

Bruce Dang, Alexandre Gazet, and Elias Bachaalany

with contributions from Sébastien Josse

PRACTICAL REVERSE ENGINEERING

X86, X64, ARM, WINDOWS® KERNEL,
REVERSING TOOLS, AND OBFUSCATION

WILEY

Ingeniería inversa práctica



Práctico reverso Ingeniería

x86, x64, ARM, núcleo de Windows®,
Herramientas de reversión y ofuscación

Bruce Dang
Alexandre Gazer
Elías Bachaalany

con contribuciones de Sébastien Josse

WILEY

Ingeniería inversa práctica: x86, x64, ARM, kernel de Windows®, herramientas de reversión y ofuscación

Publicado por
John Wiley & Sons, Inc.
10475 Bulevar Crosspoint
Indianápolis, IN 46256
www.wiley.com

Derechos de autor © 2014 por Bruce Dang
Publicado por John Wiley & Sons, Inc., Indianápolis, Indiana
Publicado simultáneamente en Canadá

Libro de bolsillo: 978-1-118-78731-1
ISBN: 978-1-118-78725-0 (versión electrónica)
ISBN: 978-1-118-78739-7 (versión electrónica)

Fabricado en los Estados Unidos de América.

10 9 8 7 6 5 4 3 2 1

Ninguna parte de esta publicación puede ser reproducida, almacenada en un sistema de recuperación o transmitida en ninguna forma o por ningún medio, electrónico, mecánico, fotocopia, grabación, escaneo o de otro modo, excepto según lo permitido por las Secciones 107 o 108 de la Ley de Derechos de Autor de los Estados Unidos de 1976, sin el permiso previo por escrito del Editor o la autorización mediante el pago de la tarifa correspondiente por copia al Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Las solicitudes de permiso al editor deben dirigirse al Departamento de Permisos, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, o en línea en <http://www.wiley.com/go/permissions>.

Límite de responsabilidad/Exención de garantía: El editor y el autor no realizan declaraciones ni otorgan garantías con respecto a la exactitud o integridad del contenido de esta obra y renuncian específicamente a todas las garantías, incluidas, entre otras, las garantías de idoneidad para un propósito particular. No se puede crear ni extender ninguna garantía mediante materiales de venta o promoción. Los consejos y estrategias aquí contenidos pueden no ser adecuados para todas las situaciones. Esta obra se vende con el entendimiento de que el editor no se dedica a brindar servicios legales, contables u otros servicios profesionales. Si se requiere asistencia profesional, se deben buscar los servicios de un profesional competente.

Ni el editor ni el autor serán responsables de los daños que se deriven de este documento. El hecho de que en este trabajo se haga referencia a una organización o sitio web como cita y/o fuente potencial de información adicional no significa que el autor o el editor respalden la información que la organización o el sitio web puedan proporcionar o las recomendaciones que puedan hacer. Además, los lectores deben tener en cuenta que los sitios web de Internet que se mencionan en este trabajo pueden haber cambiado o desaparecido entre el momento en que se escribió este trabajo y el momento en que se lee.

Para obtener información general sobre nuestros otros productos y servicios, comuníquese con nuestro Departamento de Atención al Cliente dentro de los Estados Unidos al (877) 762-2974, fuera de los Estados Unidos al (317) 572-3993 o por fax (317) 572-4002.

Wiley publica en una variedad de formatos impresos y electrónicos y mediante impresión a pedido. Es posible que algunos materiales incluidos en las versiones impresas estándar de este libro no estén incluidos en los libros electrónicos o en la impresión a pedido. Si este libro hace referencia a medios como un CD o DVD que no están incluidos en la versión que compró, puede descargar este material en <http://booksupport.wiley.com>. Para obtener más información sobre los productos Wiley, visite www.wiley.com.

Número de control de la Biblioteca del Congreso: 2013954099

Marcas comerciales: Wiley y el logotipo de Wiley son marcas comerciales o marcas comerciales registradas de John Wiley & Sons, Inc. y/o sus filiales en los Estados Unidos y otros países, y no pueden usarse sin permiso por escrito. Todas las demás marcas comerciales son propiedad de sus respectivos dueños. John Wiley & Sons, Inc. no está asociada con ningún producto o proveedor mencionado en este libro.

Este libro está dedicado a aquellos que buscan incansablemente el conocimiento y lo comparten desinteresadamente con los demás.

Acerca de los autores

Bruce Dang es un ingeniero de desarrollo de seguridad de alto nivel en Microsoft que trabaja en tecnologías de seguridad para productos de Microsoft que aún no se han lanzado al mercado. Anteriormente, trabajó en vulnerabilidades de seguridad que se informaron a Microsoft y fue el primero en compartir públicamente técnicas analíticas para ataques dirigidos con documentos de Office . Él y su equipo analizaron el famoso malware Stuxnet, que supuestamente atacó el proceso de enriquecimiento de uranio iraní. Ha hablado en RSA, BlackHat Vegas, BlackHat Tokyo, Chaos Computer Club, REcon y muchas otras conferencias de la industria.

Alexandre Gazet es investigador de seguridad en Quarkslab. Sus intereses se centran en la ingeniería inversa, las protecciones de software y la investigación de vulnerabilidades. Alexandre ha realizado presentaciones en varias conferencias, incluidas HITB Kuala Lumpur (2009) y REcon Montreal (2010 y 2011).

Elias Bachaalany ha sido programador informático, ingeniero inverso, redactor técnico independiente y, ocasionalmente, formador en ingeniería inversa durante los últimos 14 años. A lo largo de su dilatada carrera, Elias ha trabajado con diversas tecnologías, como la redacción de scripts, el desarrollo web exhaustivo, el diseño y la programación de bases de datos, la redacción de controladores de dispositivos y códigos de bajo nivel, como cargadores de arranque o sistemas operativos mínimos, la redacción de códigos administrados, la evaluación de protecciones de software y la redacción de herramientas de ingeniería inversa y seguridad de escritorio . Elias también ha realizado dos presentaciones en REcon Montreal (2012 y 2013).

Mientras trabajaba para Hex-Rays SA en Bélgica, Elias ayudó a mejorar y agregar nuevas funciones a IDA Pro. Durante ese período, escribió varias publicaciones de blogs técnicos, brindó capacitación sobre IDA Pro, desarrolló varios complementos de depuración, mejoró las funciones de scripting de IDA Pro y contribuyó al proyecto IDAPython desde la versión 1.2.0 en adelante. Elias actualmente trabaja en Microsoft con un talentoso equipo de ingenieros de seguridad de software.

viii Acerca de los autores

Sébastien Josse es investigador de seguridad en el Ministerio de Defensa francés (Direction Générale de l'Armement). Tiene más de diez años de experiencia como instructor, investigador y consultor en el campo de la seguridad de los sistemas de información, tanto en el sector civil como en el de defensa. Dedicó su tesis doctoral (École Polytechnique, 2009) al análisis dinámico de programas protegidos, centrándose principalmente en los mecanismos criptográficos resistentes a la ingeniería inversa y en el uso de un sistema de virtualización para llevar a cabo el análisis de programas protegidos. Ha publicado en la revista JICV y en varias actas de congresos , entre ellos ECRYPT (2004), EICAR (2006, 2008, 2011), AVAR (2007) y HICSS (2012, 2013 y 2014).



Acerca del editor técnico

Matt Miller es ingeniero de seguridad principal en la organización Trustworthy Computing de Microsoft, donde actualmente se centra en la investigación y el desarrollo de tecnología de mitigación de vulnerabilidades. Antes de unirse a Microsoft, Matt fue desarrollador principal del marco Metasploit y colaborador de la revista Uninformed, donde escribió sobre temas relacionados con la explotación, la ingeniería inversa, el análisis de programas y los aspectos internos de los sistemas operativos.

Créditos

Editor ejecutivo	Gerente de marketing
Carol larga	Ashley Zurcher
Editor de proyectos	Gerente de negocios
Juan Sleeva	Amy Knies
Editor técnico	Vicepresidente y Ejecutivo
Matt Miller	Editor del grupo
Editor de producción	Richard Swadley
Daniel Scribner	Editor asociado
Editor de copias	Jim Minatel
Luann Rouff	Coordinador de Proyectos, Cover
Gerente editorial	Todd Klemme
Mary Beth Wakefield	Corrector de pruebas
Gerente editorial independiente	Josh Chase, Word One Nueva York
Rosemarie Graham	Indexador
Director Asociado de	Ron Strauss
Marketing	Diseñador de portadas
David Mayhew	Ryan Sneed



Expresiones de gratitud

Escribir este libro ha sido una de las tareas más interesantes y que más tiempo nos ha llevado. El libro representa algo que nos hubiera gustado tener cuando empezamos a aprender sobre ingeniería inversa hace más de 15 años. En aquel momento, había una escasez de libros y recursos en línea (no había blogs en aquel entonces); aprendimos el arte principalmente a través de amigos y experimentos independientes de prueba y error. La “industria” de la seguridad de la información también era inexistente en aquel entonces. Hoy, el mundo es diferente. Ahora tenemos descompiladores, escáneres web, escáneres de código fuente estático, nube (?) y APT (¡impensable!). Numerosos blogs, foros, libros y clases presenciales tienen como objetivo enseñar ingeniería inversa. Estos recursos varían mucho en calidad. Algunos son de calidad inferior pero se publican o se ofrecen descaradamente para aprovechar el aumento de la demanda de seguridad informática; algunos son de calidad extremadamente alta pero no están bien atendidos. Muchos de ellos se leen por falta de publicidad, especialización o simplemente porque son “demasiado esotéricos”. No existe un recurso unificador que la gente pueda usar como base para aprender ingeniería inversa. Esperamos que este libro sea esa base.

Ahora, la mejor parte es reconocer a las personas que nos ayudaron a llegar a donde estamos hoy. Todos los autores desean agradecer a Rolf Rolles por sus contribuciones al capítulo sobre ofuscación. Rolf es un verdadero pionero en el campo de la ingeniería inversa. Su trabajo seminal sobre la desofuscación de máquinas virtuales, la aplicación del análisis de programas a la ingeniería inversa y la educación en análisis binario influyeron e inspiraron a una nueva generación de ingenieros inversos. Esperamos que continúe contribuyendo al campo e inspire a otros a hacer lo mismo. A continuación, también nos gustaría agradecer a Matt Miller, nuestro camarada y revisor técnico. Matt es otro verdadero pionero en nuestro campo y ha hecho contribuciones seminales para mitigaciones de exploits en Windows. Su dedicación a los detalles y a ayudar a otros a aprender debería ser un modelo para todos. Finalmente, nos gustaría agradecer a Carol Long, John Sleeva, Luann Rouff y al personal de John Wiley & Sons por soportarnos durante el proceso de publicación.

— Los autores

xiv Agradecimientos

Me gustaría agradecer a mis padres por sus sacrificios para darme mejores oportunidades en la vida; a mi hermana y hermano, Ivy Dang y Donald Dang, por ser una fuente constante de apoyo e inspiración; y a Rolf Rolles por ser un buen amigo y una fuente de razón durante todos estos años. No tuve muchos modelos a seguir mientras crecí, pero las siguientes personas ayudaron directamente a dar forma a mis perspectivas: Le Thanh Sang, Vint Cerf y Douglas Comer. En la universidad, aprendí la alegría de la literatura china de David Knetchges, los estudios budistas de Kyoko Tokuno, la historia de la India de Richard Salomon (¡quién hubiera pensado que se puede aprender tanto de las piedras y las monedas!), la historia de Asia Central de Daniel Waugh y el idioma chino de Nyan-Ping Bi. Si bien no son ingenieros inversos, su entusiasmo y dedicación me inspiraron para siempre y me hicieron un mejor ser humano e ingeniero. Si los hubiera conocido antes, mi trayectoria profesional probablemente sería muy diferente.

A lo largo de mi trayectoria profesional, tuve la suerte de conocer a personas inteligentes que me influyeron (sin ningún orden en particular): Alex Carp, rebel, Navin Pai, Jonathan Ness, Felix Domke, Karl J., Julien Tinnes, Josh Phillips, Daniel Radu, Maarten Boone, Yoann Guillot, Ivanlef0u (gracias por hospedarnos), Richard van Eeden, Dan Ho, Andy Renk, Elia Florio, Ilfak Guifanov, Matt Miller, David Probert, Damian Hasse, Matt Thomlinson, Shawn Hoffman, David Dittrich, Eloi Vanderbeken, LMH, Ali Rahbar, Fermin Serna, Otto Kivling, Damien Aumaitre, Tavis Ormandy, Ali Pezeshk, Gynvael Coldwind, anakata (un genio poco común), Richard van Eeden, Noah W., Ken Johnson, Chengyun Yu, Elias Bachaalany, Felix von Leitner, Michal Chmielewski, sectorx, Son Pho Nguyen, Nicolas Pouvesle, Kostya Kortchinsky, Peter Viscerola, Torbjorn L., Gustavo di Scotti, Sergiusz Fonrobert, Peter W., Ilya van Sprundel, Brian Cavenah, upb, Maarten Van Horenbeeck, Robert Hensing, Cristian Craioveanu, Claes Nyberg, Igor Skorchinsky, John Lambert, Mark Wodrich (budista modelo a seguir), David Midturi, Gavin Thomas, Sebastian Porst, Peter Vel, Kevin Broas, Michael Sandy, Christer Oberg, Mateusz "j00ru"

Jurczyk, David Ross y Raphael Rigo. Jonathan Ness y Damian Hasse siempre me apoyaron para que hiciera las cosas de manera diferente y constantemente me dieron oportunidades de fracasar o triunfar. Si me olvidé de ustedes, perdónenme.

Las siguientes personas me brindaron directamente sus comentarios y mejoraron los borradores iniciales de mis capítulos: Michal Chmielewski, Shawn Hoffman, Nicolas Pouvesle, Matt Miller, Alex Ionescu, Mark Wodrich, Ben Byer, Felix Domke, Ange Albertini, Igor Skorchinsky, Peter Ferrie, Lien Duong, iZsh, Frank Boldewin, Michael Hale Ligh, Sébastien Renaud, Billy McCourt, Peter Viscerola, Dennis Elser, Thai Duong, Eloi Vanderbeken, Raphael Rigo, Peter Vel y Bradley Spengler (un verdadero triunfador). Sin sus perspicaces comentarios y sugerencias, la mayor parte del libro sería ilegible. Por supuesto, puedes culparme por los errores restantes.

Hay muchas otras personas anónimas que contribuyeron a mi conocimiento. y por eso este libro.

También quiero agradecer a Molly Reed y Tami Needham de The Omni Group por darnos una licencia de OmniGraffle para hacer ilustraciones en los borradores anteriores.

Por último, pero no por ello menos importante, quiero agradecer a Alex, Elias y Sébastien por ayudarme con este libro. Sin ellos, el libro nunca habría visto la luz.

—Bruce

En primer lugar, me gustaría agradecer a Bruce Dang por invitarme a participar en este gran proyecto. Ha sido un viaje largo y enriquecedor. Rolf Rolles estuvo allí al principio, y personalmente le agradezco las incontables horas que pasamos juntos imaginando el capítulo de la ofuscación y recopilando material. Sébastien Josse aceptó unirse a nosotros; su contribución es invaluable y nuestro capítulo no sería lo mismo sin él. Gracias, Seb.

También quiero agradecer a mis amigos Fabrice Desclaux, Yoann Guillot y Jean-Agradecemos a Philippe Luyten sus valiosos comentarios.

Por último, gracias a Carol Long por hacer posible este libro, y a John Sleeva por mantenernos en el buen camino.

— Alejandro

Quiero empezar agradeciendo a Bruce Dang, mi amigo y colega, por darme la oportunidad de participar en este proyecto. También quiero agradecer a todos mis amigos y colegas por su apoyo y ayuda. En particular, me gustaría agradecer a Daniel Pistelli (director ejecutivo de Cerbero GmbH), Michal Chmielewski, Swamy Shivaganga Nagaraju y Alexandre Gazet por sus aportes técnicos y comentarios durante la redacción del libro.

Quiero agradecer al Sr. Ilfak Guifanov (director ejecutivo de Hex-Rays SA). Aprendí mucho de él mientras trabajaba en Hex-Rays. Su arduo trabajo, paciencia y perseverancia para crear IDA Pro siempre serán una inspiración para mí.

Queremos agradecer enormemente a John Wiley & Sons por darnos la oportunidad de publicar este libro. También queremos agradecer a la editora de adquisiciones Carol Long por su ayuda rápida y profesional, y al editor de proyectos John Sleeva y a la correctora de estilo Luann Rouff por su energía, paciencia y trabajo duro.

— Elías

Quiero agradecer a Alexandre, Elias y Bruce por darme la oportunidad de contribuir a este libro. También quiero agradecer a Jean-Philippe Luyten por ponernos en contacto. Por último, gracias a Carol Long y John Sleeva por su ayuda y profesionalismo en la realización de este proyecto.

— Sébastien



Contenido de un vistazo

Introducción	xxiii
Capítulo 1 x86 y x64	1
Capítulo 2 ARM	39
Capítulo 3 El núcleo de Windows	87
Capítulo 4 Depuración y automatización	187
Capítulo 5 Ofuscación	267
Apéndice Ejemplos de nombres y hashes SHA1 correspondientes	341
Índice	343

Contenido

Introducción	xxiii
Capítulo 1 x86 y x64	1
Conjunto de registros y tipos de datos	2
Conjunto de instrucciones	3
Sintaxis	4
Movimiento de datos	5
Ejercicio	11
Operaciones aritméticas	11
Operaciones de pila e invocación de funciones	13
Ceremonias	17
Flujo de control	17
Mecanismo del sistema	25
Traducción de direcciones	26
Interrupciones y excepciones	27
Tutorial paso a paso	28
Ejercicios	35
x64	36
Conjunto de registros y tipos de datos	36
Movimiento de datos	36
Dirección canónica	37
Invocación de función	37
Ceremonias	38
Capítulo 2 ARM	39
Características básicas	40
Tipos de datos y registros	43
Controles y configuraciones a nivel de sistema	45
Introducción al conjunto de instrucciones	46

Carga y almacenamiento de datos	47
LDR y STR	47
Otros usos de LDR	51
LDM y STM	52
EMPUJAR y POP	56
Funciones e invocación de funciones	57
Operaciones aritméticas	60
Ramificación y ejecución condicional	61
Estado del pulgar	64
Caja del interruptor	65
Misceláneas	67
Código automodificable y justo a tiempo	67
Primitivas de sincronización	67
Servicios y mecanismos del sistema	68
Instrucciones	70
Tutorial paso a paso	71
Próximos pasos	77
Ceremonias	78
 Capítulo 3 El núcleo de Windows	 87
Fundamentos de Windows	88
Disposición de la memoria	88
Inicialización del procesador	89
Llamadas al sistema	92
Nivel de solicitud de interrupción	104
Memoria de piscina	106
Listas de descriptores de memoria	106
Procesos y subprocesos	107
Contexto de ejecución	109
Primitivas de sincronización del núcleo	110
Liza	111
Detalles de implementación	112
Tutorial paso a paso	119
Ceremonias	123
Ejecución asincrónica y ad hoc	128
Hilos del sistema	128
Elementos de trabajo	129
Llamadas a procedimientos asincrónicos	131
Llamadas a procedimientos diferidos	135
Temporizadores	140
Devoluciones de llamadas de procesos y subprocesos	142
Rutinas de finalización	143
Paquetes de solicitud de E/S	144
Estructura de un controlador	146
Puntos de entrada	147
Objetos de controlador y dispositivo	149

Manejo de IRP	150
Un mecanismo común para la comunicación entre el usuario y el núcleo	150
Mecanismos de sistemas diversos	153
Recorridos paso a paso	155
Un rootkit x86	156
Un rootkit x64	172
Próximos pasos	178
Ceremonias	180
Generando confianza y solidificando	
Tu conocimiento	180
Investigando y ampliando sus conocimientos	182
Análisis de factores de la vida real	184
Capítulo 4 Depuración y automatización	187
Las herramientas de depuración y los comandos básicos	188
Configuración de la ruta del símbolo	189
Ventanas del depurador	189
Evaluando expresiones	190
Control de procesos y eventos de debut	194
Registros, memoria y símbolos	198
Puntos de interrupción	208
Inspección de procesos y módulos	211
Comandos varios	214
Creación de scripts con herramientas de depuración	216
Pseudo-registros	216
Alias	219
Idioma	226
Archivos de script	240
Uso de scripts como funciones	244
Ejemplos de scripts de depuración	249
Uso del SDK	257
Conceptos	258
Cómo escribir extensiones para herramientas de depuración	262
Extensiones, herramientas y recursos útiles	264
Capítulo 5 Ofuscación	267
Un estudio de las técnicas de ofuscación	269
La naturaleza de la ofuscación: un ejemplo motivador	269
Ofuscaciones basadas en datos	273
Ofuscación basada en control	278
Flujo de control y flujo de datos simultáneos	
Ofuscación	284
Lograr la seguridad mediante la oscuridad	288
Un estudio de las técnicas de desofuscación	289
La naturaleza de la desofuscación: inversión de la transformación	289
Herramientas de desofuscación	295
Desofuscación práctica	312

Estudio de caso	328
Primeras impresiones	328
Análisis de la semántica de los controladores	330
Ejecución simbólica	333
Resolviendo el desafío	334
Reflexiones finales	336
Ceremonias	336
Apéndice Ejemplos de nombres y hashes SHA1 correspondientes	341
Índice	343

Introducción

El proceso de aprendizaje de la ingeniería inversa es similar al de la adquisición de una lengua extranjera para adultos. La primera fase del aprendizaje de una lengua extranjera comienza con una introducción a las letras del alfabeto, que se utilizan para construir palabras con una semántica bien definida. La siguiente fase implica la comprensión de las reglas gramaticales que rigen cómo se unen las palabras para producir una oración adecuada. Una vez que uno se acostumbra a estas reglas, aprende a unir varias oraciones para articular pensamientos complejos. Finalmente, se llega al punto en que el alumno puede leer libros extensos escritos en diferentes estilos y aun así entender los pensamientos que contienen. En este punto, uno puede leer libros de referencia sobre los aspectos más esotéricos de la lengua: sintaxis histórica, fonología, etc.

