of them has its own design objectives for adding new security layers.

Key words: ASLR; memory corruption; NX/XD; operating systems; return-oriented programming.

INTRODUCCIÓN

La corrupción de memoria no es un problema reciente. En [1] se hace referencia a un informe desarrollado por la fuerza aérea de los Estados Unidos en 1972. El reporte indicaba que fabricando adecuadamente las entradas de usuario era posible manipular los apuntadores para producir una fuga de información.

Posteriormente, en 1988, se liberó el gusano "Morris-Worm" (primer programa de autopropagación liberado en Internet). Este programa introdujo lo que hoy se conoce como ataques por corrupción de memoria y el código malicioso de autopropagación. Este documento pretende explicar las diferentes técnicas de ataque a aplicaciones, los mecanismos de seguridad vigentes y la forma en que un atacante puede evadirlos.

La primera parte de este documento presenta las técnicas de ataque que han demostrado ser efectivas. Posteriormente, se presenta una generalización de las técnicas que permiten determinar el impacto de un fallo de programación con respecto a la posibilidad de obtener ejecución de código arbitrario. También se presenta una explicación sobre el motivo de estas vulnerabilidades y las acciones que se pueden tomar para disminuir el riesgo que representan estos fallos. Finalmente, se exponen las tendencias de los ataques modernos para evadir las estrategias de protección en los sistemas actuales.

1. ATAQUES CLÁSICOS POR CORRUPCIÓN DE MEMORIA

Los ataques por corrupción de memoria permiten manipular la memoria del programa vulnerable para forzar a la aplicación a comportarse de manera diferente a la que fue programada. Esta sección presenta tres tipos de ataques: desbordamiento de *buffer* en la pila [2], cadenas de formato [3], y desbordamiento de *buffer* en el montículo (del término en inglés "*heap*") [4].

1.1 Desbordamiento de buffer en la pila

En 1996, Levy publicó una explicación detallada sobre la técnica de desbordamiento de *buffer* [2]. El documento generó tal impacto que aún hoy continúa siendo un marco de referencia para otros autores [1, 5, 6].

Los programas de usuario utilizan una región de memoria llamada la pila, donde se almacena información relacionada con el flujo de ejecución del proceso. Dicha información está compuesta por datos locales a funciones, direcciones de retorno y apuntadores relativos para la dirección de datos (también conocidos como apuntadores del marco de la pila).

En las arquitecturas x86 y AMD64, se han reservado dos registros para controlar el funcionamiento de la pila. El primero de estos registros se conoce como el apuntador de la pila *ESP* (del inglés, *stack pointer*) y es el responsable de mantener la referencia al final de la pila; el segundo es el apuntador del marco de pila, gobernado por el registro *EBP* (del inglés, *base pointer*).

En la figura 1, se muestra un esquema para representar el espacio virtual de direcciones de los procesos en los sistemas Linux. Como puede observarse, el apuntador de pila suele ubicarse al final del espacio de memoria de usuario y el espacio de pila "crece" hacia las direcciones bajas donde residen el montículo, los datos globales y el código.

Figura 1. Espacio virtual de direcciones de los procesos, en sistemas GNU/Linux

Fuente propia.

Los datos en la pila utilizan direccionamiento relativo. El CÓDIGO 1 es un ejemplo de la salida de un compilador para un programa escrito en C que defina dos funciones, main y primera_funcion. Las instrucciones 1 y 2 de CÓDIGO 1 son conocidas como el preludio y su propósito es configurar un nuevo marco de pila para ofrecer la capacidad de direccionamiento relativo en los datos locales. El uso del direccionamiento relativo puede observarse en la instrucción 5 de CÓDIGO 1, donde se almacena el valor 0x01 (byte) en una ubicación 256 bytes adelante del EBP. La instrucción 4 de CÓDIGO 1 se encarga de reservar memoria en el interior de la pila.

