Fuzzing para pruebas de seguridad en software

A. Alejandro Hernández Hernández
nitrousenador@gmail.com

Muchas de las compañías saben que producir y entregar parches para corregir errores en sus productos es más caro que agregar más calidad a la etapa de desarrollo desde el principio, y aún así muchos (sino es que la mayoría) de los productos son lanzados con fallos de seguridad causando así un gran número de intrusiones informáticas. Uno de los métodos usados para encontrar fallos de seguridad es el fuzzing, el cual es un proceso semi-automático que combina técnicas de pruebas como funcional, de caja negra, exploratoria, seguridad y resistencia para descubrir fallos de seguridad. El fuzzing es usado por compañías de software y proyectos de código abierto para mejorar la calidad de su software; por analistas de vulnerabilidades para descubrir y reportar nuevos fallos; por hackers para encontrar y secretamente explotar software. Sin embargo, cuando se comparan las técnicas tradicionales de prubas de software, aún auditorias de código fuente e ingeniería inversa, el fuzzing se encuentra más efectivo y eficiente en costos. En esta revisión también se presentan resultados obtenidos de estudios anteriores a software de diferentes sistemas operativos.

Palabras clave: pruebas de software, pruebas de seguridad, pruebas de resistencia, pruebas difusas, pruebas de software, ingeniería de software.

Las pruebas de software son una parte esencial en el proceso de desarrollo de software, ya que en esta parte es donde se detectan fallos antes de que el software sea lanzado al mercado. Existen diferentes metodologías para probar software, entre las que destacan las pruebas de caja blanca y caja negra. La primera de ellas se caracteriza por el hecho de que se tiene conocimiento del comportamiento del software en si, ya sea a través de código fuente o información extra. En la segunda (caja negra) no se tiene ningún conocimiento de la estructura interna de un programa, simplemente se conocen sus entradas y sus salidas.

En general, la etapa de pruebas de software puede ser definida como el proceso usado para ayudar a identificar fallas, nivel de robustez, seguridad y calidad en el software. En términos muy generales, la etapa de pruebas nos permite conocer la brecha entre la calidad del software deseado con la calidad actual.

Muchas de las compañías saben que producir y entregar parches para corregir errores en sus productos es más caro que agregar más calidad a la etapa de desarrollo desde el principio, y aún así muchos (sino es que la mayoría) de los productos son lanzados con fallos de seguridad.

La mayoría de intrusiones en sistemas informáticos son el resultado de vulnerabilidades en las aplicaciones. La detección e identificación de dichos fallos es un proceso interesante no solo para expertos en seguridad y administradores de sistemas, sino que también para intrusos que intentan penetrar sistemas informáticos. Una vez que los fallos son detectados, los *exploits* para dichas vulnerabilidades pueden ser creados y los intrusos pueden penetrar un alto número de sistemas en Internet, lo cual es una amenaza potencial para todos los usuarios de sistemas de información (Juranic, 2006).

En las siguientes tablas se muestran las últimas estadísticas de incidentes de seguridad reportados en México (2005) según el Equipo de Respuesta a Incidentes de Seguridad en Cómputo de la UNAM (UNAM-CERT):

Año	**2005**
Incidentes	2635

Tabla 1: Total de incidentes en 2005.

Sistema operativo	**Porcentaje**
Windows XP Pro	46.73
Windows XP Home	18.47
Windows 2000 Pro	13.04
Windows ME	5.43
Windows 98	5.43
Windows 2000 Server	3.26
Cobalto	4.21
Windows NT	2.17
Linux kernel 2.4.x	2.17

Tabla 2: Estadísticas de incidentes por Sistema Operativo.

La mayoría de estos incidentes han sido por la explotación de software vulnerable, los cuales podrían ser reducidos si se aplicaran diferentes pruebas de seguridad, entre ellas fuzzing.