En ingeniería inversa, el lenguaje es la arquitectura y el lenguaje ensamblador . Una palabra es una instrucción ensambladora. Los párrafos son secuencias de instrucciones ensambladoras. Un libro es un programa. Sin embargo, para comprender completamente un libro, el lector necesita saber más que solo vocabulario y gramática. Estos elementos adicionales incluyen la estructura y el estilo de la prosa, reglas no escritas de escritura y otros. Comprender los programas informáticos también requiere un dominio de conceptos más allá de las instrucciones ensambladoras.

Puede resultar un tanto intimidante comenzar a aprender un tema técnico completamente nuevo a partir de un libro. Sin embargo, lo engañaríamos si afirmáramos que la ingeniería inversa es una tarea de aprendizaje sencilla y que se puede dominar por completo leyendo este libro. El proceso de aprendizaje es bastante complejo porque requiere conocimientos de varios dominios de conocimiento dispares. Por ejemplo, un ingeniero inverso eficaz debe tener conocimientos de arquitectura informática, programación de sistemas, sistemas operativos, compiladores, etc.; para ciertas áreas, es necesario un sólido conocimiento matemático. Entonces, ¿cómo se hace?

¿Sabes por dónde empezar? La respuesta depende de tu experiencia y tus habilidades. Como no podemos adaptarnos a los antecedentes de todos, esta introducción describe los métodos de aprendizaje y lectura para quienes no tienen conocimientos de programación. Debes encontrar tu “posición” en el espectro y comenzar desde allí.

Para efectos de discusión, definimos libremente la ingeniería inversa como el proceso de comprensión de un sistema. Es un proceso de resolución de problemas. Un sistema puede ser un dispositivo de hardware, un programa de software, un proceso físico o químico, etc. Para los fines de este libro, el sistema es un programa de software. Para entender un programa, primero debes entender cómo se escribe el software. Por lo tanto, el primer requisito es saber cómo programar una computadora a través de un lenguaje como C, C++, Java y otros. Sugerimos aprender primero C debido a su simplicidad, efectividad y ubicuidad. Algunas referencias excelentes para considerar son *The C Programming Language*, de Brian Kernighan y Dennis Ritchie (Prentice Hall, 1988) y *C: A Reference Manual*, de Samuel Harbison (Prentice Hall, 2002). Después de sentirse cómodo con la escritura, compilación y depuración de programas básicos, considere leer *Expert C Programming: Deep C Secrets*, de Peter van der Linden (Prentice Hall, 1994). En este punto, debe estar familiarizado con conceptos de alto nivel como variables, ámbitos, funciones, punteros, condicionales, bucles, pilas de llamadas y bibliotecas. El conocimiento de estructuras de datos como pilas, colas, listas enlazadas y árboles puede ser útil, pero no es completamente necesario por ahora. Para completar, puede leer *Compilers: Principles, Techniques, and Tools*, de Alfred Aho, Ravi Sethi y Jeffrey Ullman (Prentice Hall, 1994) y *Linkers and Loaders*, de John Levine (Morgan Kaufmann, 1999), para comprender mejor cómo se construye realmente un programa. El objetivo principal de leer estos libros es familiarizarse con los conceptos básicos; no es necesario que comprenda cada página por ahora (ya habrá tiempo para eso más adelante).

Los estudiantes con un alto rendimiento deberían considerar el libro *Advanced Compiler Design and Implementation*, de Steven Muchnick (Morgan Kaufmann, 1997).

Una vez que comprenda bien cómo se escriben, ejecutan y depuran los programas, debería comenzar a explorar el entorno de ejecución del programa, que incluye el procesador y el sistema operativo. Le sugerimos que primero aprenda sobre el procesador Intel leyendo rápidamente *Intel 64 e IA-32*.

Manual del desarrollador de software de arquitecturas, volumen 1: Arquitectura básica de Intel, con especial atención a los capítulos 2 a 7. Estos capítulos explican los elementos básicos de una computadora moderna. Los lectores interesados en ARM deberían considerar la serie *Cortex-A Guía del programador y manual de referencia de la arquitectura ARM ARMv7-A y ARMv7-R Edición de ARM*. Si bien nuestro libro cubre x86/x64/ARM, no analizamos todos los detalles arquitectónicos. (Suponemos que el lector consultará estos manuales, según sea necesario). Al hojear estos manuales, debe tener una comprensión básica de los componentes técnicos de un sistema informático. Para una comprensión más conceptual, considere *Structured Computer Organization* de Andrew Tanenbaum (Prentice Hall, 1998). Todos los lectores también deben consultar el Microsoft PE

y la especificación COFF. En este punto, tendrá todos los antecedentes necesarios para leer y comprender el Capítulo 1, “x86 y x64”, y el Capítulo 2, “ARM”.

A continuación, debe explorar el sistema operativo. Hay muchos sistemas operativos diferentes, pero comparten muchos conceptos comunes, incluidos procesos, subprocesos, memoria virtual, separación de privilegios, multitarea, etc. La mejor manera de comprender estos conceptos es leer *Modern Operating Systems*, de Andrew Tanenbaum (Prentice Hall, 2005). Aunque el texto de Tanenbaum es excelente para los conceptos, no analiza detalles técnicos importantes para los sistemas operativos de la vida real. Para Windows, debe considerar la posibilidad de leer rápidamente *NT Device Driver Development*, de Peter Viscarola y Anthony Mason (New Riders Press, 1998); aunque se trata de un libro sobre el desarrollo de controladores, los capítulos de introducción proporcionan una introducción excelente y concreta a Windows. (También es un excelente material complementario para el capítulo sobre el núcleo de Windows de este libro). Para obtener más inspiración (y un excelente tratamiento del administrador de memoria de Windows), también debería leer *¿Qué lo hace una página? La memoria virtual de Windows 7 (x64) Memory Manager*, de Enrico Martignetti (CreateSpace Independent Publishing Platform, 2012).

En este punto, ya tendrías todos los conocimientos necesarios para leer y comprender el Capítulo 3 “El núcleo de Windows”. También deberías considerar aprender a programar en Win32. *Windows System Programming*, de Johnson Hart (Addison-Wesley Professional, 2010), y *Windows via C/C++*, de Jeffrey Richter y Christophe Nasarre (Microsoft Press, 2007), son excelentes referencias.

Para el Capítulo 4, “Depuración y automatización”, considere *Inside Windows Depuración: una guía práctica para estrategias de depuración y seguimiento en Windows*, por Tarik Soulami (Microsoft Press, 2012), y *Depuración avanzada de Windows*, por Mario Hewardt y Daniel Pravat (Addison-Wesley Professional, 2007).

El capítulo 5, “Ofuscación”, requiere una buena comprensión del lenguaje ensamblador y debe leerse después de los capítulos x86/x64/ARM. Para obtener conocimientos básicos, consulte *Software subrepticio: Ofuscación, marcas de agua y protección contra manipulaciones para Protección de software*, por Christian Collberg y Jasvir Nagra (Addison-Wesley Professional, 2009).

NOTA: Este libro incluye ejercicios y tutoriales con virus y rootkits maliciosos reales. Hicimos esto intencionalmente para asegurar que los lectores puedan aplicar de inmediato las habilidades recién aprendidas. Las muestras de malware están referenciadas en orden alfabético (muestra A, B, C, ...) y puede encontrar los hashes SHA1 correspondientes en el Apéndice. Dado que puede haber problemas legales sobre la distribución de dichas muestras junto con el libro, decidimos no hacerlo; sin embargo, puede descargarlas buscando en varios repositorios de malware, como www.malware.lu, o solicitarlas en los foros de <http://kernelmode.info>. Muchas de las muestras proceden de famosos incidentes de piratería informática que aparecieron en las noticias de todo el mundo, por lo que deberían resultar interesantes. Tal vez algunos lectores entusiastas recopilen todas las muestras en un paquete y las compartan en BitTorrent.

Si ninguna de estas opciones le funciona, no dude en enviar un correo electrónico a los autores. Asegúrese de analizarlas en un entorno seguro para evitar una autoinfección accidental.

Además, para familiarizarte con Metasm, hemos preparado dos guiones de ejercicios: ejecución simbólica-lvl1.rb y ejecución simbólica-lvl2.rb.

Responder las preguntas lo llevará a un viaje por los aspectos internos de Metasm.

Puede encontrar los guiones en www.wiley.com/go/practicalreverseengineering.

Es importante darse cuenta de que los ejercicios son un componente vital del libro. El libro fue escrito intencionalmente de esta manera. Si simplemente lees el libro sin hacer los ejercicios, no entenderás ni retendrás mucho. Deberías sentirte libre de escribir en un blog o escribir sobre tus respuestas para que otros puedan aprender de ellas; puedes publicarlas en el reddit de Ingeniería inversa (www.reddit.com/r/ReverseEngineering) y recibir comentarios de la comunidad (y tal vez de los autores). Si completas con éxito todos los ejercicios, date una palmadita en la espalda y luego envíale tu currículum a Bruce.

El camino para convertirse en un ingeniero inverso eficaz es largo y lleva mucho tiempo, y requiere paciencia y resistencia. Puede que fracase muchas veces en el camino (por no entender los conceptos o por no completar los ejercicios de este libro), pero no se rinda. Recuerde: el fracaso es parte del éxito. Con esta guía y los capítulos siguientes, debería estar bien preparado para el viaje de aprendizaje.

A nosotros, los autores, nos encantaría conocer su experiencia de aprendizaje para poder ajustar aún más nuestro material y mejorar el libro. Sus comentarios serán invaluables para nosotros y, potencialmente, para futuras publicaciones. Puede enviar comentarios y preguntas a Bruce Dang (bruce.dang@gmail.com), Alexandre Gazet (agazet@quarkslab.com), o Elias Bachaalany (elias.bachaalany@gmail.com).

CAPÍTULO

1

x86 y x64

La arquitectura x86 es little-endian y se basa en el procesador Intel 8086. Para los fines de este capítulo, x86 es la implementación de 32 bits de la arquitectura Intel (IA-32) como se define en el Manual de desarrollo de software de Intel. En términos generales, puede funcionar en dos modos: real y protegido. El modo real es el estado del procesador cuando se enciende por primera vez y solo admite un conjunto de instrucciones de 16 bits. El modo protegido es el estado del procesador que admite memoria virtual, paginación y otras funciones; es el estado en el que se ejecutan los sistemas operativos modernos. La extensión de 64 bits de la arquitectura se denomina x64 o x86-64. Este capítulo analiza la arquitectura x86 que funciona en modo protegido.

El sistema x86 admite el concepto de separación de privilegios mediante una abstracción denominada nivel de anillo. El procesador admite cuatro niveles de anillo, numerados del 0 al 3. (Los anillos 1 y 2 no se utilizan habitualmente, por lo que no se tratan aquí). El anillo 0 es el nivel de privilegio más alto y puede modificar todas las configuraciones del sistema. El anillo 3 es el nivel de privilegio más bajo y solo puede leer o modificar un subconjunto de las configuraciones del sistema. Por lo tanto, los sistemas operativos modernos suelen implementar la separación de privilegios de usuario y kernel .

haciendo que las aplicaciones en modo usuario se ejecuten en el anillo 3 y el núcleo en el anillo 0. El nivel del anillo está codificado en el registro CS y a veces se lo denomina nivel de privilegio actual (CPL) en la documentación oficial.

Este capítulo analiza la arquitectura x86/IA-32 tal como se define en el Manual del desarrollador de software de arquitecturas Intel 64 e IA-32, volúmenes 1 a 3 (www.intel.com.com/content/www/us/en/processors/architectures-software-developer-manuales.html).

Conjunto de registros y tipos de datos

Cuando funciona en modo protegido, la arquitectura x86 tiene ocho registros de propósito general (GPR) de 32 bits : EAX, EBX, ECX, EDX, EDI, ESI, EBP y ESP. Algunos de ellos se pueden dividir en registros de 8 y 16 bits. El puntero de instrucción se almacena en el registro EIP . El conjunto de registros se ilustra en la Figura 1-1. La Tabla 1-1 describe algunos de estos GPR y cómo se utilizan.



Tabla 1-1: Algunos GPR y su uso

PROPSITO DEL REGISTRO

EAX	Contador en bucles
ESI	Fuente en operaciones de cadena/memoria
EDI	Destino en operaciones de cadena/memoria
EBP	Puntero del marco base
ESP	Puntero de pila

Los tipos de datos comunes son los siguientes:

- Bytes: 8 bits. Ejemplos: AL, BL, CL
- Palabra: 16 bits. Ejemplos: AX, BX, CX
- Palabra doble: 32 bits. Ejemplos: EAX, EBX, ECX
- Palabra cuádruple: 64 bits. Si bien x86 no tiene GPR de 64 bits, puede combinar dos registros, generalmente EDX:EAX, y tratarlos como valores de 64 bits en algunos escenarios . Por ejemplo, la instrucción RDTSC escribe un valor de 64 bits en EDX:EAX.

El registro EFLAGS de 32 bits se utiliza para almacenar el estado de las operaciones aritméticas y otros estados de ejecución (por ejemplo, el indicador de trap). Por ejemplo, si la operación de “adición” anterior dio como resultado un cero, el indicador ZF se establecerá en 1. Los indicadores en EFLAGS se utilizan principalmente para implementar la ramificación condicional.

Además de los GPR, EIP y EFLAGS, también hay registros que controlan mecanismos importantes del sistema de bajo nivel, como la memoria virtual, las interrupciones y la depuración. Por ejemplo, CR0 controla si la paginación está activada o desactivada, CR2 contiene la dirección lineal que causó un error de paginación, CR3 es la dirección base de una estructura de datos de paginación y CR4 controla la configuración de virtualización del hardware. DR0–DR7

Se utilizan para establecer puntos de interrupción de la memoria. Volveremos a estos registros más adelante en la sección “Mecanismo del sistema”.

NOTA: Aunque hay ocho registros de depuración, el sistema solo permite cuatro puntos de interrupción de memoria (DR0–DR3). Los registros restantes se utilizan para el estado.

También existen registros específicos del modelo (MSR). Como su nombre lo indica, estos registros pueden variar entre los distintos procesadores de Intel y AMD. Cada MSR se identifica por su nombre y un número de 32 bits, y se lee y escribe en él mediante las instrucciones RDMSR/WRMSR . Solo son accesibles para el código que se ejecuta en el anillo 0 y, por lo general, se utilizan para almacenar contadores especiales e implementar funciones de bajo nivel.

Por ejemplo, la instrucción SYSENTER transfiere la ejecución a la dirección almacenada en el MSR IA32_SYSENTER_EIP (0x176), que suele ser el controlador de llamadas del sistema del sistema operativo. Los MSR se analizan a lo largo del libro a medida que aparecen.

Conjunto de instrucciones

El conjunto de instrucciones x86 permite un alto nivel de flexibilidad en términos de movimiento de datos entre registros y memoria. El movimiento se puede clasificar en cinco métodos generales:

- Registro inmediato
- Regístrate para registrarte
- Inmediato a la memoria

- Registrar en la memoria y viceversa
- De memoria a memoria

Los primeros cuatro métodos son compatibles con todas las arquitecturas modernas, pero el último es específico de x86. Una arquitectura RISC clásica como ARM solo puede leer/escribir datos desde/hacia la memoria con instrucciones de carga/almacenamiento (LDR y STR, respectivamente); por ejemplo, una operación simple como incrementar un valor en la memoria requiere tres instrucciones:

1. Leer los datos de la memoria a un registro (LDR).
2. Añade uno al registro (ADD).
3. Escribe el registro en la memoria (STR).

En x86, una operación de este tipo requeriría solo una instrucción (ya sea INC o ADD) porque puede acceder directamente a la memoria. La instrucción MOVS puede leer y escribir en la memoria al mismo tiempo.

BRAZO		
01:1A 68	LDR R2, [R3]; leer el	
	valor en la dirección R3 y guardarlo en R202;	
52 1C	SUMA R2, R2, #1; súmale 1	
03:1A 60	STR R2, [R3]	
	; escribe el valor actualizado en la dirección R3	
x86		
01: FF00	C#	palabra clave d [eax]
		; incrementar directamente el valor en la dirección EAX

Otra característica importante es que x86 utiliza instrucciones de longitud variable : la longitud de las instrucciones puede variar entre 1 y 15 bytes. En ARM, las instrucciones tienen una longitud de 2 o 4 bytes.

Sintaxis

Dependiendo del ensamblador/desensamblador, hay dos notaciones de sintaxis para el código ensamblador x86, Intel y AT&T:

Intel	
movimiento ecx, AABBCCDDh	
movimiento ecx, [eax]	
movimiento ecx, eax	
AT&T	
movimiento \$0xAABBCCDD, %ecx	
movl (%eax), %ecx	
movimiento %eax, %ecx	

Es importante tener en cuenta que se trata de las mismas instrucciones, pero escritas de forma diferente . Existen varias diferencias entre la notación de Intel y la de AT&T, pero las más notables son las siguientes:

- AT&T antepone el registro con %, y los inmediatos con \$. Intel no Haz esto.
- AT&T agrega un sufijo a la instrucción para indicar el ancho de la operación. Por ejemplo, MOVL (largo), MOVB (byte), etc. Intel no hace esto.
- AT&T coloca el operando de origen antes del de destino. Intel invierte el orden.