CÓDIGO 1

COMPORTAMIENTO DE LA PILA

```
1   primera_funcion:
2   pushl %ebp
3   movl %esp,%ebp
4   subl $264,%esp
5   movb $1,-256(%ebp)
6   .L2:
7   leave
8   ret
9   main:
10  pushl %ebp
11  movl %esp,%ebp
15  jmp .L3
16  .L3:
17  leave
```

El otro segmento de código similar en las dos funciones es la forma en que finalizan y se puede observar en las instrucciones 7,8 y 17,18. La responsabilidad de recuperar la pila depende de la convención utilizada para llamar una función. A. Fog en [7] presenta diferentes convenciones para el llamado a funciones.

El uso de la pila para almacenar datos locales genera una debilidad importante: el hecho de almacenar los datos de usuario contiguos a los datos de control de la pila, permite que un error de programación sobrescriba la información de control con datos de usuario. El CÓDIGO 2 muestra un ejemplo de esta situación.

CÓDIGO 2

PROGRAMA VULNERABLE A DESBORDAMIENTO DE BUFFER EN LA PILA

```
1  #include <iostream.h>
2  int main() {
3  char buffer[128];
4  cin>> buffer;
5  return 0;
6  }
```

En el CÓDIGO 2, los datos locales a *main* (los elementos del arreglo llamado "*buffer*") se encuentran almacenados en la pila. El operador (">>") sobrecargado en *istream* no hace una validación de límites sobre la variable de destino. Por lo tanto, si el usuario ingresa una cantidad de datos superior a 128 bytes comenzará a sobrescribir datos en la pila, por ejemplo, el apuntador base de la pila de la función anterior y la dirección de retorno almacenada por la instrucción *CALL*.

Un usuario puede fabricar la entrada del programa de manera que se escriba en la dirección de retorno almacenada, una dirección con las instrucciones que se desea ejecutar. De este modo, cuando la función *main* ejecute la instrucción "*ret*" se cargará en *EIP* una dirección controlada por el usuario forzando la ejecución de código arbitrario.

1.2 Cadenas de formato

El CÓDIGO 3 muestra un programa vulnerable que utiliza cadenas de formato.

CÓDIGO 3

PROGRAMA VULNERABLE POR CADENAS DE FORMATO MAL UTILIZADAS

```
1. #include <stdio.h>
2. int main(int arc, char ** argv) {
3.  printf(«Este programa imprime el primer
    argumento ingresado por el usuario\n»);
4.
5.     if (argc != 2)
6.          return -1;
7.  printf(argv[1]);
8.  printf(«\n»);
9.     return 0;
10. }
```

El problema del CÓDIGO 3 radica en que la cadena de formato de la línea 7 puede ser controlada por el usuario. En 1999, Tymm Twillman envió un correo al equipo de desarrollo de la biblioteca estándar de C [3] ilustrando cómo se puede aprovechar este tipo de vulnerabilidades para obtener ejecución de código arbitrario. Para obtener ejecución de código arbitrario, se utiliza el modificador de formato "%n". En [8], se presenta la técnica de cómo se puede obtener ejecución de código arbitrario de este modo.

1.3 Desbordamientos en el montículo

El asignador de memoria ofrecido por la biblioteca estándar de C ofrece su interfaz a través de las funciones *calloc(), malloc(), free()* y *realloc()*. Cada desarrollador de esta biblioteca implementa de manera diferente las estructuras de datos y los mecanismos que utiliza para administrar el montículo. La biblioteca estándar de C de GNU, utiliza un asignador de memoria conocido como *ptmalloc*, cuyo código está basado en el asignador de memoria desarrollado por Doug Lea [9].

Los pedazos del asignador de memoria desarrollado por Doug Lea llevan consigo información, antes y después de los datos. El principal objetivo es permitir que dos pedazos sin utilizar puedan unirse en un pedazo más largo, reduciendo la fragmentación.