Existen varios métodos que son usados para encontrar fallos de seguridad, los siguientes son algunos de ellos:

- Análisis de código fuente
- Análisis estático del archivo binario
- Análisis dinámico del archivo binario (en tiempo de ejecución)
- Fuzzing
- Métodos híbridos(varias combinaciones de los métodos mencionados arriba)

Aunque todos los métodos mencionados arriba pueden detectar fallos de seguridad, algunos de ellos son mejores que otros. En los últimos años, se ha dado especial atención a la técnica llamada fuzzing (Juranic, 2006).

En el artículo de 2006, DeMott define fuzzing como la técnica de prueba de software utilizada para encontrar fallos en el software, la mayoría de ellos relacionados con seguridad. El fuzzing es un proceso semi-automático (requiere interacción humana) que combina técnicas de pruebas como funcional, de caja negra, exploratoria, seguridad y resistencia. El fuzzing complementa las pruebas de software tradicionales (manualmente) para descubrir combinaciones no probadas de código y datos, combinando el poder de datos aleatorios, conocimiento de protocolos o estructuras de datos y heurística de ataque.
El fuzzing es usado por compañías de software y proyectos de código abierto, por ejemplo Microsoft y Linux, para mejorar la calidad de su software; por analistas de vulnerabilidades para descubrir y reportar nuevos fallos; por hackers para encontrar y secretamente explotar software. Sin embargo, cuando se comparan las técnicas tradicionales de prubas de software, aún auditorias de código fuente e ingeniería inversa, el fuzzing se encuentra más efectivo y eficiente en costos (DeMott, 2006).

Existe un gran número de entes que pueden ser fuzzeados, entre los más comunes se encuentran los siguientes:

- Formatos de Archivo
- Entradas de datos estándar
- Protocolos de Red (Capa de Aplicación)
- Pilas de Protocolos
- Argumentos
- Variables de Entorno

Aunque Burns (2006) nos muestra también muchos elementos fuzzeables en mecanismos internos del sistema operativo Windows tales como, eventos, archivos, entradas de registro, tuberías con nombre, semáforos, secciones compartidas, temporizadores, entre otros. Otro componente de software que comúnmente se fuzzea en estos días son los controles ActiveX, donde muchos de los cuales son completamente inseguros (Wysopal, Nelson, Zovi y Dustin, 2007). Actualmente existen muchas formas de prueba más, tales como la aplicación de fuzzing en las convenciones de llamadas del lenguaje C que nos muestra Lindig (2005). Es posible encontrar en las referencias otros casos de estudio como: obteniendo señal fatal en gcc [6], múltiples fallos en Adobe Acrobat Reader [7] y desbordamiento de buffer en WarFTPD [8].

Estudios anteriores ([4], [12] y [13]) demostraron que gran cantidad de software externo y a la vez parte de sistemas operativos ampliamente usados como Microsoft Windows, MacOS y basados en UNIX, tuvieron fallas luego de haber aplicado pruebas por medio de fuzzing. Esta es una serie de estudios iniciada en 1990, en donde se estudiaron varios programas de línea de comandos de UNIX. Luego, la tercer publicación fué en 2000, donde se aplicaban pruebas pero en sistemas operativos basados en Windows NT. Finalmente, la cuarta publicación se orientó a aplicaciones en el sistema operativo MacOS.
La examinación de resultados demuestran muchos errores hechos por los programadores. Muchos de estos errores se relacionan con áreas bien conocidas por programadores experimentados (Miller, Fredriksen y So, 1990).

El fuzzing también es conocido como pruebas aleatorias (Random testing en inglés), las cuales son el proceso de seleccionar datos a probar de forma aleatoria de acuerdo a la distribución de probabilidad uniforme y enviar estos datos al dominio de entradas del programa a probar. Uniforme significa que cada punto del dominio tiene la misma probabilidad de ser seleccionado (Gotlieb y Petit, 2006). Un generador de datos para pruebas aleatorias debe ser uniforme cuando cada punto del dominio de entrada tiene la misma probabilidad de ser escogido. Sin embargo, es bien conocido que esa uniformidad solo puede ser aproximada en máquinas determinísticas (como las computadoras usuales) y solo pseudo generadores de números aleatorios pueden ser empleados, aunque muchas veces los pseudo generadores de números aleatorios hacen uso de métodos congruenciales lineares con lo que se pierde una estricta uniformidad.