Los desensambladores/ensambladores y otras herramientas de ingeniería inversa (IDA Pro, OllyDbg, MASM, etc.) en Windows suelen utilizar la notación Intel, mientras que los de UNIX suelen utilizar la notación AT&T (GCC). En la práctica, la notación Intel es la forma dominante y se utiliza en todo este libro.

Movimiento de datos

Las instrucciones operan sobre valores que provienen de registros o de la memoria principal. La instrucción más común para mover datos es MOV. El uso más simple es mover un registro o un registro inmediato a otro. Por ejemplo:

```
01: BE 3F 00 0F 00 mov esi, 0F003Fh ; establecer ESI = 0xF003
02:8B F1           movimiento esi, ecx ; establecer ESI = ECX
```

El siguiente uso común es mover datos hacia o desde la memoria. De manera similar a otras convenciones del lenguaje ensamblador, x86 utiliza corchetes ([]) para indicar el acceso a la memoria. (La única excepción a esto es la instrucción LEA , que utiliza [] pero en realidad no hace referencia a la memoria). El acceso a la memoria se puede especificar de varias maneras diferentes, por lo que comenzaremos con el caso más simple:

Asamblea

```
01: C7 00 01 00 00+ movimiento dword ptr [eax], 1
; establece la memoria en la dirección EAX en 1
02:8B08           movimiento ecx, [eax]
; establece ECX en el valor en la dirección EAX
03:8918           movimiento [eax], ebx
; establece la memoria en la dirección EAX a EBX
04:894634         movimiento [esi+34h], eax
; establece la dirección de memoria en (ESI+34) en EAX
05:8B4634         movimiento eax, [esi+34h]
; establece EAX en el valor en la dirección (ESI+0x34)
06:8B 14 01       movimiento edx, [ecx+eax]
; establece EDX en el valor en la dirección (ECX+EAX)
```

Pseudo C

```

01: *eax = 1;
02: ecx = *eax;
03: eax = ebx;
04: *(esi+0x34) = eax;
05: eax = *(esi+0x34);
06: edx = *(ecx+eax);
```

Estos ejemplos demuestran el acceso a la memoria a través de un registro base y un desplazamiento, donde el desplazamiento puede ser un registro o inmediato. Esta forma se utiliza comúnmente para acceder a miembros de la estructura o búferes de datos en una ubicación calculada en tiempo de ejecución. Por ejemplo, supongamos que ECX apunta a una estructura de tipo KDPC con el diseño

```

kd>dt nt!_KDPC
+0x000 Tipo :UCar
+0x001 Importancia :UCar
+0x002 Número :UInt2B
+0x004 Entrada de lista Dpc :_ENTRADA_DE_LISTA
+0x00c Rutina diferida: Ptr32 vacío
+0x010 DeferredContext : Ptr32 Vacío
+0x014 Argumento del sistema1: Ptr32 nulo
+0x018 Argumento del sistema2: Ptr32 nulo
+0x01c Datos de Dpc : Ptr32 Vacío
```

y se utiliza en el siguiente contexto:

Asamblea

```

01:8B 45 0C           movimiento eax, [ebp+0Ch]
02:83 E1 1C 00         y dword ptr [ecx+1Ch], 0
03:89 41 0C           movimiento [ecx+0Ch], eax
04:8B4510             movimiento eax, [ebp+10h]
05: C7 01 13 01 00+   movimiento dword ptr [ecx], 113h
06:894110             movimiento [ecx+10h], ax
```

Pseudo C

```

KDPC *p = ...;
p->DpcData = NULL;
p->RutinaDiferida = ...;
*(int *)p = 0x113;
p->ContextoDiferido = ...;
```

La línea 1 lee un valor de la memoria y lo almacena en EAX. La rutina diferida
El campo se establece en este valor en la línea 3. La línea 2 borra el campo DpcData mediante la operación AND.

con 0. La línea 4 lee otro valor de la memoria y lo almacena en EAX. El campo DeferredContext se establece en este valor en la línea 6.

La línea 5 escribe el valor de palabra doble 0x113 en la base de la estructura. ¿Por qué escribe un valor de palabra doble en la base si el primer campo tiene un tamaño de solo 1 byte? ¿Eso no establecería implícitamente también los campos Importancia y Número? La respuesta es sí. La Figura 1-2 muestra el resultado de convertir 0x113 a binario.

	00000000 00000000 00000001 00010011	
00000000 00000000	00000001	00010011
Número	Importancia	Tipo

Figura 1-2

El campo Tipo se establece en 0x13 (bits en negrita), Importancia se establece en 0x1 (bits en cursiva) y Número se establece en 0x0 (los bits restantes). Al escribir un valor, el código logró inicializar tres campos con una sola instrucción. El código podría haberse escrito de la siguiente manera:

01:8B 45 0C	movimiento eax, [ebp+0Ch]
02:83 61 1C 00	y dword ptr [ecx+1Ch], 0
03:89 41 0C	movimiento [ecx+0Ch], eax
04:8B4510	movimiento eax, [ebp+10h]
05: C6 01 13	movimiento byte ptr [ecx],13h
06: C6 41 01 01	movimiento byte ptr [ecx+1],1
07:66 C7 41 02 00+ movimiento palabra ptr [ecx+2],0	
08:89 41 10	movimiento [ecx+10h], ax

El compilador decidió combinar tres instrucciones en una porque conocía las constantes de antemano y quería ahorrar espacio. La versión de tres instrucciones ocupa 13 bytes (no se muestra el byte adicional en la línea 7), mientras que la versión de una instrucción ocupa 6 bytes. Otra observación interesante es que el acceso a la memoria se puede realizar en tres niveles de granularidad: byte (líneas 5-6), palabra (línea 6) y palabra doble (líneas 1-4, 8). La granularidad predeterminada es de 4 bytes, que se puede cambiar a 1 o 2 bytes con un prefijo de anulación. En el ejemplo, el byte del prefijo de anulación es 66 (en cursiva). Se comentan otros prefijos a medida que aparecen.

La siguiente forma de acceso a la memoria se utiliza comúnmente para acceder a objetos de tipo matriz. En general, el formato es el siguiente: [Base + Índice * escala]. Esto se entiende mejor con ejemplos:

```
01: 8B 34 B5 40 05+ mov esi, _KdLogBuffer[esi*4]
; siempre escrito como mov esi, [_KdLogBuffer + esi * 4]
; _KdLogBuffer es la dirección base de una matriz global y
; ESI es el índice; sabemos que cada elemento de la matriz
; tiene una longitud de 4 bytes (de ahí el factor de escala)
```

```

02:89 04 F7          movimiento [edi+esi*8], eax
; aquí EDI es la dirección base de la matriz; ESI es la matriz
; índice; el tamaño del elemento es 8.

```

En la práctica, esto se observa en el código que recorre una matriz. Por ejemplo:

```

01:                      inicio_de_bucle:
02:8B4704              movimiento eax, [edi+4]
03:8B 04 98              movimiento eax, [eax+ebx*4]
04:85 C0              prueba eax, eax
...
05:7414              jz corto loc_7F627F
06:                      ubicación_7F627F:
07:43              incluye ebx
08:3B 1F              cmp ebx, [edi]
09:7C DD              jl      inicio de bucle corto

```

La línea 2 lee una palabra doble del desplazamiento +4 desde EDI y luego la usa como dirección base en una matriz en la línea 3; por lo tanto, sabes que EDI es probablemente una estructura que tiene una matriz en +4. La línea 7 incrementa el índice. La línea 8 compara el índice con un valor en el desplazamiento +0 en la misma estructura. Dada esta información, este pequeño bucle se puede descompilar de la siguiente manera:

```

estructura de tipo definido _FOO
{
    Tamaño DWORD;           // +0x00
    Matriz DWORD[...]; // +0x04
} FOO, *PFOO;

barra PFOO = ....;
para (i = ...; i < barra->tamaño; i++) {
    si (barra->matriz[i] != 0) {
        ...
    }
}

```

Las instrucciones MOVS/MOVSW/MOVSD mueven datos con granularidad de 1, 2 o 4 bytes entre dos direcciones de memoria. Implícitamente utilizan EDI/ESI como dirección de origen/destino, respectivamente. Además, también actualizan automáticamente la dirección de origen/destino dependiendo del indicador de dirección (DF) en EFLAGS. Si DF es 1, las direcciones se decrementan; de lo contrario, se incrementan. Estas instrucciones se utilizan típicamente para implementar funciones de copia de cadena o de memoria cuando la longitud se conoce en tiempo de compilación. En algunos casos, van acompañadas del prefijo REP , que repite una instrucción hasta ECX veces.

Consideremos el siguiente ejemplo:

Asamblea

```

01: BE 28 B5 41 00 mov esi, desplazamiento _RamdiskBootDiskGuid
; ESI = puntero a RamdiskBootDiskGuid
02: 8D BD 40 FF FF+ edición limitada, [ebp-0C0h]
; EDI es una dirección en algún lugar de la pila
03: A5           movimiento sd
; copia 4 bytes de ESI a EDI; incrementa cada uno en 4
04: A5           movimiento sd
; lo mismo que arriba
05: A5           movimiento sd
; guardar como arriba
06: A5           movimiento sd
; lo mismo que arriba

```

Pseudo C

```

/* un GUID es una estructura de 16 bytes */
GUID RamDiskBootDiskGuid = ...; // global
...
GUID foo;
memcpy(&foo, &RamdiskBootDiskGuid, tamaño de(GUID));

```

La línea 2 merece una atención especial. Aunque la instrucción LEA utiliza [], en realidad no lee desde una dirección de memoria; simplemente evalúa la expresión entre corchetes y coloca el resultado en el registro de destino. Por ejemplo, si EBP fuera 0x1000, entonces EDI sería 0xF40 (=0x1000 – 0xC0) después de ejecutar la línea 2. El punto es que LEA no accede a la memoria, a pesar de la sintaxis engañososa.

El siguiente ejemplo, de nt!KilnItSystem, utiliza el prefijo REP :

```

01:6A 08           Empuja 8          ; empuja 8 en la pila (explicará las pilas)
; más tarde)
02: ...
03:59           estático      ecx ; abre la pila. Básicamente, establece ECX en 8.
04: ...
05:BE 00 44 61 00 mov      esi, desplazamiento _KeServiceDescriptorTable
06:BF C0 43 61 00 movimiento    editar, compensar _KeServiceDescriptorTableShadow
07:F3 A5           rep movsd ; copia 32 bytes (movsd se repite 8 veces)
; de esto podemos deducir que cualesquiera que sean estos dos objetos, son
; Probablemente tenga un tamaño de 32 bytes.

```

El equivalente aproximado en C de esto sería el siguiente:

```
memcpy(&KeServiceDescriptorTableShadow, &KeServiceDescriptorTable, 32);
```

10 Capítulo 1 ■ x86 y x64

El último ejemplo, nt!MmInitializeProcessAddressSpace, utiliza una combinación de estas instrucciones porque el tamaño de la copia no es múltiplo de 4:

```

01: 8D B0 70 01 00+ lea esi, [eax+170h]
; EAX es probablemente la dirección base de una estructura. Recuerde lo que dijimos
; acerca de LEA. ...
02: 8D BB 70 01 00+ lea edi, [ebx+170h]
; Es probable que EBX sea la dirección base de otra estructura del mismo tipo
03: A5           movimiento ad
04: A5           movimiento ad
05: A5           movimiento ad
06: 66 A5       movimiento
07: A4           movimiento sb

```

Después de las líneas 1 y 2, sabes que es probable que EAX y EBX sean del mismo tipo porque se utilizan como origen/destino y el desplazamiento es idéntico.

Este fragmento de código simplemente copia 15 bytes de un campo de estructura a otro. Tenga en cuenta que el código también podría haberse escrito utilizando la instrucción MOVS con un prefijo REP y ECX establecido en 15; sin embargo, eso sería ineficiente porque da como resultado 15 lecturas en lugar de solo cinco.

Otra clase de instrucciones de movimiento de datos con origen y destino implícitos incluye las instrucciones SCAS y STOS . De manera similar a MOVS, estas instrucciones pueden operar con una granularidad de 1, 2 o 4 bytes. SCAS compara implícitamente AL/AX/EAX con datos que comienzan en la dirección de memoria EDI; EDI se incrementa automáticamente/ Se decrementa en función del bit DF en EFLAGS. Dada su semántica, SCAS se utiliza comúnmente junto con el prefijo REP para buscar un byte, una palabra o una palabra doble en un búfer. Por ejemplo, la función strlen() de C se puede implementar de la siguiente manera:

```

01:30 C0          xor      al, al
; establece AL en 0 (byte NUL). Observarás con frecuencia el registro XOR, reg
; patrón en código.
02:89 FB          movimiento    ebx, edición
; guarda el puntero original a la cadena
03: F2 AE          costa de repne
; escanear hacia adelante repetidamente un byte a la vez hasta que AL no coincida con el
; byte en EDI cuando termina esta instrucción, significa que llegamos al byte NUL en
; el buffer de cadena
04:29 DF          sub      edición, ebx
; edi es ahora la ubicación del byte NUL. Resta eso del puntero original
; a la longitud.

```

STOS es lo mismo que SCAS, excepto que escribe el valor AL/AX/EAX en EDI. Se utiliza habitualmente para inicializar un búfer con un valor constante (como memset()). He aquí un ejemplo:

```

01:33 C0          xor      Ea, Ea
; establecer EAX en 0
02:6A 09          Empuja 9
; empuja 9 en la pila
03:59          estalido    Ceros (0)
; colóquelo nuevamente en ECX. Ahora ECX = 9.

```

04:8B FE	movimiento	editar, editar
: establecer la dirección de destino		
05: F3 AB	representante stosd	
: escribe 36 bytes de cero en el búfer de destino (STOSD repetido 9 veces)		
: esto es equivalente a prestar a memset(edi, 0, 36)		

LODS es otra instrucción de la misma familia. Lee un valor de 1, 2 o 4 bytes de ESI y lo almacena en AL, AX o EAX.

Ejercicio

1. Esta función utiliza una combinación de SCAS y STOS para realizar su trabajo. Primero, explique cuál es el tipo de [EBP+8] y [EBP+C] en las líneas 1 y 8, respectivamente.

A continuación, explique qué hace este fragmento.

01:8B7D08	edición móvil, [ebp+8]
02:8B D7	movimiento edx, edi
03:33 C0	xor eax, eax
04:83 C9 FF	o ecx, 0xFFFFFFFFh
05: F2 AE	costra de repne
06:83 C1 02	añadir ecx, 2
07: F7 D9	ecx negativo
08:8 a 45 0 C	movimiento al, [ebp+0Ch]
09:8B FA	edición mov, edx
10: F3 AA	representante stosb
11:8B C2	movimiento eax, edx

Operaciones aritméticas

Las operaciones aritméticas fundamentales, como la suma, la resta, la multiplicación y la división, están soportadas de forma nativa por el conjunto de instrucciones. Las operaciones a nivel de bits, como AND, OR, XOR, NOT y el desplazamiento a la izquierda y a la derecha, también tienen instrucciones nativas correspondientes . Con excepción de la multiplicación y la división, las demás instrucciones son sencillas en términos de uso. Estas operaciones se explican con los siguientes ejemplos:

01:83 C4 14	añadir esp, 14h sub ecx, eax	,esp = esp + 0x14 ; ecx = ecx - eax
02:28 C8		
03:83 CE 0C	sub esp, 0Ch inc ecx	,esp = esp - 0xC ; ecx = ecx + 1
04:41		
05:4F	edición de diciembre	; edi = edi-1
06:83 C8 FF	o eax, 0xFFFFFFFFh ; eax = eax 0xFFFFFFFF	
07:83 E1 07	y ecx, 7	; ecx = ecx y 7
08:33 C0	xor eax, eax no edi	; eax = eax ; edi = ~edi
09: F7 D7		
10: C0 E1 04	clase shl, 4	; cl = cl << 4
11: D1 E9	ecx de la esclerosis múltiple, 1	; ecx = ecx >> 1
12: C0 C0 03	rol al, 3	; rotar AL 3 posiciones hacia la izquierda
13: D0 C8	ror al, 1	; rotar AL 1 posición a la derecha

12 Capítulo 1 ■ x86 y x64

Las instrucciones de desplazamiento hacia la izquierda y hacia la derecha (líneas 11 y 12) merecen una explicación, ya que se observan con frecuencia en el código de la vida real. Estas instrucciones se utilizan normalmente para optimizar las operaciones de multiplicación y división en las que el multiplicando y el divisor son una potencia de dos. Este tipo de optimización se conoce a veces como reducción de fuerza porque reemplaza una operación computacionalmente costosa por una más barata. Por ejemplo, la división de números enteros es una operación relativamente lenta, pero cuando el divisor es una potencia de dos, se puede reducir a desplazar bits hacia la derecha; $100/2$ es lo mismo que $100 \gg 1$. De forma similar, la multiplicación por una potencia de dos se puede reducir a desplazar bits hacia la izquierda; $100*2$ es lo mismo que $100<<1$.

La multiplicación con y sin signo se realiza mediante las instrucciones MUL e IMUL , respectivamente. La instrucción MUL tiene la siguiente forma general: MUL reg/ memoria. Es decir, solo puede operar sobre valores de registros o de memoria. El registro se multiplica por AL, AX o EAX y el resultado se almacena en AX, DX:AX o EDX:EAX, según el ancho del operando. Por ejemplo:

01: F7 E1	mul ecx	;EDX:EAX = EAX * ECX
02: F76604	mul dword ptr [esi+4] ; EDX:EAX = EAX * dword_at(ESI+4)	
03: F6 E1	mul cl	;AX = AL * CL
04: 66 F7 E2	Dx múltiple	;DX:AX = AX * DX

Consideremos algunos otros ejemplos concretos:

01: B8 03 00 00 00 movimiento eax,3	; establecer EAX=3
02: B9 22 22 22 22 movimiento ecx,22222222h ; establecer ECX=0x22222222	
03: F7 E1	mul ecx ;EDX:EAX = 3 * 0x22222222 =
	; 0x66666666
	; por lo tanto, EDX=0, EAX=0x66666666
04: B8 03 00 00 00 movimiento eax,3	; establecer EAX=3
05: B9 00 00 00 80 movimiento ecx,80000000h; establecer ECX=0x80000000	
06: F7 E1	mul ecx ;EDX:EAX = 3 * 0x80000000 =
	; 0x180000000
	; por lo tanto, EDX=1, EAX=0x80000000

La razón por la que el resultado se almacena en EDX:EAX para la multiplicación de 32 bits es porque el resultado potencialmente puede no caber en un registro de 32 bits (como se demuestra en las líneas 4 a 6).