Cuando se realiza un desbordamiento en el montículo, se pueden modificar los datos de control de los pedazos. Estos datos de control contienen apuntadores para la inclusión de pedazos libres en listas doblemente enlazadas. Cuando se hace un llamado a free() utilizando un pedazo que tiene un pedazo adyacente P libre, el algoritmo de la función hace un llamado al macro del CÓDIGO 4.

CÓDIGO 4

MACRO UNLINK

```
1 #define unlink( P, BK, FD ) {
2 BK = P->bk;
3 FD = P->fd;
4 FD->bk = BK;
5 BK->fd = FD;
6 }
```

Si el pedazo contiguo P ha sido malformado, se puede escribir en posiciones arbitrarias de memoria y obtener ejecución de código arbitrario.

La técnica del macro *unlink* no tiene una aplicación en las versiones posteriores a 2004 de la biblioteca estándar de C de GNU. Sin embargo, presenta un precedente de que la escritura en posiciones arbitrarias de memoria puede resultar en ejecución de código arbitrario. En [10] se presentan cinco técnicas aplicables al asignador de memoria actual.

2. MECANISMOS DE PROTECCIÓN A ATAQUES POR CORRUPCIÓN DE MEMORIA

Existen dos alternativas para mitigar el riesgo que representan los fallos por corrupción de memoria. El primer enfoque consiste en utilizar mecanismos en el lado del programador para evitar

que las vulnerabilidades existan. El segundo enfoque consiste en otorgar protecciones adicionales desde el sistema operativo, el compilador y las bibliotecas.

Esta sección introduce los mecanismos de defensa utilizados para responder a los ataques por corrupción de memoria. Las protecciones presentadas en esta sección no son exhaustivas; sin embargo, se presentan las técnicas más relevantes y comunes de los sistemas modernos.

2.1 Defensa al lado del programador

Las entradas de usuario comunican las decisiones que este ha tomado. Se debe prestar especial atención al tamaño, la naturaleza y la validez contextual de la entrada. Muchos sistemas almacenan información de control en lugares adyacentes a los datos, por lo que un fallo en el control de la memoria puede permitir la modificación de estos datos de control. Tal es el caso de las direcciones de retorno almacenadas en la pila [2], los datos de control del montículo [4], los apuntadores virtuales [11], apuntadores a otras estructuras [12], entre otros.

Algunos lenguajes son más susceptibles a los ataques por corrupción de memoria. La mayoría de los lenguajes interpretados ofrecen control de límites y detección de desbordamientos en tiempo de ejecución; algunos ofrecen detección de estas situaciones en tiempo de compilación (PERL, JAVASCRIPT). Otros lenguajes (JAVA, .NET, LISP, OCAML) ofrecen una capa de abstracción entre la administración de la memoria de usuario y el programador; esta capa de abstracción puede detectar y prevenir las vulnerabilidades que permiten la corrupción de la memoria.

Es importante anotar que los mecanismos de detección y prevención que ofrecen estos lenguajes tienen un impacto significativo en el rendimiento de las aplicaciones. Esto puede ser tolerable dependiendo de los requerimientos; la decisión final es una pregunta abierta de ingeniería de software. Sin embargo, los programas deben usar las mismas reglas para ejecutar su código; esto quiere decir que pueden existir vulnerabilidades en los intérpretes o los mecanismos que utilizan estos programas. Burch [13] presenta un escenario de ataque por cadenas de formato aplicado al lenguaje PERL, y Shahin [14] presenta una vulnerabilidad por corrupción de memoria de una biblioteca en la máquina virtual JAVA.

Aún en lenguajes que exponen la administración de la memoria, el uso de apuntadores dinámicos [15] puede reducir el riesgo de experimentar corrupción de memoria. La biblioteca estándar de C++ incluye algo conocido como "smart pointers" que facilita el control de datos en el montículo.