Por consecuencia, a lo largo de los años las técnicas de fuzzing han ido evolucionando. Se han propuesto diferentes métodos, así como también comparaciones entre los mismos. En el estudio de Hildebrandt y Zeller (2000) plantean que se debe de hacer un uso limitado de casos de prueba, es decir, seleccionar solo los casos que detecten fallos efectivamente [6]. Dentro de sus estudio, también plantean que, dada una serie de casos de prueba, el programa falla, pero ¿que caso de prueba y que parte de ella es la responsable de ese fallo en particular?. Por consiguiente, diseñaron un algoritmo que generaliza y simplifica el caso de prueba mínimo para producir el fallo.

Estudios previos [2] demuestran que los métodos adaptativos son más efectivos en la detección de fallos que las pruebas aleatorias normales. Ya que los métodos adaptativos son tan simples como los métodos normales, y aún así preservan cierto grado de aleatoriedad, se puede decir que los métodos adaptativos podrían ser un reemplazo efectivo los métodos normales de pruebas aleatorias (Chen y Kuo, 2006).

Existen también diferentes métodos adaptativos, de los cuales, según en un estudio empírico [4], los autores confirman que los métodos “basado en distancia” y “prueba aleatoria restringida” son definitivamente los mejores en su rama. Estos métodos deberían ser usados en casos donde una sola ejecución del sistema sometido a pruebas tome mucho tiempo (Forrester y Miller, 2000).

BUSQUEDA DE LA LITERATURA

La primer búsqueda hecha fué en la Biblioteca Digital de la ACM con las siguientes palabras clave: “+fuzz +testing”, lo cual nos arrojó 199,787 fichas pero no toda eran referidas a pruebas de software. El mismo número de resultados nos arrojan búsquedas con diferentes criterios, por ejemplo, “+fuzzing +"random testing"” y “+fuzz +testing software”. Entonces, procedimos a hacer una búsqueda avanzada con el texto: “+fuzz +testing” solamente en el resumen obteniendo así solo 241 resultados. Analizando los resultados y aplicando los criterios de inclusión y exclusión descritos mas adelante, finalmente contamos con 10 publicaciones de la ACM.

Como segunda fuente, se acudió a la Biblioteca Digital de la IEEE buscando con las palabras clave: “random testing” obteniendo 100 fichas, aunque muy pocas relacionadas con Ingeniería de Software. Luego de analizar algunos de los resultados, solamente se pudieron revisar pequeños resúmenes de cada investigación, ya que es necesario ser miembro de dicha biblioteca.

Finalmente, se decidió buscar en fuentes externas tales como organizaciones y empresas que hacen investigación en dicha rama. Luego de dicha búsqueda, se obtuvo un total de 5 publicaciones con las siguientes fuentes:

- AppliedSec
- Netric security research group
- iSEC Partners
- Ilfigo Security
- The Art of Software Security Testing (Libro)

Y es por ello que también tomamos en cuenta dichas publicaciones debido al alto contenido informativo de las mismas.

Para obtener los resultados deseados, se aplicaron los siguientes criterios de inclusión y exclusión para obtener los artículos para ser analizados. Cada artículo debe cumplir alguno de los siguientes criterios:

- El término *fuzzing* debe estar completamente enfocado al Software.
- El término *fuzzing* debe estar enfocado a la etapa de pruebas en la Ingeniería de Software.
- Estar relacionado con cualquier tipo de *fuzzing* como protocolos, datos, entradas, archivos, sistemas de archivo, argumentos y variables de entorno.

Actualmente, la mayoría de publicaciones con alto nivel de calidad las podemos encontrar en Bibliotecas digitales de asociaciones profesionales tales como las de la ACM y la IEEE por mencionar algunas. Es por ello, que como base de conocimiento fundamental en Ciencias de la Computación utilizamos dichas fuentes de información, ya que los artículos allí publicados fueron previamente revisados por expertos dependiendo del tema de la publicación.