IMUL tiene tres formas:

- IMUL reg/mem : igual que MUL
- IMUL reg1, reg2/mem — reg1 = reg1 * reg2/mem
- IMUL reg1, reg2/mem, imm — reg1 = reg2 * imm

Algunos desensambladores acortan los parámetros. Por ejemplo:

01: F7 E9	imul ecx	;EDX:EAX = EAX * ECX
02: 69 F6 A0 01 00+ imul esi, 1A0h ; ESI = ESI * 0x1A0		

03:0F de la UE

imul ecx, esi ; ECX = ECX * ESI

La división con signo y sin signo se realiza mediante las instrucciones DIV e IDIV , respectivamente. Estas instrucciones solo toman un parámetro (divisor) y tienen la siguiente forma: DIV/IDIV reg/mem. Según el tamaño del divisor, DIV utilizará AX, DX:AX o EDX:EAX como dividendo, y el par cociente/resto resultante se almacena en AL/AH, AX/DX o EAX/EDX. Por ejemplo:

01: F7 F1	div.ecx	; EDX:EAX / ECX, cociente en EAX,
02: F6 F1	clase div	; AX / CL, cociente en AL, resto en AH
03:F77624	div dword ptr [esi+24h] ; ver línea 1	
04:B1 02	movimiento cl,2	; establecer CL = 2
05: B8 0A 00 00 00	movimiento eax,0Ah; establecer EAX = 0xA	
06: F6 F1	clase div	; AX/CL = A/2 = 5 en AL (cociente), ; AH = 0 (resto)
07:B1 02	movimiento cl,2	; establecer CL = 2
08: B8 09 00 00 00	mov eax,09h ; establecer EAX = 0x9	
09: F6 F1	clase div	; AX/CL = 9/2 = 4 en AL (cociente), ; AH = 1 (resto)

Operaciones de pila e invocación de funciones

La pila es una estructura de datos fundamental en los lenguajes de programación y los sistemas operativos. Por ejemplo, las variables locales en C se almacenan en el espacio de pila de las funciones. Cuando el sistema operativo pasa del anillo 3 al anillo 0, guarda la información de estado en la pila. Conceptualmente, una pila es una estructura de datos de tipo "último en entrar, primero en salir" que admite dos operaciones: push y pop. Push significa poner algo en la parte superior de la pila; pop significa quitar un elemento de la parte superior. Hablando en términos concretos, en x86, una pila es una región de memoria contigua a la que apunta ESP y que crece hacia abajo. Las operaciones push/pop se realizan a través de las instrucciones PUSH/POP y modifican implícitamente ESP. La instrucción PUSH decremente y luego escribe los datos en la ubicación indicada por ESP; POP lee los datos e incrementa ESP. El valor predeterminado de incremento/decremento automático es 4, pero se puede cambiar a 1 o 2 con una anulación del prefijo. En la práctica, el valor es casi siempre 4 porque el sistema operativo requiere que la pila esté alineada con dos palabras.

Supongamos que ESP apunta inicialmente a 0xb20000 y tiene el siguiente código:

; ESP inicial = 0xb20000		
01: B8 AA AA AA AA	movimiento	eax,0AAAAAAAAAh
02: BB BB BB BB BB	movimiento	ebx,0BBBBBBBBBh
03: B9 CC CC CC CC	movimiento	ecx,0CCCCCh
04: BA DD DD DD DD	movimiento	edx,0DDDDDDDDh
05: 50	Empujar EAX	
; la dirección 0xb1fffc contendrá el valor 0xAAAAAAA y ESP; será 0xb1ffc (=0xb20000-4)		

14 Capítulo 1 ■ x86 y x64

```

06:53           Empujar ebx
: la dirección 0xb1fff8 contendrá el valor 0BBBBBBBB y ESP
: será 0xb1fffc (=0xb1ffc-4)
07:5E           estallido      esi
: ESI contendrá el valor 0BBBBBBBB y ESP será 0xb1fffc
: (=0xb1fff8+4)
08:5F           estallido      editar
: EDI contendrá el valor 0AAAAAAA y ESP será 0xb20000
: (=0xb1ffc+4)

```

La figura 1-3 ilustra el diseño de la pila.



Figura 1-3

ESP también puede modificarse directamente mediante otras instrucciones, como ADD y SUB.

Si bien los lenguajes de programación de alto nivel tienen el concepto de funciones que se pueden llamar y de las que se puede regresar, el procesador no proporciona tal abstracción. En el nivel más bajo, el procesador opera solo sobre objetos concretos, como registros o datos que provienen de la memoria. ¿Cómo se traducen las funciones a nivel de máquina? ¡Se implementan a través de la estructura de datos de la pila!

Considere la siguiente función:

```

do
    entero
    __cdecl addme(a corta, b corta)
    {
        devuelve a+b;
    }

Asamblea
01:004113A0 55          Empujar ebp

```

02:004113A1 8B CE	movimiento	ebp, esp
03: ...		
04: 004113BE 0F BF 45 08 movsx eax, palabra ptr [ebp+8]		
05: 004113C2 0F BF 4D 0C movsx ecx, palabra ptr [ebp+0Ch]		
06:004113C6 03 C1	añadir eax, ecx	
07: ...		
08:004113CB8BE5	movimiento	esp, ebp
09:004113CD5D	estallido	-----
10:004113CE C3	Retirada	

La función se invoca con el siguiente código:

do		
suma = sumame(x, y);		
Asamblea		
01:004129F3 50	Empujar EAX	
02: ...		
03:004129F8 51	Empujar ECX	
04:004129F9 E8 F1 E7 FF FF llamar addme		
05:004129FE83C408	añadir esp, 8	

Antes de entrar en detalles, primero considere las instrucciones CALL/RET y Convenciones de llamada. La instrucción CALL realiza dos operaciones:

1. Introduce la dirección de retorno (dirección inmediatamente después de la instrucción CALL).
2. Cambia el EIP al destino de la llamada. Esto transfiere efectivamente el control a el objetivo de la llamada y comienza la ejecución allí.

RET simplemente introduce la dirección almacenada en la parte superior de la pila en EIP y le transfiere el control (literalmente, como un “POP EIP” , pero esa secuencia de instrucciones no existe en x86). Por ejemplo, si desea comenzar la ejecución en 0x12345678, puede hacer lo siguiente:

01:6878563412 empujar 0x12345678		
02: C3	retirado	

Una convención de llamada es un conjunto de reglas que dictan cómo funcionan las llamadas de función a nivel de máquina. Está definida por la Interfaz binaria de aplicación (ABI) para un sistema en particular. Por ejemplo, ¿los parámetros deben pasarse a través de la pila, en registros o en ambos? ¿Los parámetros deben pasarse de izquierda a derecha o de derecha a izquierda? ¿El valor de retorno debe almacenarse en la pila, en registros o en ambos?

Existen muchas convenciones de llamada, pero las más populares son CDECL, STDCALL, THISCALL y FASTCALL. (El compilador también puede generar su propia convención de llamada personalizada, pero no se tratarán aquí). La Tabla 1-2 resume su semántica.

16 Capítulo 1 ■ x86 y x64

Tabla 1-2: Convenciones de llamadas

	CDECL	LLAMADA ESTÁNDAR	LLAMADA RÁPIDA
Parámetros	Empujado en el pila de derecha a izquierda. El llamador debe Limpia la pila después de la llamada.	Igual que CDECL excepto que el llamado debe limpiar La pila.	Los dos primeros parámetros se pasan en ECX y EDX. El resto se almacena en la pila.
Valor de retorno	Almacenado en EAX.	Almacenado en EAX.	Almacenado en EAX.
Registros no volátiles	EBP, ESP, EBX, EDI.	EBP, ESP, EBX, EDI.	Sistemas de gestión de documentos electrónicos (SEC).

Ahora volvemos al fragmento de código para analizar cómo se invoca la función addme . En las líneas 1 y 3, los dos parámetros se insertan en la pila; ECX y EAX son el primer y el segundo parámetro, respectivamente. La línea 4 invoca la función addme . Función con la instrucción CALL . Esto coloca inmediatamente la dirección de retorno, 0x4129FE, en la pila y comienza la ejecución en 0x4113A0. La Figura 1-4 ilustra la disposición de la pila después de ejecutar la línea 4.

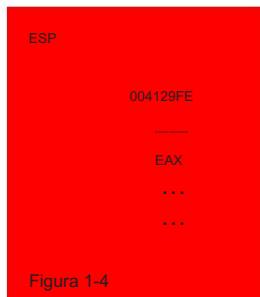


Figura 1-4

Después de que se ejecuta la línea 4, ahora estamos en el cuerpo de la función addme . La línea 1 inserta EBP en la pila. La línea 2 establece EBP en el puntero de pila actual. Esta secuencia de dos instrucciones se conoce normalmente como el prólogo de la función porque establece un nuevo marco de función. La línea 4 lee el valor en la dirección EBP+8, que es el primer parámetro en la pila; la línea 5 lee el segundo parámetro. Tenga en cuenta que se accede a los parámetros utilizando EBP como registro base. Cuando se utiliza en este contexto, EBP se conoce como el puntero del marco base (consulte la línea 2) porque apunta al marco de pila para la función actual, y se puede acceder a los parámetros/variables locales en relación con él. También se puede ordenar al compilador que genere código que no utilice EBP como puntero de marco base a través de una optimización llamada omisión de puntero de marco.

Con esta optimización, el acceso a las variables y parámetros locales se realiza en relación con ESP, y EBP se puede utilizar como un registro general como EAX, EBX, ECX, etc. La línea 6 suma los números y guarda el resultado en EAX. La línea 8 establece el puntero de pila en el puntero del marco base. La línea 9 extrae el EBP guardado de la línea 1 en

EBP. Esta secuencia de dos instrucciones se conoce comúnmente como el epílogo de la función porque se encuentra al final de la función y restaura el marco de la función anterior. En este punto, la parte superior de la pila contiene la dirección de retorno guardada por la instrucción CALL en 0x4129F9. La línea 10 realiza un RET, que vacía la pila y reanuda la ejecución en 0x4129FE. La línea 5 del fragmento reduce la pila en 8 porque el llamador debe limpiar la pila según la convención de llamada de CDECL .

Si la función addme tuviera variables locales, el código necesitaría hacer crecer la pila restando ESP después de la línea 2. Todas las variables locales serían entonces accesibles a través de un desplazamiento negativo desde EBP.

Ceremonias

1. Teniendo en cuenta lo que aprendiste sobre CALL y RET, explica cómo lees el valor de EIP. ¿Por qué no puedes simplemente hacer MOV EAX, EIP?
2. Piense en al menos dos secuencias de código para establecer EIP en 0xAABBCCDD.
3. En la función de ejemplo, addme, ¿qué sucedería si el puntero de la pila ¿No se restauraron correctamente antes de ejecutar RET?
4. En todas las convenciones de llamada explicadas, el valor de retorno se almacena en un registro de 32 bits (EAX). ¿Qué sucede cuando el valor de retorno no cabe en un registro de 32 bits? Escriba un programa para experimentar y evaluar su respuesta. ¿El mecanismo cambia de un compilador a otro?

Flujo de control

Esta sección describe cómo el sistema implementa la ejecución condicional para construcciones de nivel superior como if/else, switch/case y while/for. Todas ellas se implementan a través de las instrucciones CMP, TEST, JMP y Jcc y el registro EFLAGS . La siguiente lista resume los indicadores comunes en EFLAGS:

- Indicador ZF/Cero : se establece si el resultado de la operación aritmética anterior es cero.
- SF/Bandera de signo : se establece en el bit más significativo del resultado.
- Indicador CF/Carry : se establece cuando el resultado requiere un carry. Se aplica a los datos sin signo. números.
- Indicador OF/Desbordamiento : se establece si el resultado supera el tamaño máximo. Se aplica a números con signo.

Las instrucciones aritméticas actualizan estos indicadores en función del resultado. Por ejemplo, la instrucción SUB EAX, EAX haría que se estableciera ZF . Las instrucciones Jcc , donde “cc” es un código condicional, cambian el flujo de control en función de estos indicadores.

banderas. Puede haber hasta 16 códigos condicionales, pero los más comunes se describen en la Tabla 1-3.

Tabla 1-3: Códigos condicionales comunes

CONDICIONAL CÓDIGO	DESCRIPCION EN INGLÉS	MÁQUINA DESCRIPCIÓN
B/NAE	Ni por encima ni por debajo de/ni igual. Se utiliza para operaciones sin signo.	FC=1
Nota:	Ni inferior ni superior ni igual. Se utiliza para operaciones sin signo.	CF=0
Mi/Z	Igual/Cero	ZF=1
NE/Nueva Zelanda	No es igual/no es cero	ZF=0
yo	Menor que/Ni mayor ni igual. Se utiliza para operaciones con signo.	(SF ^ DE) = 1
GE/NL	Mayor o igual/No menor que. Se utiliza para operaciones con signo.	(SF ^ DE) = 0
G/NLE	Mayor/Ni menor ni igual. Se utiliza para operaciones con signo.	((SF ^ DE) ZF) = 0

Debido a que el lenguaje ensamblador no tiene un sistema de tipos definido, una de las pocas formas de reconocer tipos con signo/sin signo es a través de estos códigos condicionales.

La instrucción CMP compara dos operandos y establece el código condicional apropiado en EFLAGS; compara dos números restando uno del otro sin actualizar el resultado. La instrucción TEST hace lo mismo, excepto que realiza un AND lógico entre los dos operandos.

Si-Si no

Las construcciones if-else son bastante simples de reconocer porque implican una comparación/Prueba seguida de Jcc. Por ejemplo:

Asamblea

```

01: movimiento esi, [ebp+8]
02: edición especial de movimiento, [esi]
03: prueba edx, edx
04:jz          loc_4E31F9 corto
05: movimiento ecx, desplazamiento _FsRtlFastMutexLookasideList
06: llamar a _ExFreeToNPagedLookasideList@8
07: y dword ptr [esi], 0
08: lea eax, [esi+4]
09: empujar eax
10: llamar a _FsRtlUninitializeBaseMcb@4
11: loc_4E31F9:

```

```

12: pop esi 13: pop ebp 14:
retn 4

15: _FsRtlUninitializeLargeMcb@4 fin

Pseudo C

si (*esi == 0) { devolver;

}

ExFreeToNPagedLookasideList(...);

*esi = 0;
...
devolver;

o

si (*esi != 0) {
...
ExFreeToNPagedLookasideList(...);
*esi = 0;
...
}

} devolver;

```

La línea 2 lee un valor en la ubicación ESI y lo almacena en EDX. La línea 3 realiza la operación AND de EDX consigo misma y establece los indicadores apropiados en EFLAGS. Tenga en cuenta que este patrón se utiliza comúnmente para determinar si un registro es cero. La línea 4 salta a loc_4E31F9 (línea 12) si ZF=1. Si ZF=0, entonces ejecuta la línea 5 y continúa hasta que la función retorna.

Tenga en cuenta que hay dos traducciones de C ligeramente diferentes pero lógicamente equivalentes para este fragmento.

Caja del interruptor

Un bloque switch-case es una secuencia de instrucciones if/else. Por ejemplo:

```

Caja del interruptor

cambiar(ch) {
    caso 'c':
        manejar_C();
        romper;
    caso 'h':
        handle_H(); romper;

    por defecto:
        romper;

} hacer más();
...
```

20 Capítulo 1 ■ x86 y x64

```

Si-Si no

    si (ch == 'c') {
        manejar_C();
    } demás
    si (ch == 'h') {
        manejar_H();
    }
    hacer más();
    ...

```

Por lo tanto, la traducción del código de máquina será una serie if/else. Un ejemplo sencillo ilustra la idea:

Asamblea	01: empuje ebp 02: movimiento ebp, esp 03: movimiento Ej., [ebp+8] 04: sub eax, 41h 05:jz loc_caseA corto 06: dic fácil 07:jz loc_caseB corto 08: dic fácil 09:jz loc_caseC corto 10: movimiento al, 5Ah 11: movzx eax, al 12: estallido ... 13: retornar 14: loc_caseC; 15: mov al, 43h 16: movzx eax, al 17: estallido ... 18: retornado 19: loc_caseB; 20: movimiento al, 42h 21: movzx eax, al 22: estallido ... 23: retornado 24: caso_ubicaciónA: 25: movimiento al, 41h 26: movzx eax, al 27: estallido ... 28: retornado
----------	---

```

do

    carácter sin signo switchme(int a)
    {
        char res sin signo;

```

```
cambiar(a) {
    caso 0x41:
        res = 'A';
        romper;
    caso 0x42:
        res = 'B';
        romper;
    caso 0x43:
        res = 'C';
        romper;
    por defecto:
        res = 'Z';
        romper;
    }
    devolver res;
}
```

Las declaraciones de cambio de caso de la vida real pueden ser más complejas y los compiladores comúnmente construyen una tabla de saltos para reducir la cantidad de comparaciones y saltos condicionales. La tabla de saltos es esencialmente una matriz de direcciones, cada una de las cuales apunta al controlador de un caso específico. Este patrón se puede observar en la muestra J en sub_10001110:

Asamblea

```
01: cmp     edición, 5
02: si      loc_10001141 corta
03: salto   ds:off_100011A4[edi*4]
04: loc_10001125; 05: movimiento
            esi, 40h
06: salto   loc_10001145 corto
07: loc_1000112C; 08: movimiento
            esi, 20h
09: salto   loc_10001145 corto
10: loc_10001130; 11: movimiento
            esi, 38h
12: salto   loc_10001145 corto
13: loc_1000113A; 14: movimiento
            esi, 30h
15: salto   loc_10001145 corto
16: loc_10001141; 17: movimiento
            esi, [esp+0Ch]
18: ...
19: off_100011A4 desplazamiento dd loc_10001125
20: desplazamiento dd loc_10001125 21:
desplazamiento dd loc_1000113A
22: desplazamiento de dd loc_1000112C
23: desplazamiento de dd loc_10001133
24: desplazamiento de dd loc_1000113A
```

22 Capítulo 1 ■ x86 y x64

Pseudo C

```
cambiar(edi) {
    caso 0:
        Caso 1:
            // ir a loc_10001125; esi = 0x40;

            romper;
        caso 2:
        caso 5:
            // ir a loc_1000113A; esi = 0x30;

            romper;
        caso 3:
            // ir a loc_1000112C; esi = 0x20;

            romper;
        caso 4:
            // ir a loc_10001133; esi = 0x38;

            romper;
    por defecto:
        // ir a loc_10001141; esi =
        *(esp+0xC)
        romper;
}
```

...

Aquí, el compilador sabe que solo hay cinco casos y que el valor del caso es consecutivo; por lo tanto, puede construir la tabla de saltos e indexarla directamente (línea 3). Sin la tabla de saltos, habría 10 instrucciones adicionales para probar cada caso y ramificar al controlador. (Existen otras formas de optimización de switch/case, pero no las cubriremos aquí).

Bucles

A nivel de máquina, los bucles se implementan utilizando una combinación de instrucciones Jcc y JMP . En otras palabras, se implementan utilizando las construcciones if/else y goto . La mejor manera de entender esto es reescribir un bucle utilizando solo if/else y goto. Considere el siguiente ejemplo: Uso de for

```
para (int i=0; i<10; i++) { printf("%d\n", i);

} printf("hecho!\n");
```

Uso de if/else y goto

```

int i = 0;
inicio_de_bucle:
    si (i < 10) {
        printf("%d\n", i);
        yo++;
        ir a loop_start;
    }
printf("hecho\n");

```

Una vez compiladas, ambas versiones son idénticas a nivel de código máquina:

01:00401002	movimiento	edición, ds:_imp__printf
02:00401008	xor	Sí, sí
03:0040100A	pista	ebx, [ebx+0]
04: 00401010	loc_401010. 05:	
00401010	empujar esi	
06: 00401011	desplazamiento de empuje Formato	; "%d\n"
07:00401016	llamar a edi: __imp__printf	
08:00401018	incluido	esi
09:00401019	añadir	esp, 8
10:0040101C	cmp	esi, 0Ah
11:0040101F	jeje	loc_401010 corto
12: 00401021	desplazamiento de inserción realizado	; "¡Hecho!\n"
13:00401026	llamar edi : __imp__printf	
14:00401028	añadir	esp, 4

La línea 1 establece EDI en la función printf . La línea 2 establece ESI en 0. La línea 4 comienza el bucle; sin embargo, tenga en cuenta que no comienza con una comparación. No hay comparación aquí porque el compilador sabe que el contador se inicializó en 0 (consulte la línea 2) y, obviamente, será menor que 10, por lo que omite la verificación. Las líneas 5 a 7 llaman a la función printf con los parámetros correctos (especificador de formato y nuestro número). La línea 8 incrementa el número. La línea 9 limpia la pila porque printf usa la convención de llamada CDECL . La línea 10 verifica si el contador es menor que 0xA. Si lo es, vuelve a loc_401010. Si el contador no es menor que 0xA, continúa la ejecución en la línea 12 y finaliza con un printf.