Es importante incluir pruebas relacionadas con el comportamiento seguro de la aplicación en el proceso de desarrollo. Las pruebas relacionadas con los fallos por corrupción de memoria fueron clasificadas por Sutton et al. [16] como pruebas de caja blanca, caja gris y caja negra. Las pruebas de caja blanca consisten en la revisión del código fuente para buscar instrucciones que lleven a fallos de seguridad. Las pruebas de caja gris consisten en la revisión del programa utilizando herramientas de ingeniería inversa. Las pruebas de caja negra, también conocidas como "*fuzzing*" consisten en someter la aplicación a entradas pseudoaleatorias para detectar comportamientos anómalos de la aplicación.

2.2 Defensa del sistema operativo

Aunque existen muchos mecanismos de defensa, este documento presenta dos de las técnicas más importantes para prevenir la ejecución de código arbitrario.

La primera técnica fue introducida por AMD con un bit de seguridad conocido como NX (del inglés, No eXecute), que posteriormente Intel comercializó como XD en su procesador pentium 4 (del inglés, eXecuteDisable).

Sin importar el nombre que se dé a esta tecnología, se trata de un bit en las entradas de la tabla de páginas del proceso que, una vez habilitado, hace que el procesador genere excepciones para

prevenir la ejecución de instrucciones en ese espacio de memoria.

El bit NX/XD es una modificación importante al sistema operativo y, por lo tanto, algunos programas no están diseñados para trabajar con esta tecnología.

La segunda técnica es conocida como distribución aleatoria del espacio de direcciones (del inglés, *Adress Space Layout Randomization ASLR*); obtuvo su nombre en 2001, cuando el equipo PAX liberó la primera implementación como un parche del núcleo de Linux.

La técnica se basa en el hecho de que un atacante utiliza direcciones de memoria fijas para lograr la ejecución de código arbitrario. La idea principal es responder a esta estrategia distribuyendo aleatoriamente el espacio de direcciones virtuales de los procesos. Con esta medida, un atacante no puede determinar de manera confiable las regiones del espacio de direcciones del proceso.

Existen diferentes implementaciones de esta tecnología para diferentes sistemas operativos y en algunas situaciones la relación es muchos a uno (diferentes implementaciones para el mismo sistema operativo).

3. OFENSA DE APLICACIONES EN LOS SISTEMAS OPERATIVOS MODERNOS

La técnica de "retorno a biblioteca" [17] es el concepto general que utilizan los atacantes para evadir las restricciones de los privilegios de ejecución en las regiones de datos. Esta técnica es más conocida como "retorno a libc" (del inglés, "returntolibc"), aunque no es necesario que la biblioteca sea libc. La estrategia aprovecha la capacidad de escribir en la pila del proceso para forzar el programa a ejecutar funciones arbitrarias (ubicadas en páginas que no tienen habilitado el bit NX/XD) [17].

Un atacante puede configurar la pila como se muestra en la figura 2, para invocar a *mprotect* y modificar los privilegios del código inyectado.

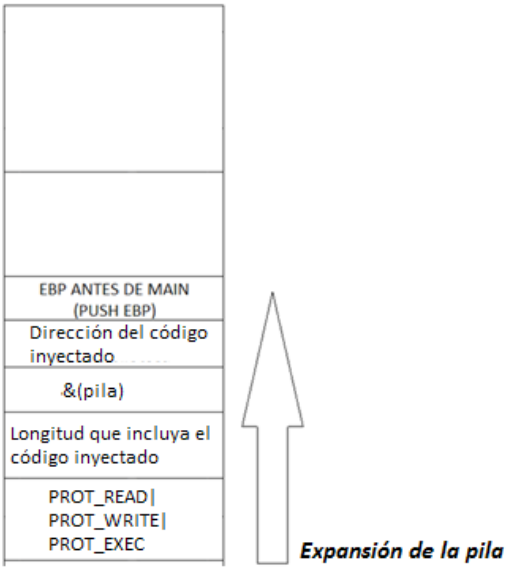

Figura 2. Técnica de evasión de los privilegios de ejecución. Fuente: propia.