Así pues, utilizamos como primer recurso la Biblioteca digital de la ACM para buscar publicaciones y artículos relacionados con nuestro tema de interés. Nuestra segunda fuente de información fueron diversos centros de investigación, universidades, empresas, investigadores independientes, etc. Que tienen alto grado de conocimiento en dicha rama de la ciencia, y por supuesto, su material expuesto contiene un alto grado de confiabilidad y aportación a la materia.

Finalmente, contamos con diez publicaciones de la Biblioteca Digital de la ACM y otras cinco de fuentes externas que no publicaron su trabajo en la ACM, que hacen un total de 15 publicaciones para ser analizadas.

RESULTADOS

La siguiente información es el resultado de diferentes investigaciones en la materia, las cuales muestran el impacto que diferentes variables pueden ocasionar en distintas áreas de tecnologías de información. Las siguientes tablas muestran algunos de los resultados obtenidos en los tres estudios citados anteriormente ([4], [12] y [13]).

*●=utility crashed, ○=utility hung, *=crashed on SunOS 3.2 but not on SunOS 4.0, ⊕ = crashed only on SunOS 4.0, not 3.2. —=utility unavailable on that system. !=utility caused the operating system to crash.*

Utility	VAX (v)	Sun (s)	HP (h)	I386 (x)	AIX 1.1 (a)	Sequent (d)
adb	● ○	●	●	○	—	—
as	●			●	●	●
awk						
bc				● ○		
bib			—	—	—	—
calendar				—		
cat						
cb	●		●	●	○	●
cc						
/lib/ccom				—	—	●
checkeq				—		
checknr				—	—	
col	● ○	●	●	● ○	●	●
colort				—	—	
colrm				—	—	
comm						
compress					—	
/lib/cpp						
csh	● ○	○	○	—	○	○
dbx		*	—	—		
dc				○		
deqn		●	—	—	—	—
deroff	●	●	●		●	●
diction	●	—	●		—	●
diff						
ditroff	● ○	●	—	—	—	
dtbl			—	—	—	—
emacs	●	●	○	—	—	
eqn		○	●	●		
expand					—	
f77	●		—	—	—	—
fmt						
fold					—	
ftp	●	●	●	—	●	●
graph					—	
grep						
grn			—	—	—	—
head					—	
ideal			—	—	—	—
indent	● ○	● ○	●	—	—	●
join		⊕				
latex			—	—	—	—
lex	●	●	●	●	●	●
lint						
lisp		—		—	—	—
look	●	○	●	●	—	●

Tabla 3 (parte 1): Lista de utilidades probadas y los sistemas sobre los cuales fueron probadas.

*●=utility crashed, ○=utility hung, *=crashed on SunOS 3.2 but not on SunOS 4.0, ⊕ = crashed only on SunOS 4.0, not 3.2. —=utility unavailable on that system. !=utility caused the operating system to crash.*

Utility	VAX (v)	Sun (s)	HP (h)	I386 (x)	AIX 1.1 (a)	Sequent (d)
m4				●		
mail						
make			●			
more					—	
nm						
nroff				●		
pc				—	—	—
pic			—	—	—	—
plot	—	○	●	—	—	
pr						—
prolog	● ○	● ○	● ○	—	—	—
psdit				—	—	
ptx	—	●	●	○		○
refer	●	*	●	—	—	! ●
rev				—	—	
sed						
sh				—		
soelim					—	
sort						
spell	● ○	●	●	○	●	●
spline					—	
split						
sql		—			—	—
strings					—	
strip						
style	●	—	●		—	●
sum						
tail						
tbl						
tee						
telnet	●	●	●	—	●	○
tex			—	—	—	—
tr						
troff	—	—	—			
tsort	●	*	●	●	●	●
ul	●	●	●	—	—	●
uniq	●	●	●	●	●	●
units	● ○	●	●	●	●	●
vgrind	●		—	—	—	
vi	●		●	—		
wc						
yacc						
# tested	85	83	75	55	49	73
# crashed/hung	25	21	25	16	12	19
%	29.4%	25.3%	33.3%	29.1%	24.5%	26.0%

Tabla 3 (parte 2): Lista de utilidades probadas y los sistemas sobre los cuales fueron probadas.