Una observación importante que se debe hacer es que el desensamblaje nos permitió inferir que el contador es un entero con signo. La línea 11 utiliza el código condicional “menor que” (JL), por lo que sabemos inmediatamente que la comparación se realizó con enteros con signo. Recuerde: si es “arriba/abajo”, no tiene signo; si es “menor que/mayor que”, tiene signo. La muestra L tiene una pequeña función, sub_1000AE3B, con el siguiente bucle interesante:

Asamblea

```

01: sub_1000AE3B proc cerca
02: empajar           editar

```

24 Capítulo 1 ■ x86 y x64

```

03: empujar 04:      esi
llamar          ds:IstrlenA
05: movimiento     edi, eax
06: xor            ecx, ecx
07: xor            educación, educación
08: prueba         edito, edito
09:jle           loc_1000AE5B corto
10:loc_1000AE4D:
11: movimiento al, [edx+esi]
12: movimiento     [ecx+esi], al
13: añadir         edición, 3
14: aum            Cálculo Único
15: cmp 16: jl    edx, edición
17:               loc_1000AE4D corto
loc_1000AE5B:
18: movimiento     byte ptr [ecx+esi], 0
19: movimiento     EAX, ESI
20: pop 21: retn   editar

22: sub_1000AE3B fin

```

```

do
    carácter *sub_1000AE3B (carácter *cadena)
{
    int len, i=0, j=0;
    len = IstrlenA(str);
    si (len <= 0) {
        cadena[] = 0;
        devuelve str;
    }
    mientras (j < len) {
        cadena[i] = cadena[j];
        j = j+3;
        yo = yo+1;
    }
    cadena[i] = 0;
    devuelve str;
}

```

La función sub_1000AE3B tiene un parámetro que se pasa usando una convención de llamada personalizada (ESI contiene el parámetro). La línea 2 guarda EDI. La línea 3 llama a IstrlenA. con el parámetro; por lo tanto, inmediatamente sabes que ESI es de tipo char *. La línea 5 guarda el valor de retorno (longitud de la cadena) en EDI. Las líneas 6 y 7 borran ECX y EDX. Las líneas 8 y 9 comprueban si la longitud de la cadena es menor o igual a cero. Si es así, el control se transfiere a la línea 18, que establece el valor en ECX+ESI en 0. Si no lo es , la ejecución continúa en la línea 11, que es el inicio de un bucle. Primero, lee el carácter en ESI+EDX (línea 11) y luego lo almacena en ESI+ECX (línea 12).

A continuación, incrementa EDX y ECX en tres y uno, respectivamente. Las líneas 15 y 16 comprueban si EDX es menor que la longitud de la cadena; si es así, la ejecución vuelve al inicio del bucle. Si no, la ejecución continúa en la línea 18.

Puede parecer complicado al principio, pero esta función toma una cadena ofuscada. cuyo valor desofuscado es cada tercer carácter. Por ejemplo, la cadena SX] OTYFKPTY\W\laAFKRW\NE es en realidad SOFTWARE. El propósito de esta función es evitar que los escáneres de cadenas sean ingenuos y evadir la detección. Como ejercicio, debería descompilar esta función para que parezca más "natural" (en contraposición a nuestra traducción literal).

Además de las construcciones Jcc normales, se pueden implementar ciertos bucles utilizando La instrucción LOOP ejecuta un bloque de código hasta ECX . tiempo. Por ejemplo:

```

Asamblea
01:8B CA           mov    eax, [edi+0]
02:                 mov    edx, [edi+4]
03: d.C.            mov    [edi], al
04: F7 D0           not    eax, eax
05: AB              mov    [edi], al
06: E2 FA           loop   loc_ _CFBB8F

C áspero
mientras (ecx != 0) {
    eax = *esi;
    esi++;
    edi = ~eax;
    edi++;
    ecx--;
}

```

La línea 1 lee el contador de EDX. La línea 3 es el inicio del bucle; lee una palabra doble en la dirección de memoria ESI y la guarda en EAX; también incrementa EDI. por 4. La línea 4 ejecuta el operador NOT en el valor que se acaba de leer. La línea 5 escribe el valor modificado a la dirección de memoria EDI e incrementa ESI en 4. Línea 6 comprueba si ECX es 0; si no, la ejecución continúa desde el inicio del bucle.

Mecanismo del sistema

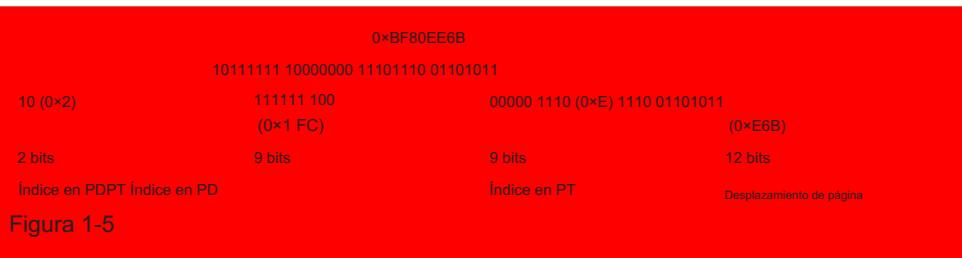
Las secciones anteriores explican los mecanismos e instrucciones que están disponibles para el código que se ejecuta en todos los niveles de privilegio. Para comprender mejor la arquitectura, en esta sección se analizan dos mecanismos fundamentales a nivel de sistema: virtual Traducción de direcciones y manejo de excepciones e interrupciones. Puede omitir esta sección en Una primera lectura.

Traducción de direcciones

La memoria física de un sistema informático se divide en unidades de 4 KB llamadas páginas. (Una página puede tener más de 4 KB, pero no analizaremos los otros tamaños aquí). Las direcciones de memoria se dividen en dos categorías: virtuales y físicas. Las direcciones virtuales son las que utilizan las instrucciones que se ejecutan en el procesador cuando la paginación está habilitada. Por ejemplo:

```
01: A1 78 56 34 12 mov eax, [0x12345678]; leer memoria en el virtual
01.89 08           movimiento [eax], ecx      ; dirección 0x12345678
                                         ; escribe ECX en el virtual
                                         ; dirección EAX
```

Las direcciones físicas son las ubicaciones de memoria reales que utiliza el procesador al acceder a la memoria. La unidad de administración de memoria (MMU) del procesador traduce de manera transparente cada dirección virtual en una dirección física antes de acceder a ella. Si bien una dirección virtual puede parecer simplemente otro número para el usuario, existe una estructura para ella cuando la ve la MMU. En los sistemas x86 con soporte para extensión de dirección física (PAE), una dirección de memoria virtual se puede dividir en índices en tres tablas y desplazamiento: tabla de puntero de directorio de página (PDPT), directorio de página (PD), tabla de página (PT) y entrada de tabla de página (PTE). Una PDPT es una matriz de cuatro elementos de 8 bytes, cada uno de los cuales apunta a un PD. Una PD es una matriz de 512 elementos de 8 bytes, cada uno de los cuales apunta a un PT. Una PT es una matriz de 512 elementos de 8 bytes, cada uno de los cuales contiene un PTE. Por ejemplo, la dirección virtual 0xBF80EE6B se puede entender como se muestra en la Figura 1-5.



Los elementos de 8 bytes de estas tablas contienen datos sobre las tablas, los permisos de memoria y otras características de la memoria. Por ejemplo, hay bits que determinan si la página es de solo lectura o legible/escribible, ejecutable o no ejecutable, accesible para el usuario o no, etc.

El proceso de traducción de direcciones gira en torno a estas tres tablas y al registro CR3 . CR3 contiene la dirección base física del PDPT. El resto de esta sección explica la traducción de la dirección virtual 0xBF80EE6B en un sistema real (consulte la Figura 1-5):

```
kd>r @cr3          ; CR3 es la dirección física de la base de un PDPT
cr3=085c01e0
kd>ldq @cr3+2*8 L1 ; lee la entrada PDPT en el índice 2
#85c01f0 00000000'0d66e001
```

Según la documentación, los 12 bits inferiores de una entrada PDPT son indicadores/ bits reservados, y los restantes se utilizan como dirección física de la base PD. El bit 63 es el indicador NX en PAE, por lo que también deberá borrarlo. En este ejemplo en particular, no lo borramos porque ya es 0. (Estamos viendo páginas de códigos que son ejecutables).

```
; 0x00000000`0d66e001 = 00001101 01100110 11100000 00000001
; después de limpiar los 12 bits inferiores, tenemos
; 0x0d66e000          = 00001101 01100110 11100000 00000000
; Esto nos indica que la base PD está en la dirección física 0x0d66e000
kd> !dq 0d66e000+0x1fc*8 L1 ; lee la entrada PD en el índice 0x1FC
#d66efe0 00000000`0964b063
```

Nuevamente, según la documentación, los 12 bits inferiores de una entrada PD se utilizan para banderas/bits reservados, y los restantes se utilizan como base para el PT:

```
; 0x0964b063 = 00001001 01100100 10110000 01100011
; después de limpiar los 12 bits inferiores, obtenemos
; 0x0964b000 = 00001001 01100100 10110000 00000000
; Esto nos indica que la base PT está en 0x0964b000
kd> !dq 0964b000+e*8 L1 # 964b070           ; lee la entrada PT en el índice 0xE
00000000`06694021
```

Nuevamente, los 12 bits inferiores se pueden borrar para llegar a la base de una entrada de página:

```
; 0x06694021 = 00000110 01101001 01000000 00100001
; después de limpiar los 12 bits inferiores, obtenemos
; 0x06694000 = 00000110 01101001 01000000 00000000
; Esto nos indica que la base de la entrada de la página está en 0x06694000
kd>!db 06694000+e6b L8                   ; lee 8 bytes de la entrada de la página en el desplazamiento
0xE6B
# 6694e6b 8b y siguientes 55 8b ec 83 ec 0c ..U.....t.... ; nuestros datos en ese momento
; página física
kd>!dbbf80ee6bL8                         ; lee 8 bytes de la dirección virtual
bf80ee6b 8b ff 55 8b ec 83 ec ..U.....t.... ; mismos datos!
```

Después de todo el proceso, se determina que la dirección virtual 0xBF80EE6B se traduce a la dirección física 0x6694E6B.

Los sistemas operativos modernos implementan la separación del espacio de direcciones de los procesos mediante este mecanismo. Cada proceso está asociado con un CR3 diferente, lo que da como resultado una traducción de dirección virtual específica del proceso. Es la magia detrás de la ilusión de que cada proceso tiene su propio espacio de direcciones. ¡Espero que la próxima vez que su programa acceda a la memoria, tenga más en cuenta el procesador!

Interrupciones y excepciones

Esta sección analiza brevemente las interrupciones y excepciones, ya que los detalles completos de implementación se pueden encontrar en el Capítulo 3, "El núcleo de Windows".

En los sistemas informáticos contemporáneos, el procesador normalmente está conectado a los dispositivos periféricos a través de un bus de datos como PCI Express, FireWire o USB.

Cuando un dispositivo requiere la atención del procesador, provoca una interrupción que obliga al procesador a pausar lo que esté haciendo y a procesar la solicitud del dispositivo. ¿Cómo sabe el procesador cómo manejar la solicitud? En el nivel más alto, se puede pensar en una interrupción como algo asociado a un número que luego se utiliza para indexar en una matriz de punteros de función. Cuando el procesador recibe la interrupción, ejecuta la función en el índice asociado con la interrupción y reanuda la ejecución donde estaba antes de que ocurriera la interrupción. Estas se denominan interrupciones de hardware porque son generadas por dispositivos de hardware.

Son asincrónicos por naturaleza.

Cuando el procesador está ejecutando una instrucción, puede encontrarse con excepciones. Por ejemplo, una instrucción podría generar un error de división por cero, hacer referencia a una dirección no válida o activar una transición de nivel de privilegio. Para los fines de esta discusión, las excepciones se pueden clasificar en dos categorías: fallas y trampas. Una falla es una excepción corregible. Por ejemplo, cuando el procesador ejecuta una instrucción que hace referencia a una dirección de memoria válida pero los datos no están presentes en la memoria principal (fueron paginados), se genera una excepción de falla de página. El procesador maneja esto guardando el estado de ejecución actual, llamando al manejador de falla de página para corregir esta excepción (paginando los datos) y volviendo a ejecutar la misma instrucción (que ya no debería causar una falla de página). Una trampa es una excepción causada por la ejecución de tipos especiales de instrucciones. Por ejemplo, la instrucción SYSENTER hace que el procesador comience a ejecutar el manejador de llamada de sistema genérico ; después de que el manejador termina, la ejecución se reanuda en la instrucción inmediatamente después de SYSENTER. Por lo tanto, la principal diferencia entre una falla y una trampa es dónde se reanuda la ejecución. Los sistemas operativos comúnmente implementan llamadas de sistema a través del mecanismo de interrupción y excepción.

Tutorial paso a paso

Terminamos el capítulo con un recorrido por una función con menos de 100 instrucciones. Se trata de la rutina DllMain de Sample J. Este ejercicio tiene dos objetivos.

En primer lugar, aplica casi todos los conceptos tratados en el capítulo (excepto el caso switch).

En segundo lugar, enseña un requisito importante en la práctica de la ingeniería inversa: leer manuales técnicos y documentación en línea. Esta es la función:

```

01:      ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason,
; LPVOID lpvReservado)
02:          _DllMain@12 proc cerca de enviar
03:55          ebp
04:8B CE      movimiento    ebp, esp
05:81 EC 30 01 00+ subtítulos    esp, 130h
06:57          empujar     editar
07:0F 01 4D F8      al lado    Palabra clave [ebp-8]
08:8B 45FA      movimiento    eax, [ebp-6]
09:3D 00 F4 03 80 cm    EAX, 8003F400h

```

10:7610	yo soy	loc_10001C88 corto (línea 18)
11:3D 00 74 04 80 cmp 12:73 09 jnb		eje, 80047400h
		loc_10001C88 corto (línea 18)
13:33 C0	xor	Ea, Ea
14:5F	estallido	editar
15:88 E5	movimiento	esp, ebp
16:5D	estallido
17: C2 0C 00	Retirada	0Ch
18:	ubicación_10001C88:	
19:33 C0	xor	Ea, Ea
20:B9 49 00 00 00 movimiento		ecx, 49 h
21:8D BD D4 FE FF+ lea		editado, [ebp-12Ch]
22: C7 85 D0 FE FF+ movimiento		palabra clave dword [ebp-130h], 0
23:50	Empuja,	fácil
24:6A 02	empuja	2
25: F3 AB	representante	slosd
26: E8 2D 2F 00 00 llamada		CrearHerramientaAyuda32Instantánea
27:88 F8	movimiento	edi, eax
28:83 FFFF		editado, 0FFFFFFFh
29:75 09	cmpjnz-es	loc_10001CB9 corto (línea 35)
30:33 C0	xor	Ea, Ea
31:5F	estallido	editar
32:88 E5	movimiento	esp, ebp
33:5D	estallido
34: C2 0C 00	Retirada	0Ch
35:	loc_10001CB9:	
36: 8D 85 D0 FE FF+ lea		EAX, [ebp-130h]
37:56	Empuja,	esi
38:50	empuja	fácil
39:57	empujar	editar
40: C7 85 D0 FE FF+ movimiento		palabra clave dword [ebp-130h], 128h
41:E8 FF 2E 00 00 llamada		Proceso32Primero
42:85 C0	prueba	Ea, Ea
43:74 4F	yo	loc_10001D24 corto (línea 70)
44:8B 35 C0 50 00+ movimiento		esi, ds:_strcmp
45: 8D 8D F4 FE FF+ lea		ecx, [ebp-10Ch]
46:68 50 7C 00 10 empuje		10007C50h
47:51	Llamada	Locales_Globales
48: FF D6	push	esi : _strcmp
49:83 C4 08	agregar	esp, 8
50:85 C0	prueba	Ea, Ea
51:7426	jz short loc_10001D16 (línea 66)	
52:	loc_10001CF0:	
53: 8D 95 D0 FE FF+ lea		edx, [ebp-130h]
54:52	empujar	edicion
55:57	empujar	editar
56:E8 CD 2E 00 00 llamada		Proceso32Siguiente
57:85 C0	prueba	Ea, Ea
58:7423	yo	loc_10001D24 corto (línea 70)
59: 8D 85 F4 FE FF+ lea		eax, [ebp-10Ch]
60:68 50 7C 00 10 empuje		10007C50h
61:50	Llamada	fácil
62: FF D6	push	esi : _strcmp
63:83 C4 08	agregar	esp, 8

30 Capítulo 1 ■ x86 y x64

64:85 C0	prueba	Ea, Ea
65:75 D1A	jnz corto loc_10001CF0 (línea 52)	
66:	ubicación_10001D16:	
67: 8B 85 E8 FE FF+ movimiento	EAX, [ebp-118h]	
68: 8B 8D D8 FE FF+ movimiento	ecx, [ebp-128h]	
69:EB06	jmp short loc_10001D2A (línea 73)	
70:	loc_10001D24 eax,	
71:8B 45 0C	movimiento	[ebp+0Ch]
72:8B 4D 0C	movimiento	ecx, [ebp+0Ch]
73:	loc_10001D2A:	
74:3B C1	cmp	eax, ecx
75:5E	pop	esi
76:75 09	jnz	loc_10001D38 corto (línea 82)
77:33 C0	xor	Ea, Ea
78:5F	estallido	editar
79:8B E5	movimiento	esp, ebp
80:5D	estallido	-----
81: C2 0C 00	Retirada	0Ch
82:	ubicación_10001D38:	
83:8B450C	movimiento	eax, [ebp+0Ch]
84:48	dic	fácil
85:75 15	jnz	loc_10001D53 corto (línea 93)
86:6 a. m. 00	empujar	0
87:6 a. m. 00	Empuja,	0
88:6 a. m. 00	empuja	0
89:68 D0 32 00 10 empuje		100032D0h
90:6A 00 empuje		0
91:6 a. m. 00	empujar	0
92: FF 15 20 50 00+ llamada		ds:Crear hilo
93:	loc_10001D53 eje, 1	
94: B8 01 00 00 00 movimiento		
95: 5F	estallido	editar
96:8B E5	movimiento	esp, ebp
97:5D	estallido	-----
98: C2 0C 00	Retirada	0Ch
99:	_DlIMain@12 final	

Las líneas 3 y 4 configuran el prólogo de la función, que guarda el puntero del marco base anterior y establece uno nuevo. La línea 5 reserva 0x130 bytes de espacio de pila.

La línea 6 guarda EDI. La línea 7 ejecuta la instrucción SIDT , que escribe el registro IDT de 6 bytes en una región de memoria específica. La línea 8 lee una palabra doble en EBP-6.

y lo guarda en EAX. Las líneas 9 y 10 comprueban si EAX es menor o igual a 0x8003F400. Si es así , la ejecución se transfiere a la línea 18; de lo contrario, continúa la ejecución en la línea 11.

Las líneas 11 y 12 realizan una comprobación similar excepto que la condición no es inferior a 0x80047400.

Si es así, la ejecución se transfiere a la línea 18; de lo contrario, continúa ejecutándose en la línea 13. La línea 13 borra EAX. La línea 14 restaura el registro EDI guardado en la línea 6. Las líneas 15 y 16 restauran el marco base y el puntero de pila anteriores. La línea 17 agrega 0xC bytes al puntero de pila y luego regresa al llamador.