La estrategia de *mprotect* puede implementarse en otros sistemas con funciones similares. Sin embargo, Le [5] y Buchanan et al. [18] presentan una estrategia para reutilizar las instrucciones de la aplicación y construir código arbitrario. La técnica es conocida como programación orientada a retornos (del inglés, Return Oriented Porgramming ROP).

Para evadir la distribución aleatoria del espacio de direcciones pueden emplearse diferentes estrategias. Müller [19] hace una recopilación sobre algunas de estas técnicas. Una alternativa consiste en burlar el mecanismo de aleatoriedad utilizando fuerza bruta; esta estrategia tiene como desventaja que puede ser fácilmente detectada y requiere que la aplicación vulnerable pueda recuperarse de un fallo de segmentación.

Las implementaciones deben evitar consumir la entropía del sistema operativo, por lo tanto es posible que existan debilidades estadísticas en la implementación. Whitehouse [20] muestra debilidades importantes de la implementación del sistema operativo Microsoft Windows Vista.

Por la forma en que algunos procesadores manejan la memoria, es posible modificar únicamente los bytes menos significativos de las direcciones. Dependiendo de donde se encuentre el fallo de

seguridad, se puede aprovechar esto para reducir la cantidad de bytes desconocidos y mejorar las posibilidades del atacante. En algunos casos, es posible utilizar esta técnica para tener un ataque determinista. Algunas bibliotecas pueden ser excluidas de esta medida de seguridad. En este caso, es posible utilizar retornos a biblioteca para lograr ejecución de código arbitrario. En las versiones del kernel inferiores a 2.6.20, la biblioteca Linux-gate no era distribuida aleatoriamente y, por lo tanto, un atacante podía utilizarla para lograr su objetivo.

En los sistemas Microsoft, es usual que algunos módulos no sean incluidos en el proceso de distribución aleatoria.

Aunque algunas implementaciones pueden hacer aleatoria la ubicación del código principal del programa, esto no es usual. Para que esto sea posible, el código del programa debe ser compilado con posición independiente. Esto significa que se deben recompilar las aplicaciones para soportar esta característica. Una primera observación podría indicar que siempre se pueden utilizar direcciones en el código del programa; sin embargo, las instrucciones útiles en dicho código suelen ser escasas, debido a que gran parte de la funcionalidad de las aplicaciones se encuentra almacenada en las bibliotecas [19].

Algunos espacios de memoria de la pila pueden contener direcciones de memoria que referencian la pila. En estos casos, puede utilizarse una estrategia de retornar a instrucciones "RET" múltiples veces de manera que se pueda forzar el programa a saltar al código inyectado (utilizando las instrucciones almacenadas en la región de código del programa). La figura 3 muestra la pila con un ataque del tipo retorno a retorno.

La dirección de retorno almacenada en la pila se encuentra en el lugar donde aparece por primera vez 0x08049654 que es una dirección usual para la región del código del programa. En el cuadro de la izquierda se puede observar que la instrucción en dicha dirección es "RET". Por lo tanto, el programa comenzará a retornar hasta que *EIP* sea sobrescrito con 0xBFFFFA09 que es una dirección en el medio de la región con *NOPS*.

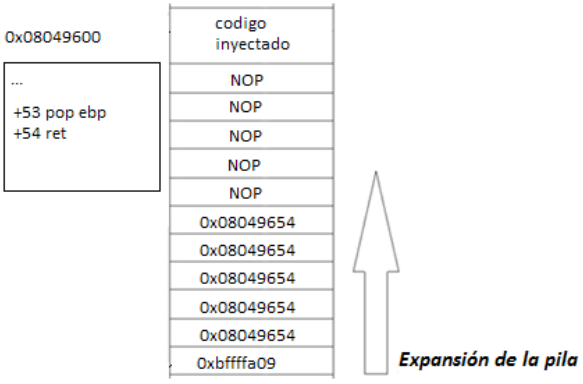

Figura 3. Ataque del tipo retorno a retorno, donde se almacenan direcciones para modificar la ubicación de ESP.