Como se puede apreciar en la tabla tres, se probaron 85 aplicaciones en plataforma VAX, de las cuales 25 fallaron (29.4 %); 83 aplicaciones en Sun, de las cuales 21 fallaron (25.3 %); 75 aplicaciones en HP, de las cuales 25 fallaron (33.3 %); 55 aplicaciones en i386, de las cuales 16 fallaron (29.1 %) y finalmente 49 aplicaciones en AIX 1.1, de las cuales 12 aplicaciones fallaron (24.5 %).

En la siguiente publicación, de la misma serie de estudios, se muestran los resultados obtenidos luego de una etapa de pruebas aleatorias al sistema operativo Windows NT 4.0. Estos resultados se muestran en la tabla cuatro, los cuales nos dicen que cuando se probaron las aplicaciones con la API *SendMessage*, el 72.7 % de los programas murieron y el 9 % quedaron colgados, dando un total de 81.7 % de programas fallados. Con la API *PostMessage*, el 90.9 % de los programas murieron y el 6 % quedaron colgados, dando un total de 96.9 % de programas fallados. Finalmente, probando con eventos del sistema aleatorios, el 21.2 % de los programas murieron y el 24.2 % quedaron colgados, dando un total de 45.4 % de programas fallados.

Application	Vendor	SendMessage	PostMessage	Random Valid Events
Access 97	Microsoft	●	●	○
Access 2000	Microsoft	●	●	○
Acrobat Reader 4.0	Adobe Systems	●	●	
Calculator 4.0	Microsoft		●	
CD-Player 4.0	Microsoft	●	●	
Codewarrior Pro 3.3	Metrowerks	●	●	●
Command AntiVirus 4.54	Command Software Systems	●	●	
Eudora Pro 3.0.5	Qualcomm	●	●	○
Excel 97	Microsoft	●	●	
Excel 2000	Microsoft	●	●	
FrameMaker 5.5	Adobe Systems		●	
FreeCell 4.0	Microsoft	●	●	
Ghostscript 5.50	Aladdin Enterprises	●	●	
Ghostview 2.7	Ghostgum Software Pty	●	●	
GNU Emacs 20.3.1	Free Software Foundation	●	●	
Internet Explorer 4.0	Microsoft	●	●	●
Internet Explorer 5.0	Microsoft	●	●	
Java Workshop 2.0a	Sun Microsystems		●	○
Netscape Communicator 4.7	Netscape Communications	●	●	●
NotePad 4.0	Microsoft	●	●	
Paint 4.0	Microsoft	●	●	
Paint Shop Pro 5.03	Jasc Software		○	
PowerPoint 97	Microsoft	○	○	○
PowerPoint 2000	Microsoft	○		○
Secure CRT 2.4	Van Dyke Technologies	●	●	○
Solitaire 4.0	Microsoft		●	
Telnet 5 for Windows	MIT Kerberos Group		●	
Visual C++ 6.0	Microsoft	●	●	●
Winamp 2.5c	Nullsoft	○	●	
Word 97	Microsoft	●	●	●
Word 2000	Microsoft	●	●	●
WordPad 4.0	Microsoft	●	●	●
WS_FTP LE 4.50	Ipswitch	●	●	○
Percent Crashed		72.7%	90.9%	21.2%
Percent Hung		9.0%	6.0%	24.2%
Total Percent Failed		81.7%	96.9%	45.4%

Tabla 4: Resumen de los resultados de pruebas en Windows NT 4.0.
● = muerte, ○ = cuelgue

Finalmente, en la tablas cinco y seis se muestran los resultados aplicados a las utilidades de línea de comandos y los resultados aplicados a las utilidades gráficas respectivamente. Estas pruebas fueron hechas bajo el sistema operativo Mac OS X 10.4.3. En la tabla cinco se hizo un análisis comparativo de fallos en programas de línea de comandos entre Linux y Mac OS, resultando este último con menos fallos que el primero ya que de 54 aplicaciones probadas, 5 murieron o se colgaron (9 %), mientras que en Mac OS se probaron 135 aplicaciones y solo 10 murieron o se colgaron (7 %).