Antes de analizar el siguiente tema, cabe señalar algunas cosas sobre estas primeras 17 líneas. La instrucción SIDT (línea 7) escribe el contenido del registro IDT en un archivo de 6 bytes.

Ubicación de la memoria. ¿Qué es el registro IDT? El manual de referencia de Intel/AMD indica que IDT es una matriz de 256 entradas de 8 bytes, cada una de las cuales contiene un puntero a un controlador de interrupción, selector de segmento y desplazamiento. Cuando se produce una interrupción o excepción, el procesador utiliza el número de interrupción como índice en el IDT y llama al controlador especificado de la entrada. El registro IDT es un registro de 6 bytes; los 4 bytes superiores contienen la base de la matriz/tabla IDT y los 2 bytes inferiores almacenan el límite de la tabla. Con esto en mente, ahora sabe que la línea 8 en realidad está leyendo la dirección base de IDT. Las líneas 9 y 11 verifican si la dirección base está en el rango (0x8003F400, 0x80047400). ¿Qué tienen de especial estas constantes aparentemente aleatorias? Si busca en Internet, notará que 0x8003F400 es una dirección base de IDT en Windows XP en x86. Esto se puede verificar en el depurador del kernel:

```
0: kd> vertarget  
Windows XP Kernel Versión 2600 (Service Pack 3) MP (2 procedimientos) Compatibilidad x86 gratuita  
ible  
Creado por: 2600.xpsp.080413-2111  
...  
0: kd>r @idtr  
Identificación del autor=8003f400  
0: kd>~1  
1: kd>r @idtr  
Identificación = bab3c590
```

¿Por qué el código comprueba este comportamiento? Una posible explicación es que el desarrollador supuso que una dirección base IDT que se encuentra en ese rango se considera “inválida” o puede ser el resultado de una virtualización. La función devuelve automáticamente cero si la IDTR es “inválida”. Puede descompilar este código en C de la siguiente manera:

```
estructura de tipo definido _IDTR {  
    Base DWORD;  
    Límite CORTO;  
} IDTR, *PIDTR;  
BOOL __stdcall DllMain (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvRe-  
servido)  
{  
    IDTR idtr;  
    __sidt(&idtr);  
    si (idtr.base > 0x8003F400 && idtr.base < 0x80047400h) { devolver FALSO; }  
    //línea 18  
    ...  
}
```

NOTA Si lee el manual con atención, notará que cada procesador tiene su propio IDT y, por lo tanto, su IDTR. Por lo tanto, en un sistema de múltiples núcleos, el IDTR será diferente para cada núcleo. Claramente, 0x8003F400 es válido solo para el núcleo 0 en Windows XP. Si la instrucción se programara para ejecutarse en otro núcleo, el IDTR sería 0BAB3C590. En versiones posteriores de Windows, las direcciones base de IDT cambian entre reinicios; por lo tanto, la práctica de codificar direcciones base de forma rígida no funcionará.

32 Capítulo 1 ■ x86 y x64

Si la base IDT parece válida, el código continúa la ejecución en la línea 18. Las líneas 19 y 20 borran EAX y establecen ECX en 0x49. La línea 21 establece EDI en cualquier EBP-0x12C es; dado que EBP es el puntero del marco base, EBP-0x12C es la dirección de una variable local . La línea 22 escribe cero en la ubicación indicada por EBP-0x130. Las líneas 23 y 24 insertan EAX y 2 en la pila. La línea 25 pone a cero un búfer de 0x124 bytes a partir de EBP-0x12C. La línea 26 llama a CreateToolhelp32Snapshot:

```
MANEJAR WINAPI CreateToolhelp32Snapshot(
    _En_ DWORD dwFlags,
    _En_ DWORD th32ProcessID
);
```

Esta función de la API Win32 acepta dos parámetros enteros. Como regla general, las funciones de la API Win32 siguen la convención de llamada STDCALL . Por lo tanto, dwFlags y los parámetros th32ProcessId son 0x2 (línea 24) y 0x0 (línea 23). Esta función enumera todos los procesos del sistema y devuelve un identificador que se utilizará en Process32Next. Las líneas 27 y 28 guardan el valor de retorno en EDI y comprueban si es -1. Si lo es, el valor de retorno se establece en 0 y retorna (líneas 30 a 34); de lo contrario, la ejecución continúa en la línea 35. La línea 36 establece EAX en la dirección de la variable local previamente inicializada en 0 en la línea 22; la línea 40 la inicializa en 0x128. Las líneas 37 a 39 insertan ESI, EAX y EDI en la pila. La línea 41 llama a Process32First:

Prototipo de función

```
BOOL WINAPI Proceso32Primero(
    _En_ MANEJAR hSnapshot,
    _Entrada_ LPPROCESSENTRY32 lppe
);
```

Definición de estructura relevante

```
tipo de definición de estructura etiqueta PROCESSENTRY32 {
    Palabra D      tamañodw;
    Palabra D      cntUso;
    Palabra D      th32ID de proceso;
    ULONG_PTR th32ID predeterminado del montón;
    Palabra D      th32MóduloID;
    Palabra D      cntHilos;
    Palabra D      th32ParentProcessID;
    LARGO          pcPriClassBase;
    Palabra D      Banderas dw;
    TCHAR          szExeFile[RUTA_MÁXIMA];
} PROCESSENTRY32, *PPROCESSENTRY32;
```

```
00000000 PROCESSENTRY32 estructura; (tamaño=0x128)
00000000 dwTamaño dd ?
00000004 cntUso dd ?
00000008 th32ID de proceso dd?
```

```
0000000C th32DefaultHeapID dd?
00000010 th32ModuleID dd?
00000014 cntHilos dd ?
00000018 th32ParentProcessID dd?
0000001C pcPriClassBase dd?
00000020 dwFlags¿?
00000024 szExeFile db 260 duplicado (?)
00000128 PROCESSENTRY32 finaliza
```

Debido a que esta API toma dos parámetros, hSnapshot es EDI (línea 39, anteriormente el identificador devuelto desde CreateToolhelp32Snapshot en la línea 27), y lppe es la dirección de una variable local (EBP-0x130). Debido a que lppe apunta a un PROCESSENTRY32 estructura, sabemos inmediatamente que la variable local en EBP-0x130 es del mismo tipo. También tiene sentido porque la documentación de Process32First indica que antes de llamar a la función, el campo dwSize debe establecerse en el tamaño de una estructura PROCESSENTRY32 (que es 0x128). Ahora sabemos que las líneas 19 a 25 simplemente inicializaban esta estructura en 0. Además, podemos decir que esta variable local comienza en EBP-0x130 y termina en EBP-0x8.

La línea 42 prueba el valor de retorno de Process32Next. Si es cero, la ejecución comienza en la línea 70; de lo contrario, continúa en la línea 43. La línea 44 guarda la dirección de la función stricmp en ESI. La línea 45 establece ECX en la dirección de una variable local (EBP-0x10C), que resulta ser un campo en PROCESSENTRY32 (consulte el párrafo anterior). Las líneas 46 a 48 introducen 0x10007C50/ECX en la pila y llaman a stricmp. Sabemos que stricmp toma dos cadenas de caracteres como argumentos; por lo tanto, ECX debe ser el campo szExeFile en PROCESSENTRY32 y 0x10007C50 es la dirección de una cadena:

```
.datos:10007C50 65 78 70 6C 6F+Str2 base de datos 'explorer.exe',0
```

La línea 49 limpia la pila porque stricmp usa la convención de llamada CDECL . La línea 50 comprueba el valor de retorno de stricmp . Si es cero, lo que significa que la cadena coincide con "explorer.exe", la ejecución comienza en la línea 66; de lo contrario, continúa en la línea 52. Ahora podemos descompilar las líneas 18 a 51 de la siguiente manera:

```
MANEJAR h;
PROCESAMIENTO32 procentry;
h = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
si (h == VALOR_IDENTIFICADOR_INVÁLIDO) { devolver FALSO; }

memset(&procentry, 0, tamaño de(PROCESSENTRY32));
procentry.dwSize = tamaño de(procentry); // 0x128
si (Proceso32Siguiente(h, &procentry) == FALSO) {
    // línea 70
    ...
}
si (stricmp(procentry.szExeFile, "explorer.exe") == 0) {
    // línea 66
    ...
}
// línea 52
```

Las líneas 52 a 65 son casi idénticas al bloque anterior, excepto que forman un bucle con dos condiciones de salida. La primera condición de salida es cuando Process32Next devuelve FALSO (línea 58) y el segundo es cuando stricmp devuelve cero. Podemos descompilar las líneas 52 a 65 de la siguiente manera:

```
mientras (Process32Next(h, &procentry) != FALSO) {
    si (stricmp(procentry.szExeFile, "explorer.exe") == 0)
        romper;
}
```

Una vez que el bucle sale, la ejecución se reanuda en la línea 66. Las líneas 67 y 68 guardan el th32ParentProcessID/th32ProcessID de PROCESSENTRY32 coincidente en EAX/ECX y continúan la ejecución en 37. Observe que la línea 66 también es un destino de salto en la línea 43.

Las líneas 70 a 74 leen el parámetro fdwReason de DllMain (EBP+C) y comprueban si es 0 (DLL_PROCESS_DETACH). Si es así, el valor de retorno se establece en 0 y retorna; de lo contrario, pasa a la línea 82. Las líneas 82 a 85 comprueban si fdwReason es mayor que 1 (es decir, DLL_THREAD_ATTACH, DLL_THREAD_DETACH). Si es así, el valor de retorno se establece en 1 y retorna; de lo contrario, la ejecución continúa en la línea 86. Las líneas 86 a 92 llaman a CreateThread:

```
MANEJAR WINAPI CreateThread(
    _En_opt_ ATRIBUTOS_DE_SEGURIDAD_LP Atributos_de_subproceso_lp,
    _En_         TAMANO_T dwStackSize,
    _En_         LPTHREAD_START_ROUTINE Dirección_de_inicio_de_lp,
    _En_opt_ LPVOID parámetro_lp,
    _En_         DWORD dwCreationFlags,
    _Out_opt_ LPDWORD lpThreadId
);
```

con lpStartAddress como 0x100032D0. Este bloque se puede descompilar de la siguiente manera:

```
si (fdwReason == DLL_PROCESS_DETACH) { devolver FALSO; }
si (fdwReason == DLL_THREAD_ATTACH || fdwReason == DLL_THREAD_DETACH) {
    devuelve VERDADERO;
CrearHilo(0, 0, (RUTINA_DE_INICIO_LPTHREAD) 0x100032D0, 0, 0, 0);
devuelve VERDADERO;
```

Habiendo analizado la función, podemos deducir que la original del desarrollador La intención era esta:

1. Detectar si la máquina de destino tiene un IDT “sensato”.
2. Compruebe si “explorer.exe” se está ejecutando en el sistema, es decir, si alguien conectado
3. Crea un hilo principal que infecte la máquina de destino.

Ceremonias

1. Repita el procedimiento usted mismo. Dibuje el diseño de la pila, incluidos los parámetros y las variables locales.
2. En el ejemplo, hicimos una traducción casi uno a uno del código ensamblador a C. Como ejercicio, vuelva a descompilar toda esta función para que se vea más natural. ¿Qué puede decir sobre el nivel de habilidad/experiencia del desarrollador? Explique sus razones. ¿Puede hacer un mejor trabajo?
3. En algunas de las listas de ensamblajes, el nombre de la función tiene un prefijo @ seguido de un número. Explique cuándo y por qué existe esta decoración.
4. Implemente las siguientes funciones en el ensamblado x86: strlen, strchr, memcpy, memset, strcmp, strset.
5. Descompila las siguientes rutinas del kernel en Windows:

- KeInitializeDpc
- InicializarApc
- ObFastDereferenceObject (y explique su convención de llamada)
- KeInitializeQueue
- KxWaitForLockChainValid
- Hilo KeReady
- KiInitializeTSS
- Cadena Unicode RtlValidate

6. Ejemplo H. La función sub_13846 hace referencia a varias estructuras cuyos tipos no están del todo claros. Su tarea consiste en recuperar primero el prototipo de la función y luego intentar reconstruir los campos de la estructura. Después de leer el Capítulo 3, vuelva a este ejercicio para ver si ha cambiado su comprensión. (Nota: este ejemplo está destinado a Windows XP x86).
7. Ejemplo H. La función sub_10BB6 tiene un bucle que busca algo. Primero recupera el prototipo de la función y luego infiere los tipos en función del contexto. Sugerencia: probablemente deberías tener una copia de la especificación PE cerca.
8. Ejemplo H. Descompila sub_11732 y explica la construcción de programación más probable utilizada en el código original.
9. Ejemplo L. Explique qué función realiza sub_1000CEA0 y luego descompílela. De vuelta a C.

10. Si el nivel de privilegio actual está codificado en CS, que se puede modificar mediante Código de modo de usuario, ¿por qué el código de modo de usuario no puede modificar CS para cambiar CPL?
 11. Lea el capítulo Memoria virtual en el Manual del desarrollador de software Intel. Manual del programador de arquitectura AMD64 y volumen 3 , volumen 2: Sistema Programación. Realice usted mismo algunas traducciones de direcciones virtuales a direcciones físicas y verifique el resultado con un depurador de kernel. Explique cómo funciona la prevención de ejecución de datos (DEP).
 12. La biblioteca de desensamblaje x86/x64 favorita de Bruce es BeaEngine de BeatriX (www.beaengine.org). Experimente con ello escribiendo un programa para desmontar un binario en su punto de entrada.

x64

x64 es una extensión de x86, por lo que la mayoría de las propiedades de la arquitectura son las mismas, con pequeñas diferencias como el tamaño de los registros y algunas instrucciones que no están disponibles (como PUSHAD). Las siguientes secciones analizan las diferencias relevantes.

Conjunto de registros y tipos de datos

El conjunto de registros tiene 18 GPR de 64 bits y se puede ilustrar como se muestra en la Figura 1-6. Tenga en cuenta que los registros de 64 bits tienen el prefijo "R".



Figura 1-6

Si bien RBP todavía se puede utilizar como puntero de marco base, rara vez se utiliza para ese propósito en el código generado por el compilador de la vida real. La mayoría de los compiladores x64 simplemente tratan a RBP como otro GPR y hacen referencia a variables locales relativas a RSP.

Movimiento de datos

x64 admite un concepto denominado direccionamiento relativo a RIP, que permite que las instrucciones hagan referencia a datos en una posición relativa a RIP. Por ejemplo:

```
01: 0000000000000000 48 8B 05 00 00+ movimiento          rax, qword ptr cs:loc_A
02:                                         ; originalmente escrito como "mov rax,
[rotura]"
03: 0000000000000000 ubicación_A:
```

04:0000000000000007 48 31 C0	xor	Rax, Rax
05: 000000000000000A 90	no	

La línea 1 lee la dirección de loc_A (que es 0x7) y la guarda en RAX. RIP: el direccionamiento relativo se utiliza principalmente para facilitar el código independiente de la posición.

La mayoría de las instrucciones aritméticas se promueven automáticamente a 64 bits, incluso aunque los operandos son de solo 32 bits, por ejemplo:

48 B8 88 77 66+ movimiento de rax, 1122334455667788h		
31 C0	xor eax, eax; también borrarás los 32 bits superiores de RAX.	
		; es decir, RAX=0 después de esto
48 C7 C0 FF FF+ mov rax,0xFFFFFFFFFFFFFFFh		
FF C0	inc eax; RAX=0 después de esto	

Dirección canónica

En x64, las direcciones virtuales tienen un ancho de 64 bits, pero la mayoría de los procesadores no admiten un espacio de direcciones virtuales completo de 64 bits. Los procesadores Intel/AMD actuales solo utilizan 48 bits para el espacio de direcciones. Todas las direcciones de memoria virtual deben estar en formato canónico. Una dirección virtual está en formato canónico si los bits 63 hasta el bit implementado más significativo son todos 1 o 0. En términos prácticos, significa que los bits 48 a 63 deben coincidir con el bit 47. Por ejemplo:

0xfffff801'c9c11000 = 11111111 11111111 11110000 00000001 11001001 11000001		
00010000 00000000 ; canónico		
0x0000007f'bdb67000 = 00000000 00000000 00000111 11110111 10111101 10110110		
01110000 00000000 ; canónico		
0xfffff800'00000000 = 11111111 11111111 00001000 00000000 00000000 00000000		
00000000 00000000 ; no canónico		
0xfffff8000'00000000 = 11111111 11111111 10000000 00000000 00000000 00000000		
00000000 00000000 ; canónico		
0xfffff960'00098910 = 11111111 11111111 11110001 01100000 00000000 00001001		
10001001 11110000 ; canónico		

Si el código intenta desreferenciar una dirección no canónica, el sistema provocará una excepción.

Invocación de función

Recuerde que algunas convenciones de llamada requieren que los parámetros se pasen a la pila en x86. En x64, la mayoría de las convenciones de llamada pasan los parámetros a través de registros. Por ejemplo, en Windows x64, solo hay una convención de llamada y los primeros cuatro parámetros se pasan a través de RCX, RDX, R8 y R9; los restantes se colocan en la pila de derecha a izquierda. En Linux, los primeros seis parámetros se pasan a través de RDI, RSI, RDX, RCX, R8 y R9.

NOTA Para obtener más información sobre x64 ABI en Windows, consulte la sección “Convenciones de software x64” en MSDN (<http://msdn.microsoft.com/en-us/biblioteca/7kcdt6fy.aspx>).

Ceremonias

1. Explique dos métodos para obtener el puntero de instrucción en x64. Al menos uno de ellos Los métodos deben utilizar direccionamiento RIP.
2. Realice una traducción de direcciones virtuales a físicas en x64. ¿Hubo diferencias importantes en comparación con x86?

CAPÍTULO

2

BRAZO

A finales de los años 80 , una empresa llamada Acorn Computers desarrolló una arquitectura RISC de 32 bits denominada Acorn RISC Machine (que más tarde pasó a llamarse Advanced RISC Machine) . Esta arquitectura resultó útil más allá de su limitada línea de productos, por lo que se formó una empresa llamada ARM Holdings para obtener la licencia de la arquitectura para su uso en una amplia variedad de productos. Se encuentra habitualmente en dispositivos integrados como teléfonos móviles, dispositivos electrónicos para automóviles, reproductores de MP3, televisores, etc. La primera versión de la arquitectura se introdujo en 1985 y, en el momento de escribir este artículo, se encuentra en la versión 7 (ARMv7). ARM ha desarrollado una serie de núcleos específicos (por ejemplo, ARM7, ARM7TDMI, ARM926EJS, Cortex), que no deben confundirse con las diferentes especificaciones de arquitectura, que se numeran como ARMv1–ARMv7. Si bien hay varias versiones, la mayoría de los dispositivos son ARMv4, 5, 6 o 7. Las versiones ARMv4 y v5 son relativamente “antiguas”, pero también son las más dominantes y comunes del procesador (según el marketing de ARM, existen más de 10 mil millones de núcleos). Los productos electrónicos de consumo más populares suelen utilizar versiones más recientes de la arquitectura. Por ejemplo, el iPod Touch y el iPhone de tercera generación de Apple funcionan con un chip ARMv6, y los dispositivos iPhone/iPad y Windows Phone 7 posteriores funcionan con ARMv7.

Mientras que empresas como Intel y AMD diseñan y fabrican sus procesadores, ARM sigue un modelo ligeramente diferente. ARM diseña la arquitectura y cede la licencia a otras empresas, que luego fabrican e integran los procesadores en sus dispositivos. Empresas como Apple, NVIDIA, Qualcomm y Texas Instruments comercializan sus propios procesadores (A, Tegra, Snapdragon,

y OMAP, respectivamente), pero su arquitectura central está licenciada por ARM.