Fuente: propia.

Algunas aplicaciones permiten al atacante asignar grandes bloques de datos en el montículo (esto es usual en ataques al lado del cliente, donde se pueden utilizar lenguajes que utilizan compiladores que permiten este propósito); en estas situaciones se puede forzar la aplicación a utilizar una dirección del montículo que tenga altas posibilidades de contener los datos inyectados. Para mejorar la posibilidad de éxito, se suelen asignar datos con la forma de la figura 4.

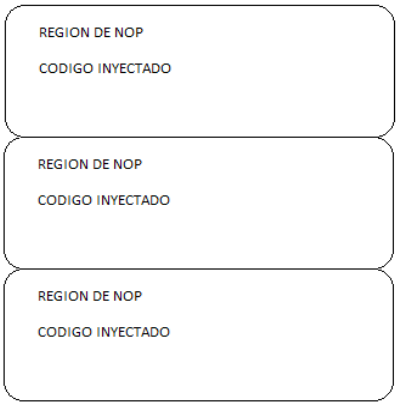

Figura 4. Esparcimiento del montículo; se asignan bloques con espacios de no-operación y código inyectado

Fuente: propia.

4. TRABAJOS FUTUROS

Los adelantos de los mecanismos de protección han demostrado ser importantes elementos para mitigar el riesgo. Existen diferentes formas de implementar estas protecciones. La forma en que se implementan los mecanismos de defensa puede disminuir la efectividad de estas tecnologías.

Existen alternativas para determinar la existencia de los mecanismos de protección presentados en este documento. De manera aislada se han realizado mediciones para evaluar la efectividad de algunas de estas medidas de seguridad. Sin embargo, no existe un método que permita evaluar o comparar diferentes implementaciones en las medidas de seguridad de manera objetiva.

Los adelantes en los mecanismos de defensas de los sistemas operativos de propósito general han creado la pregunta de si estas medidas son aplicables en el contexto de sistemas embebidos y si en dichas arquitecturas se cuenta con la seguridad adecuada para el contexto que representan.

Las medidas de protección afectan principalmente a las aplicaciones de usuario y esto ha generado que muchos atacantes comiencen a evaluar los fallos de seguridad del núcleo del sistema operativo. Se deberían incluir mecanismos de defensa adecuados para disminuir el riesgo que experimentan los programas en modo supervisor, ya que el interés por atacar los componentes del sistema operativo cada vez es mayor.

5. CONCLUSIONES

Los fallos por corrupción de memoria, al igual que otro tipo de debilidades, son causados por errores humanos que pueden existir en cualquier etapa del proceso de desarrollo de los sistemas computacionales. Es importante concientizar a los ingenieros de los requerimientos no funcionales y las buenas prácticas de programación que ayudan a disminuir el riesgo de experimentar vulnerabilidades en los sistemas. La primera línea de defensa en las aplicaciones es un buen proceso de desarrollo. El análisis de requerimientos de seguridad, las buenas prácticas de diseño e implementación y un adecuado control de calidad con un conjunto de pruebas adecuado, es fundamental para reducir el riesgo de experimentar este tipo de fallos.

Aunque esta línea de protección debería ser suficiente para detener los ataques, se prefieren los enfoques de protección en diferentes capas. Para ayudar a mitigar los riesgos, los sistemas operativos han incluido diferentes mecanismos de protección. Dos de los mecanismos más representativos son: hacer aleatorio el espacio virtual de direcciones y los privilegios de ejecución en las regiones de memoria. Sin embargo, estos mecanismos no evitan que la vulnerabilidad exista; por el contrario, hacen que el proceso de un atacante para aprovechar una vulnerabilidad sea más complicado. Es importante mencionar que estos mecanismos de protección no son perfectos y que los atacantes han desarrollado técnicas para evadir su funcionamiento. Esto no quiere decir que las medidas de protección no sean efectivas o que sean obsoletas; simplemente, es una manifestación de la teoría del riesgo residual.