Utility	Linux - 1995	Mac OS- 2006
Only Mac OS:		
expr		●
groff		●
zic		●
zsh		●
Both Mac OS and Linux:		
as		●
ctags	○	
flex	●	
gdb	●	
indent	●	●
nroff		●
ditroff/troff		●
ul	●	●
vi(ex)/vim		●
Number crash/hang:	5	10
Number tested:	54	135
Percentage:	9%	7%

Tabla 5: Resultados de pruebas de utilidades en línea de comandos.
● = muerte, ○ = cuelgue

En la tabla seis, se muestran los resultados obtenidos de las pruebas en el mismo sistema operativo (Mac OS X 10.4.3) pero solo a utilidades con interfaz gráfica. En esta prueba se utilizaron una gran serie de programas de Apple, así como también de otros proveedores muy conocidos como Microsoft, Adobe, Mozilla, entre otros. En total se probaron 30 aplicaciones, de las cuales casi la mayoría tuvo fallos (22 aplicaciones), correspondiendo así al 73 % de aplicaciones con fallos.

Vendor	Application	Result
Adobe	Acrobat Reader 7.0.5	●
adium.com	Adium X 0.87	●
Apple Computer	Calculator 10.4.3	
	Dictionary 10.4.3	○
	Finder 10.4.3	●
	GarageBand 2.0.2	
	iCal 10.4.3	○
	iChat 10.4.3	●
	iDVD 5.0.1	
	iMovie 5.0.2	
	iPhoto 5.0.4	
	iTunes 6.0.1	●
	Keynote 2.0.2	●
	Mail 10.4.3	●○
	Pages 1.0.2	●
	Preview 10.4.3	
	Safari 10.4.3	
	Sherlock 10.4.3	●
	TextEdit 10.4.3	
	Xcode 2.2	●
aquamacs.org	Aquamacs Emacs 0.9.7	●
Microsoft Corporation	Excel 11.2.0	●
	Internet Explorer 5.2.3	●
	PowerPoint 11.2.0	●
	Word 11.2.0	●
Mozilla Foundation	Camino 0.8.4	●○
	Firefox 1.5	●
	Thunderbird 1.5	●
Omni Group	OmniWeb	●
Opera Software	Opera 8.51.2182	●
	# tested:	30
	# crash/hang:	22
	%:	73%

Tabla 6: Resultados de pruebas de utilidades con interfaz gráfica.
● = muerte, ○ = cuelgue

Estos resultados muestran claramente que existe un alto porcentaje de fallos en muchas de las aplicaciones probadas, sin embargo, día con día las vulnerabilidades se hacen más complejas y difíciles de explotar, y es ahí en donde el fuzzing entra en juego.

DISCUSION

CONCLUSION

Como se puede ver, el impacto que causan los errores de seguridad en el software es muy grande. Es por ello que se han creado nuevos métodos cada vez más complejos para detección y corrección de errores, entre ellos el fuzzing. Como se puede analizar en los resultados de estudios previos, el número de errores encontrados por medio de fuzzing son cada vez mayores, y mientras siga existiendo la falta de pruebas exhaustivas antes de lanzar un software al mercado, siempre existirá la posibilidad de que dicho producto tenga errores de seguridad que puedan ser explotados por muchos del gran número de amenazas que alberga el Internet.

Las herramientas de detección de errores conocidas como fuzzers, son de gran ayuda en la etapa de pruebas del software y en el análisis de vulnerabilidades. Los mejores fuzzers de hoy en día son desarrollados por probadores de software y analistas de vulnerabilidades con experiencia. Estos incluyen descripciones automáticas de protocolos, aleatoriedad, heurística, seguimiento/depuramiento, logueo, y más.