Todos ellos implementan el conjunto de instrucciones base y el modelo de memoria definidos en el manual de referencia de la arquitectura ARM. Se pueden añadir extensiones adicionales al procesador; por ejemplo, la extensión Jazelle permite ejecutar el bytecode de Java de forma nativa en el procesador. La extensión Thumb añade instrucciones que pueden tener 16 o 32 bits de ancho, lo que permite una mayor densidad de código (las instrucciones ARM nativas siempre tienen 32 bits de ancho). La extensión Debug permite a los ingenieros analizar el procesador físico utilizando hardware de depuración especial. Cada extensión suele estar representada por una letra (J, T, D, etc.). Según sus requisitos, los fabricantes pueden decidir si necesitan licenciar estas extensiones adicionales. Por eso, los procesadores ARMv6 y anteriores tienen letras después de ellos (por ejemplo, ARM1156T2 significa ARMv6 con extensión Thumb-2). Estas convenciones ya no se utilizan en ARMv7, que en su lugar utiliza tres perfiles (Aplicación, Tiempo real y Microcontrolador) y un nombre de modelo (Cortex) con diferentes características. Por ejemplo, la serie ARMv7 Cortex-A son procesadores con perfil de aplicación; y la serie Cortex-M están pensados para microcontroladores y solo admiten la ejecución en modo Thumb.

Este capítulo cubre la arquitectura ARMv7 tal como se define en la Arquitectura ARM.
Manual de referencia: Edición ARMv7-A y ARMv7-R (ARM DDI 0406B).

Características básicas

Debido a que ARM es una arquitectura RISC, existen algunas diferencias básicas entre las arquitecturas ARM y CISC (x86/x64). (Desde una perspectiva práctica, las nuevas versiones de los procesadores Intel también tienen algunas características RISC, es decir, no son "puramente" CISC). En primer lugar, el conjunto de instrucciones ARM es muy pequeño en comparación con x86, pero ofrece registros de propósito más general. En segundo lugar, la longitud de la instrucción es de ancho fijo (16 bits o 32 bits, según el estado). En tercer lugar, ARM utiliza un modelo de carga y almacenamiento para el acceso a la memoria. Esto significa que los datos deben trasladarse de la memoria a los registros antes de ser operados, y solo las instrucciones de carga y almacenamiento pueden acceder a la memoria. En ARM, esto se traduce en las instrucciones LDR y STR . Si desea incrementar un valor de 32 bits en una dirección de memoria en particular, primero debe cargar el valor en esa dirección en un registro, incrementarlo y almacenarlo nuevamente. A diferencia de x86, que permite que la mayoría de las instrucciones operen directamente sobre los datos en la memoria, una operación tan simple en ARM requeriría tres instrucciones (una de carga, una de incremento y una de almacenamiento). Esto puede implicar que hay más código para leer para la ingeniería inversa, pero en la práctica realmente no importa mucho una vez que uno se acostumbra.

ARM también ofrece varios niveles de privilegios diferentes para implementar el aislamiento de privilegios. En x86, los privilegios se definen mediante cuatro anillos, siendo el anillo 0 el que tiene el

El anillo 3 tiene el privilegio más alto y el anillo 0 el más bajo. En ARM, los privilegios se definen mediante ocho modos diferentes:

- Usuario (USR)
- Solicitud de interrupción rápida (FIQ)
- Solicitud de interrupción (IRQ)
- Supervisor (SVC)
- Monitoreo (MON)
- Abortar (ABT)
- Indefinido (UND)
- Sistema (SYS)

El código que se ejecuta en un modo determinado tiene acceso a ciertos privilegios y registros que otros no pueden tener; por ejemplo, al código que se ejecuta en modo USR no se le permite modificar los registros del sistema (que normalmente se modifican solo en modo SVC).

USR es el modo menos privilegiado. Si bien existen muchas diferencias técnicas, para simplificar, se puede hacer la analogía de que USR es como el anillo 3 y SVC es como el anillo 0. La mayoría de los sistemas operativos implementan el modo kernel en SVC y el modo usuario en USR. Tanto Windows como Linux lo hacen.

Si recuerdas el Capítulo 1, los procesadores x64 pueden ejecutarse en 32 bits, 64 bits o ambos indistintamente. Los procesadores ARM son similares en el sentido de que también pueden funcionar en dos estados: ARM y Thumb. El estado ARM/Thumb determina solo el conjunto de instrucciones, no el nivel de privilegio. Por ejemplo, el código que se ejecuta en modo SVC puede ser ARM o Thumb. En el estado ARM, las instrucciones siempre tienen un ancho de 32 bits; en el estado Thumb, las instrucciones pueden tener un ancho de 16 bits o 32 bits. El estado en el que se ejecuta el procesador depende de dos condiciones:

- Al ramificarse con las instrucciones BX y BLX , si el bit menos significativo del registro de destino es 1, entonces cambiará al estado Thumb. (Aunque las instrucciones están alineadas en 2 o 4 bytes, el procesador ignorará el bit menos significativo, por lo que no habrá problemas de alineación).
- Si el bit T del registro de estado del programa actual (CPSR) está configurado, entonces está en modo Thumb. La semántica de CPSR se explica en la siguiente sección, pero por ahora puede pensar en él como un registro EFLAGS extendido en x86.

Cuando se inicia un núcleo ARM, la mayoría de las veces entra en estado ARM y permanece así hasta que se produce un cambio explícito o implícito en Thumb. En la práctica, muchos códigos de sistemas operativos recientes utilizan principalmente código Thumb porque se desea una mayor densidad de código (una combinación de instrucciones de 16/32 bits de ancho puede ser más pequeña que todas las de 32 bits); las aplicaciones pueden funcionar en cualquier modo que deseen.

Si bien la mayoría de las instrucciones Thumb y ARM tienen el mismo mnemónico, las instrucciones Thumb de 32 bits tienen un sufijo .W .

NOTA Es un error común pensar que Thumb es como el modo real y ARM es como el modo protegido en x86/x64. No lo piense de esta manera. La mayoría de los sistemas operativos en la plataforma x86/x64 se ejecutan en modo protegido y rara vez, o nunca, vuelven al modo real. Los sistemas operativos y las aplicaciones en la plataforma ARM pueden ejecutarse tanto en estado ARM como en Thumb indistintamente. Tenga en cuenta también que estos estados son completamente diferentes de los modos de privilegio explicados en el párrafo anterior (USR, SVC, etc.).

Hay dos versiones de Thumb: Thumb-1 y Thumb-2. Thumb-1 se utilizó en ARMv6 y arquitecturas anteriores, y sus instrucciones siempre tienen un ancho de 16 bits. Thumb-2 amplía esta característica añadiendo más instrucciones y permitiendo que tengan un ancho de 16 o 32 bits. ARMv7 requiere Thumb-2, por lo que siempre que hablamos de Thumb, nos referimos a Thumb-2.

Existen otras diferencias entre los estados ARM y Thumb, pero no podemos cubrirlas todas aquí. Por ejemplo, algunas instrucciones están disponibles en el estado ARM pero no en el estado Thumb, y viceversa. Puede consultar la documentación oficial de ARM para obtener más detalles.

Además de tener diferentes estados de ejecución, ARM también admite la ejecución condicional. Esto significa que una instrucción codifica ciertas condiciones aritméticas que deben cumplirse para que se ejecute. Por ejemplo, una instrucción puede especificar que solo se ejecutará si el resultado de la instrucción anterior es cero. Contraste esto con x86, para el cual casi cada instrucción se ejecuta incondicionalmente. (Intel tiene un par de instrucciones que admiten directamente la ejecución condicional: CMOV y SETNE). La ejecución condicional es útil porque reduce las instrucciones de bifurcación (que son muy caras) y reduce la cantidad de instrucciones a ejecutar (lo que conduce a una mayor densidad de código). Todas las instrucciones en el estado ARM admiten la ejecución condicional, pero por defecto se ejecutan incondicionalmente. En el estado Thumb, se requiere una instrucción especial IT para habilitar la ejecución condicional.

Otra característica exclusiva de ARM es el desplazador de barril. Algunas instrucciones pueden "contener" otra instrucción aritmética que desplaza o rota un registro. Esto es útil porque puede reducir varias instrucciones a una sola; por ejemplo, desea multiplicar un registro por 2 y luego almacenar el resultado en otro registro.

Normalmente, esto requeriría dos instrucciones (una multiplicación seguida de un movimiento), pero con el desplazador de barril puedes incluir la multiplicación (desplazamiento a la izquierda en 1) dentro de la instrucción MOV. La instrucción sería algo como lo siguiente:

MOV R1, R0, LSL n. ^o 1	,R1 = R0 * 2
-----------------------------------	--------------

Tipos de datos y registros

De manera similar a los lenguajes de alto nivel, ARM admite operaciones con distintos tipos de datos. Los tipos de datos admitidos son: 8 bits (byte), 16 bits (media palabra), 32 bits (palabra) y 64 bits (palabra doble).

La arquitectura ARM define dieciséis registros de propósito general de 32 bits, numerados R0 , R1, R2, . . . , R15. Si bien todos ellos están disponibles para el programador de aplicaciones , en la práctica los primeros 12 registros son para uso de propósito general (como EAX, EBX, etc., en x86) y los últimos tres tienen un significado especial en la arquitectura:

- R13 se denomina puntero de pila (SP). Es el equivalente de ESP/RSP en x86/x64. Apunta a la parte superior de la pila del programa.
- R14 se denomina registro de enlace (LR). Normalmente contiene la dirección de retorno durante una llamada de función. Ciertas instrucciones utilizan implícitamente este registro. Por ejemplo, BL siempre almacena la dirección de retorno en LR antes de bifurcarse al destino. x86/x64 no tiene un registro equivalente porque siempre almacena la dirección de retorno en la pila. En el código que no utiliza LR para almacenar la dirección de retorno, se puede utilizar como un registro de propósito general.
- R15 se denota como el contador de programa (PC). Cuando se ejecuta en estado ARM, PC es la dirección de la instrucción actual más 8 (dos instrucciones ARM por delante); en estado Thumb, es la dirección de la instrucción actual más 4 (dos instrucciones Thumb de 16 bits por delante). Es análogo a EIP/RIP en x86/x64 excepto que siempre apuntan a la dirección de la siguiente instrucción que se ejecutará. Otra diferencia importante es que el código puede leer y escribir directamente en el registro de PC. Escribir una dirección en PC hará que la ejecución comience inmediatamente en esa dirección. Esto se puede explicar un poco más para evitar confusiones. Considere el siguiente fragmento en estado Thumb:

```
1: 0x00008344 inserción
2: 0x00008346 movimiento      r0, ordenador
3: 0x00008348 movimiento w r2, r1, lsl #31
4: 0x0000834c aparece
```

Después de ejecutar la línea 2, R0 contendrá el valor 0x0000834a (=0x00008346+4):

```
(gdb) br principal
Punto de interrupción 1 en 0x8348
...
Punto de interrupción 1, 0x00008348 en main()
(gdb) desaconseja principal
```

```
Volvido de código ensamblador para la función principal:
0x00008344 <+0>:           empujar      (apuntando al resultado)
    0x00008346 <+2>:           movimiento   r0, ordenador
=> 0x00008348 <+4>:           movimiento w r2, r1, lsl #31
    0x0000834c <+8>:           LSL          (apuntando a r0)
    0x0000834e <+10>:          pop         r0, r0, #0

Fin del volcado del ensamblador.
(gdb) información del registro pc
0x8348 0x8348 <principal+4>

registro de información pc (gdb) r0
r0          0x634a 33610
```

Aquí establecemos un punto de interrupción en 0x00008348. Cuando lo alcanza, mostramos el registro PC y R0 ; como se muestra, PC apunta a la tercera instrucción en 0x00008348 (a punto de ejecutarse) y R0 muestra el valor PC leído anteriormente. En este ejemplo, puede ver que cuando se lee directamente PC, sigue la definición; pero cuando se realiza la depuración, PC apunta a la instrucción que se debe ejecutar.

La razón de esta peculiaridad se debe a la segmentación heredada de los procesadores ARM más antiguos, que siempre obtenían dos instrucciones antes de la instrucción que se estaba ejecutando en ese momento. Hoy en día, las segmentaciones son mucho más complicadas, por lo que esto realmente no importa mucho, pero ARM mantiene esta definición para garantizar la compatibilidad con procesadores anteriores.

De manera similar a otras arquitecturas, ARM almacena información sobre el estado de ejecución actual en el registro de estado del programa actual (CPSR). Desde la perspectiva de un programador de aplicaciones, el CPSR es similar al registro EFLAGS/RFLAG en x86/x64. Es posible que en alguna documentación se hable del registro de estado del programa de aplicación (APSR), que es un alias para ciertos campos del CPSR. Hay muchos indicadores en el CPSR, algunos de los cuales se ilustran en la Figura 2-1 (otros se tratan en secciones posteriores).

- E (bit de endianidad): ARM puede funcionar en modo big endian o little endian.
Este bit se establece en 0 o 1 para little endian o big endian, respectivamente. La mayoría de las veces, se utiliza little endian, por lo que este bit será 0.
- T (bit Thumb): se configura si se encuentra en estado Thumb; de lo contrario, se encuentra en estado ARM. Una forma de realizar una transición explícita de Thumb a ARM (y viceversa) es modificar este bit.
- M (bits de modo): estos bits especifican el modo de privilegio actual (USR, SVC, etc.)



Controles y configuraciones a nivel de sistema

ARM ofrece el concepto de coprocesadores para admitir instrucciones adicionales y configuraciones a nivel de sistema. Por ejemplo, si el sistema admite una unidad de administración de memoria (MMU), entonces sus configuraciones deben exponerse al código de arranque o del núcleo. En x86/x64, estas configuraciones se almacenan en CR0 y CR4; en ARM, se almacenan en el coprocesador 15. Hay 16 coprocesadores en la arquitectura ARM, cada uno identificado por un número: CP0, CP1, , CP15. (Cuando se utilizan en código, se denominan P0, . . . , P15). Los primeros 13 son opcionales o reservados por ARM; los opcionales pueden ser utilizados por los fabricantes para implementar instrucciones o características específicas del fabricante . Por ejemplo, CP10 y CP11 se utilizan normalmente para compatibilidad con NEON y punto flotante. Cada coprocesador contiene "códigos de operación" y registros adicionales que pueden controlarse mediante instrucciones ARM especiales. CP14 y CP15 se utilizan para depuración y configuración del sistema; CP15, normalmente conocido como el control del sistema coprocesador, almacena la mayoría de las configuraciones del sistema (almacenamiento en caché, paginación, excepciones, etc.).

NOTA: NEON proporciona el conjunto de instrucciones SIMD (instrucción única, múltiples datos) que se utiliza habitualmente en aplicaciones multimedia. Es similar a las instrucciones SSE/MMX. en arquitecturas basadas en x86.

Cada coprocesador tiene 16 registros y ocho códigos de operación correspondientes. La semántica de estos registros y códigos de operación es específica del coprocesador. El acceso a los coprocesadores solo se puede realizar a través de las instrucciones MRC (lectura) y MCR (escritura); toman un número de coprocesador, un número de registro y códigos de operación. Por ejemplo, para leer el registro base de traducción (similar a CR3 en x86/x64) y guardarlo en R0, se utiliza lo siguiente:

MRC p15, 0, r0, c2, c0, 0; guardar TTBR en r0

Esto dice, “leer el registro C2/C0 del coprocesador 15 usando el código de operación 0/0 y almacenar el resultado en el registro de propósito general R0”. Debido a que hay tantos registros y códigos de operación dentro de cada coprocesador, debe leer la documentación para determinar el significado preciso de cada uno. Algunos registros (C13/C0) están reservados para los sistemas operativos con el fin de almacenar datos específicos del proceso o del subproceso.

Si bien las instrucciones MRC y MCR no requieren privilegios elevados (es decir, se pueden ejecutar en modo USR), algunos de los registros y códigos de operación del coprocesador solo son accesibles en modo SVC. Los intentos de leer ciertos registros sin privilegios suficientes darán como resultado una excepción. En la práctica, rara vez verá estas instrucciones en el código de modo de usuario; se encuentran comúnmente en código de nivel muy bajo, como ROM, cargadores de arranque, firmware o código de modo kernel.

Introducción al conjunto de instrucciones

En este punto, está listo para ver las instrucciones ARM importantes. Además de la ejecución condicional y los barrel shifters, hay varias otras peculiaridades sobre las instrucciones que no se encuentran en x86. Primero, algunas instrucciones pueden operar en un rango de registros en secuencia. Por ejemplo, para almacenar cinco registros, R6–R10, en una ubicación de memoria particular referenciada por R1, escribiría STM R1, {R6-R10}. R6 se almacenaría en la dirección de memoria R1, R7 en R1+4, R8 en R1+8, y así sucesivamente. Los registros no consecutivos también se pueden especificar mediante separación por comas (por ejemplo, {R1,R5,R8}). En la sintaxis de ensamblaje ARM, los rangos de registros generalmente se especifican dentro de llaves. Segundo, algunas instrucciones pueden actualizar opcionalmente el registro base después de una operación de lectura/escritura. Esto generalmente se hace agregando un signo de exclamación (!) después del nombre del registro. Por ejemplo, si reescribiera la instrucción anterior como STM R1!, {R6-R10} y la ejecutara, entonces R1 se actualizará con la dirección inmediatamente posterior a la que se almacenó R10 . Para que quede más claro, aquí se muestra un ejemplo:

```

01: (gdb) desas principal
02: Volcado de código ensamblador para la función principal:
03: => 0x00008344 <+0>;           movimiento      r6, n.º 10
04:     0x00008348 <+4>;           movimiento      r7, n.º 11
05:     0x0000834c <+8>;           movimiento      r8, n.º 12
06:     0x00008350 <+12>;          movimiento      r9, n.º 13
07:     0x00008354 <+16>;          movimiento      r10, n.º 14
08:     0x00008358 <+20>;         stmia spl, {r6, r7, r8, r9, r10}
09:     0x0000835c <+24>;         caja          lr
10: Fin del volcado del ensamblador.
11: (gdb) sí
12: 0x00008348 en el servidor principal ()
13: ...
14: 0x00008358 en el servidor principal ()
15: (gdb) información reg sp
16: sp 17:           0xbefdf5848           0xbefdf5848
(gdb) sí
18: 0x0000835c en principal()
19: (gdb) información reg sp
20: 21:           0xbefdf585c           0xbefdf585c
(gdb) x/6x 0xbefdf5848
22:0xbefdf5848:   0x0000000a           0x0000000b           0x0000000c
0x0000000d
23:0xbefdf5858:   0x0000000e           0x00000000

```

La línea 15 muestra el valor de SP (0xbefdf5848) antes de ejecutar la instrucción STM ; las líneas 17 y 19 ejecutan la instrucción STM y muestran el valor actualizado de SP. La línea 21 vuelca seis palabras comenzando en el valor anterior de SP. Tenga en cuenta que R6 se almacenó en el SP anterior, R7 en SP+0x4, R8 en SP+0x8, R9 en SP+0xc y R10 en SP+0x10. El nuevo SP (0xbefdf585c) está inmediatamente después de donde se almacenó R10 .

NOTA: STMIA y STMEA son pseudoinstrucciones para STM, es decir, tienen el mismo significado. Los desensambladores pueden elegir cualquiera de las dos para mostrar. Algunos mostrarán STMEA si el registro base es SP y STMIA para otros registros; algunos siempre usan STM; y algunos siempre usan STMIA. No hay una regla estricta, por lo que debe acostumbrarse a esto si está usando varios desensambladores.