REFERENCIAS

[1] H. Meer, "Memory Corruption Attacks The (Almost) Complete History," presentado en Black Hat USA, Las Vegas, 2010.

[2] E. Levy. «Smashing the stack for fun and profit, PhrackInc,» [En línea], acceso octubre 2010; Disponible: http://www.phrack.com/issues.html?issue=49&id=14, 1996.

[3] T. Twillman. "Exploitforproftpd 1.2.0pre6," [En línea], acceso octubre 2010; Disponible: http://seclists.org/bugtraq/1999/Sep/328, 1999.

[4] M. Kaempf. "VudoMallocTricks,PhrackInc," [En línea], acceso octubre 2010; Disponible: http://www.phrack.org/issues.html?issue=57&id=8, 2001.

[5] L. Le, "Payload Already Inside: Data Reuse ForRop Exploits," presentado en Black Hat USA, Las Vegas, 2010.

[6] J. Pincus, y B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *IEEE Security & Privacy,* vol. 2, no. 4, pp. 20-27, 2004.

[7] A. Fog. "Calling conventions for different C++ compilers and operating systems, Copenhagen University College of Engineering," [En línea], acceso septiembre 2010; Disponible: http://www.agner.org/optimize/calling_conventions.pdf, 2010.

[8] R. Gera. "Advances in format string exploitation, PhrackInc," [En línea], acceso septiembre 2010; Disponible: http://www.phrack.org/issues.html?issue=57&id=9#article, 2002.

[9] D. Lea. "The design of the basis of the glibc allocator," [En línea], acceso septiembre 2010; Disponible: http://gee.cs.oswego.edu/dl/html/malloc.html, 1996.

[10] P. Phantasmagoria. "The Malloc Maleficarum. Glibc Malloc Exploitation Techniques, Security Focus," [En línea], acceso septiembre 2010; Disponible: http://www.securityfocus.com/archive/1/413007/30/0/threaded, 2005.

[11] J. Afek, y A. Sharabani, "Dangling Pointers Smashing the pointer for fun and profit," presentado en Black Hat USA, Las Vegas, 2007.

[12] M. Conover. "Double Free Vulnerabilities, Symantec," [En línea], acceso octubre 2010; Disponible: http://www.symantec.com/connect/blogs/double-free-vulnerabilities-part-1, 2007.

[13] H. Burch. "Vulnerability Note VU#946969, US-CERT," [En línea], acceso septiembre 2010; Disponible: http://www.kb.cert.org/vuls/id/946969, 2005.

[14] Shahin. "Java CMM readMabCurveData stack overflow, abysssec," [En línea], acceso agosto 2010; Disponible: http://www.exploit-db.com/exploits/15056/, 2010.

[15] S. M. Pike *et al.*, "Checkmate: Cornering C++ Dynamic Memory Errors With Checked Pointers," presentado en 31st SIGCSE Technical Symposium on Computer Science Education: ACM Press, 2000.

[16] M. Sutton *et al.*, *Fuzzing: Brute Force Vulnerability Discovery*, Indiana: Addison-Wesley Professional, 2007, p.

[17] Nergal. "The advanced return-into-lib(c) exploits, PhrackInc," [En línea], acceso septiembre 2010; Disponible: http://www.phrack.org/issues.html?issue=58&id=4, 2001.

[18] E. E. Buchanan *et al.*, "Return-oriented Programming: Exploitation without Code Injection," presentado en Black Hat USA, Las Vegas, 2008.

[19] T. Müller, "ASLR Smack & Laugh Reference," presentado en Seminar on Advanced Exploitation Techniques, Germany, 2008.

[20] O. Whitehouse. "An Analysis of Address Space Layout Randomization on Windows Vista, SYMANTEC," [En línea], acceso octubre 2010; Disponible: http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf, 2007. w