Es por ello que es recomendable aplicar buenas prácticas de programación, para así aumentar la calidad del software, por consecuente, seguridad.

INVESTIGACION FUTURA

Este tema cada vez más está tomando suma importancia, logrando así que centros de investigación e investigadores independientes se enfoquen en nuevos métodos e implementaciones de fuzzing, llamados fuzzers.

Entre dichos métodos destaca el de la simplificación de un dominio específico y métodos adaptativos, es por ello que se debe poner mayor importancia a estos tópicos. Sabemos que una herramienta para la implementación de los fuzzers es la generación de números aleatorios, y es por ello que la aleatoriedad inteligente es otro tema de estudio importante.

Actualmente se encuentran decenas de fallos diariamente en listas de correo sobre seguridad, y cada vez es mayor el número de errores de desbordamiento de enteros (Integer overflows). Uno de los vectores de explotación de dichos fallos es a través de formatos de archivo. El autor actualmente está desarrollando un fuzzer para formatos de archivo ELF, ya que es un tópico no abordado en el tema de fuzzing y por consiguiente es un interesante punto para análisis de vulnerabilidades.

REFERENCIAS

[1] Burns, J. (2006). Fuzzing win32 interprocess communication mechanisms. Presentado en Black Hat 2006 Conference, Las Vegas, NV, USA.

[2] Chen, T. Y. y Kuo, F. C. (2006). *Is adaptive random testing really better than random testing*. En proceedings of the First International Workshop on Random Testing (RT '06), Julio 20, 2006, Portland, ME, USA.

[3] DeMott, J. (2006). *The evolving art of fuzzing*. Recuperado el 23 de marzo de 2007, de http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.pdf

[4] Forrester, J. y Miller, B. (2000). *An empirical study of the robustness of Windows NT applications using random testing.* En proceedings of the 4th USENIX Windows System Symposium, August 2000, Seattle, USA.

[5] Gotlieb, A. y Petit, M. (2006). *Path-oriented random testing.* En proceedings of the First International Workshop on Random Testing (RT '06), Julio 20, 2006, Portland, ME, USA.

[6] Hildebrandt, R. y Zeller, A. (2000). *Simplifying failure-inducing input*. En Proceedings of ISSTA '00, páginas 135-145, Portland, Oregon.

[7] Jorgensen, A. A. (2003). Testing with hostile data streams. *ACM SIGSOFT, Software Engineering Notes, 28*(2).

[8] Juranic, L. (2006). *Using fuzzing to detect security vulnerabilities*. Recuperado el 23 de marzo de 2007, de http://www.infigo.hr/files/INFIGO-TD-2006-04-01-Fuzzing-eng.pdf

[9] Lindig, C. (2005). *Random testing of C calling conventions*. En proceedings of AADEBUG '05, September 19–21, 2005, Monterey, California, USA.

[10] Marquis, S., Dean, T. R. y Knight, S. (2005). *SCL: A language for security testing of network applications*. Kingston, Canada.

[11] Mayer, J. y Schneckenburger, C. (2006). *An empirical analysis and comparison of random testing techniques*. Ulm, Alemania.

[12] Miller, B., Cooksey, G. y Moore, F. (2006). *An empirical study of the robustness of MacOS applications using random testing*. Madison, USA.

[13] Miller, B., Fredriksen, L. y So, B. (1990). Study of the reliability of UNIX utilities. *ACM Communications, 33* (12), 32-44.

[14] Sprundel, I. V. (2005). *Fuzzing: Breaking software in an automated fashion.* Recuperado el 23 de marzo de 2007, de http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf

[15] Wysopal, C., Nelson, L., Zovi, D. D. y Dustin, E. (2007). Local fault injection (cap. 11) en *The art of software security testing,* páginas 201-229. Addison Wesley Professional. Recuperado el 23 de marzo de 2007, de http://www.securityfocus.com/images/infocus/Wysopal_CH11.pdf