Carga y almacenamiento de datos

En la sección anterior se menciona que ARM es una arquitectura de carga y almacenamiento, lo que significa que los datos deben cargarse en registros antes de poder operar sobre ellos. Las únicas instrucciones que pueden tocar la memoria son las de carga y almacenamiento; todas las demás instrucciones pueden operar únicamente sobre registros. Cargar significa leer datos de la memoria y guardarlos en un registro; almacenar significa escribir el contenido de un registro en una ubicación de memoria. En ARM, las instrucciones de carga y almacenamiento son LDR/STR, LDM/STM y PUSH/POP.

LDR y STR

Estas instrucciones pueden cargar y almacenar 1, 2 o 4 bytes en la memoria y desde ella. Su sintaxis completa es algo complicada porque hay varias formas diferentes de especificar el desplazamiento y los efectos secundarios para actualizar el registro base. Consideremos el caso más simple:

01:03 68	LDR	R3, [R0] ; R3 = *R0
02:23 60	STR	R3, [R4] ; *R4 = R3;

Para la instrucción de la línea 1, R0 es el registro base y R3 es el destino; carga el valor de la palabra en la dirección R0 en R3. En la línea 2, R4 es el registro base y R3 es el destino; toma el valor en R3 y lo almacena en la dirección de memoria R4. Este ejemplo es simple porque la dirección de memoria está especificada por el registro base.

En un nivel fundamental, las instrucciones LDR/STR toman un registro base y un desplazamiento; hay tres formas de desplazamiento y tres modos de direccionamiento para cada forma.

Comenzamos discutiendo las formas de desplazamiento: inmediato, registro y registro escalado.

La primera forma de desplazamiento utiliza un valor inmediato como desplazamiento. Un valor inmediato es simplemente un entero. Se suma o se resta del registro base para acceder a los datos en un desplazamiento conocido en el momento de la compilación. El uso más común es acceder a un campo particular en una estructura o tabla virtual. El formato general es el siguiente:

- STR Ra, [Rb, imm]
- LDR Ra, [Rc, imm]

Rb es el registro base e imm es el desplazamiento que se agregará a Rb.

Por ejemplo, supongamos que R0 contiene un puntero a una estructura KDPC y el siguiente código:

Definición de estructura

```
0:000> !Hola, ntkrnlmp!_KDPC!
+0x000 Tipo :UCar
+0x001 Importancia :UCar
+0x002 Número :UInt2B
+0x004 Entrada de lista Dpc :_ENTRADA_DE_LISTA
+0x00c Rutina diferida: Ptr32 void
+0x010 DeferredContext : Ptr32 Vacío
+0x014 Argumento del sistema1: Ptr32 nulo
+0x018 Argumento del sistema2: Ptr32 nulo
+0x01c Datos de Dpc : Ptr32 Vacío
```

Código

01:13 23	Movimientos	R3, #0x13
02:03 70	mov-----	R3, [R0]
03:01 23	Movimientos	R3, n. ^o 1
04:4370	-----	R3, [R0,#1]
05:00 23	Movimientos	R3, #0
06:4380	ESTRELLA	R3, [R0,#2]
07: C3 61	STR	R3, [R0,#0x1C]
08: C1 60	STR	R1, [R0,#0xC]
09:02 61	STR	R2, [R0,#0x10]

En este caso, R0 es el registro base y los inmediatos son 0x1, 0x2, 0xC, 0x10 y 0x1C. El fragmento se puede traducir a C de la siguiente manera:

```
KDPC *obj = ...; obj->Tipo /* R0 es objeto */
= 0x13;
obj->Importancia = 0x1;
obj->Número = 0x0;
obj->DpcData = NULL;
obj->RutinaDiferida = R1; obj->ContextoDiferido = R2; /*R1 es desconocido para nosotros*/
/*R2 es desconocido para nosotros*/
```

Esta forma de desplazamiento es similar al MOV Reg, [Reg + Imm] en x86/x64.

La segunda forma de desplazamiento utiliza un registro como desplazamiento. Se utiliza habitualmente en código que necesita acceder a una matriz, pero el índice se calcula en tiempo de ejecución. El formato general es el siguiente:

- STR Ra, [Rb, Rc]
- LDR Ra, [Rb, Rc]

Según el contexto, Rb o Rc pueden ser la base o el desplazamiento. Consideremos los dos ejemplos siguientes:

Ejemplo 1

```

01:03 F0 F2 FA BL estirar
02:06 46           Movimiento R6, R0
; R0 es el valor de retorno de strien
03: ...
04: BB57           LDRSB R3, [R7,R6]
; en este caso, R6 es el desplazamiento

```

Ejemplo 2

```

01: B3 EB 05 08 SUBS.W R8, R3, R5
02:2F 78           LDRB      R7, [R5]
03: 18 F8 05 30 LDRB.W R3, [R8,R5]
; aquí, R5 es la base y R8 es el desplazamiento
04:9F 42           CMP       R7, R3

```

Esto es similar al formato `MOV Reg, [Reg + Reg]` en x86/x64.

La tercera forma de desplazamiento utiliza un registro escalado como desplazamiento. Se utiliza habitualmente en un bucle para iterar sobre una matriz. El desplazador de barril se utiliza para escalar el desplazamiento. El formato general es el siguiente:

- `LDR Ra, [Rb, Rc, <desplazador>]`
- `STR Ra, [Rb, Rc, <cambiador>]`

Rb es el registro base; Rc es un inmediato; y `<shifter>` es la operación realizada en el inmediato, normalmente, un desplazamiento hacia la izquierda/derecha para escalar el inmediato.

Por ejemplo:

```

01:0E4B           LDR      R3, =KeNumberNodes
02: ...
03:00 24           Movimiento R4, #0
04:19 88           LDH      R1, [R3]
05:09 48           LDR      R0, =KeNodeBlock
06:00 23           Movimiento R3, #0
07:               inicio de bucle
08:50 F8 23 20 LDR.W R2, [R0,R3,LSL#2]
09:00 23           Movimiento R3, #0
10:A2 F8 90 30 STRH W R3, [R2,#0x90]
11:92 F8 89 30 LDRB.W R3, [R2,#0x89]
12:53 F0 02 03 ORRS.W R3, R3, #2
13:82 F8 89 30 STRB.W R3, [R2,#0x89]
14:63 1C           AGREGA   R3, R4, #1
15:9C B2           UXTH    R4, R3
16:23 46           Movimiento R3, R4
17:8C 42           CMP     R4, R1
18:EF-DB           BLT     inicio de bucle

```

50 Capítulo 2 ■ BRAZO

KeNumberNodes y KeNodeBlock son un entero global y una matriz de KNODE punteros, respectivamente.

Las líneas 1 y 5 simplemente cargan esas variables globales en un registro (explicaremos esta sintaxis más adelante). La línea 8 itera sobre la matriz KeNodeBlock (R0 es la base), R3 es el índice multiplicado por 4 (porque es una matriz de punteros; los punteros tienen un tamaño de 4 bytes en esta plataforma). Las líneas 10 a 13 inicializan algunos campos del elemento KNODE . Línea 14 incrementa el índice. La línea 17 compara el índice con el tamaño de la matriz (R1 es el tamaño; ver línea 4) y si es menor que el tamaño, entonces continúa el bucle.

Este fragmento se puede traducir aproximadamente a C de la siguiente manera:

```
int KeNumberNodes = ...;
KNODE *KeNodeBlock[KeNumberNodes] = ...;
para (int i=0; i < KeNumberNodes; i++) {
    KeNodeBlock[i].x = ...;
    KeNodeBlock[i].y = ...;
    ...
}
```

Esto es similar al formato MOV, Reg, [Reg + idx * escala] en x86/x64.

Habiendo cubierto las tres formas de desplazamiento, el resto de esta sección analiza los modos de direccionamiento: desplazamiento, preindexado y postindexado. La única distinción entre La pregunta es si el registro base se modifica y, en caso afirmativo, de qué manera. Todos los ejemplos de desplazamiento anteriores utilizan el modo de direccionamiento de desplazamiento, lo que significa que el registro base nunca se modifica. Este es el modo más simple y más común. Puede reconocerlo rápidamente porque no contiene un signo de exclamación (!) en ninguna parte y lo inmediato está dentro de los corchetes. (Algunas publicaciones categorizan estos modos como pre-índice, pre-índice con escritura diferida y post-índice).

(La terminología utilizada aquí refleja la documentación oficial de ARM). La sintaxis general para el modo de desplazamiento es LDR Rd, [Rn, desplazamiento].

El modo de dirección preindexada significa que el registro base se actualizará con la dirección de memoria final utilizada en la operación de referencia. La semántica es muy similar a la forma prefijada del operador unario ++ y -- en C. La sintaxis para este modo es LDR Rd, [Rn, offset]!. Por ejemplo:

```
12 F9 01 3D LDRSB.W R3, [R2,#-1]! ; R3 = *(R2-1)
;R2 = R2-1
```

El modo de dirección postindexada significa que el registro base se utiliza como la dirección final y luego se actualiza con el desplazamiento calculado. Esto es muy similar a la forma postfija del operador unario ++ y -- en C. La sintaxis para este modo es LDR Rd, [Rn], desplazamiento. Por ejemplo:

```
10 F9 01 6B LDRSB.W R6, [R0],#1 ; R6 = *R0
;R0 = R0+1
```

Las formas pre y post índice se observan normalmente en código que accede a un desplazamiento en el mismo búfer varias veces. Por ejemplo, supongamos que el código necesita realizar un bucle y verificar si un carácter en una cadena coincide con uno de cinco caracteres; el compilador puede actualizar el puntero base para que pueda eliminar un carácter. instrucción de incremento.

NOTA Aquí hay un consejo para reconocer y recordar los diferentes modos de dirección en LDR/

STR: Si hay un !, entonces es prefijo; si el registro base está entre paréntesis por sí solo, entonces es posfijo; cualquier otra cosa está en modo desfasado.

Otros usos de LDR

Como se explicó anteriormente, LDR se utiliza para cargar datos desde la memoria a un registro; sin embargo, a veces lo verá en estas formas:

```
01: DF F8 50 82 LDR.W R8, =0x2932E00 ; LDR R8, [PC, x]
02:80 4A          LDR      R2, =a04d ; "%04d" ; LDR R2, [PC, y]
03:0E 4B          LDR      R3,=__imp_realloc ; LDR R3, [PC, z]
```

Claramente, esta no es una sintaxis válida según la sección anterior. Técnicamente, se denominan pseudoinstrucciones y las utilizan los desensambladores para facilitar la inspección manual. Internamente, utilizan la forma inmediata de LDR con PC como registro base; a veces, esto se denomina direccionamiento relativo a PC (o relativo a RIP). Los binarios ARM suelen tener un grupo literal que es un área de memoria en una sección para almacenar constantes, cadenas o desplazamientos a los que otros pueden hacer referencia de manera independiente de la posición. (El grupo literal es parte del código, por lo que estará en la misma sección). En el fragmento anterior, el código hace referencia a una constante de 32 bits , una cadena y un desplazamiento a una función importada almacenada en el grupo literal. Esta pseudoinstrucción en particular es útil porque permite mover una constante de 32 bits a un registro en una sola instrucción. Para que quede más claro, considere el siguiente fragmento:

```
01:.text:0100B134 35 4B LDR          R3, =0x68DB8MALO
; en realidad LDR R3, [PC, #0xD4]
; en este punto, PC = 0x0100B138
02: ...
03: .text:0100B20C AD 8B DB 68 dword_100B20C DCD 0x68DB8BAD
```

¿Cómo acortó el desensamblador la primera instrucción de LDR R3, [PC, #0xD4] a la forma alternativa? Debido a que el código está en estado Thumb, PC es la instrucción actual más 4, que es 0x0100B138; está utilizando la forma inmediata de PC, por lo que está intentando leer la palabra en 0x0100B20C (=0x100B138+0xD4), que resulta ser la constante que queremos cargar.

Otra instrucción relacionada es ADR, que obtiene la dirección de una etiqueta/función. y lo pone en un registro. Por ejemplo:

```
01:0000939065A5          ADR      R5, palabra_dword_9528
02:00009392 D5 E9 00 45 LDRD.W R4, R5, [R5]
03: ...
04: 00009528 00 CE 22 A9+dword_9528 DCD 0xA922CE00 , 0xC0A4
```

Esta instrucción se utiliza normalmente para implementar tablas de salto o devoluciones de llamadas en las que es necesario pasar la dirección de una función a otra. Internamente, esta instrucción solo calcula un desplazamiento desde la PC y lo guarda en el registro de destino.

LDM y STM

LDM y STM son similares a LDR/STR, excepto que cargan y almacenan múltiples palabras en un registro base determinado. Son útiles cuando se mueven múltiples bloques de datos hacia y desde la memoria. La sintaxis general es la siguiente:

- LDM<modo> Rn[!], {Rm}
- STM<modo> Rn[!], {Rm}

Rn es el registro base y contiene la dirección de memoria desde la que se cargará o almacenará; el signo de exclamación opcional (!) significa que el registro base debe actualizarse con la nueva dirección (escritura diferida). Rm es el rango de registros que se cargarán o almacenarán. Hay cuatro modos:

- IA (Increment After) (Incrementar después): almacena datos comenzando en la ubicación de memoria especificada por la dirección base. Si hay escritura diferida, se vuelve a escribir la dirección 4 bytes por encima de la última ubicación. Este es el modo predeterminado si no se especifica nada.
- IB (Incremento previo): almacena datos a partir de la ubicación de memoria 4 bytes por encima de la dirección base. Si hay escritura diferida, se vuelve a escribir la dirección de la última ubicación.
- DA (Decrement After) (Decrementar después): almacena los datos de manera que la última ubicación sea la dirección base. Si hay escritura diferida, se vuelve a escribir la dirección 4 bytes por debajo de la ubicación más baja.
- DB (Decrement Before): almacena datos de manera que la última ubicación esté 4 bytes por debajo de la dirección base. Si hay escritura diferida, se vuelve a escribir la dirección de la primera ubicación.

Esto puede sonar un poco confuso al principio, así que veamos un ejemplo. con el depurador:

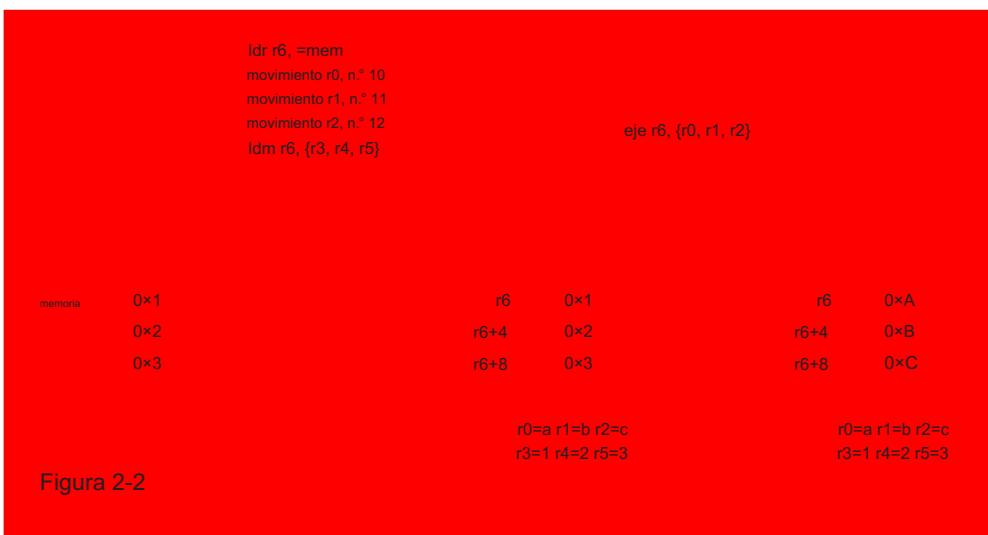
```
01: (gdb) br principal
02: Punto de interrupción 1 en 0x8344
03: (gdb) desas principal
04: Volcado de código ensamblador para la función principal:
```

```

05:    0x00008344 <+0>;           SetPoint      r6, =mem ; editado un poco
06:    0x00008348 <+4>;          movimiento   r0, n.º 10
07:    0x0000834c <+8>;          movimiento   r1, n.º 11
08:    0x00008350 <+12>;         movimiento   r2, n.º 12
09:    0x00008354 <+16>;         LDR.LDM     r6, {r3, r4, r5} ; modo IA
10:    0x00008358 <+20>;         sm          r6, {r0, r1, r2} ; modo IA
11: ...
12: (gdb) r
13: Punto de interrupción 1, 0x00008344 en main()
14: (gdb) si
15: 0x00008348 en el servidor principal ()
16: (gdb) x/3x $r6
17: 0x1050c <mem>; 0x00000001          0x00000002          0x00000003
18: (gdb) si
19: 0x0000834c en principal()
20: ...
21: (bdf)
22: 0x00008358 en el servidor principal ()
23: (gdb) información reg r3 r4 r5
24:r3          0x1      1
25:r4          0x2      2
26:r5          0x3      3
27: (gdb) si
28: 0x0000835c en principal()
29: (gdb) x/3x $r6
30: 0x1050c <mem>; 0x0000000a          0x0000000b          0x0000000c

```

La línea 5 almacena una dirección de memoria en R6; el contenido de esta dirección de memoria (0x1050c) son tres palabras (línea 17). Las líneas 6 a 8 configuran R0 a R2 con algunas constantes. La línea 9 carga tres palabras en R3 a R5, comenzando en la ubicación de memoria especificada por R6. Como se muestra en las líneas 24 a 26, R3 a R5 contienen el valor esperado. La línea 10 almacena R0 a R2, comenzando en la ubicación de memoria especificada por R6. La línea 29 muestra que se escribieron los valores esperados. La figura 2-2 ilustra el resultado de las operaciones anteriores.



Aquí está el mismo experimento con escritura diferida:

```

01: (gdb) br principal
02: Punto de interrupción 1 en 0x8344
03: (gdb) desas principal
04: Volcado de código ensamblador para la función principal:
05:      0x00008344 <+0>:           $señales          r6, =mem ; editado un poco
06:      0x00008348 <+4>:           $movimiento     r0, n.º 10
07:      0x0000834c <+8>:           $movimiento     r1, n.º 11
08:      0x00008350 <+12>:          $movimiento     r2, n.º 12
09:      0x00008354 <+16>:          LDM-LDM        r6!, {r3, r4, r5} ; modo IA con escritura diferida
10:      0x00008358 <+20>:          stmia r6!, {r0, r1, r2} ; modo IA con escritura diferida
11: ...
12: (gdb) r
13: Punto de interrupción 1, 0x00008344 en main()
14: (gdb) sf
15: 0x00008348 en el servidor principal ()
16: ...
17: (bdf)
18: 0x00008354 en el servidor principal ()
19: (gdb) x/3x $r6
20: 0x1050c <mem>: 0x00000001          0x00000002          0x00000003
21: (gdb) sf
22: 0x00008358 en el servidor principal ()
23: (gdb) información reg r6
24: r6                  0x10518 66840
25: (gdb) sf
26: 0x0000835c en principal()
27: (gdb) información reg $r6
28: r6                  0x10524 66852
29: (gdb) x/4x $r6-12
30: 0x10518 :            0x0000000a          0x0000000b          0x0000000c
0x00000000

```

La línea 9 utiliza el modo IA con escritura diferida, por lo que el r6 se actualiza con una dirección 4 bytes por encima de la última ubicación (línea 23). Lo mismo se puede observar en las líneas 10, 27 y 30. La figura 2-3 muestra el resultado del fragmento anterior.

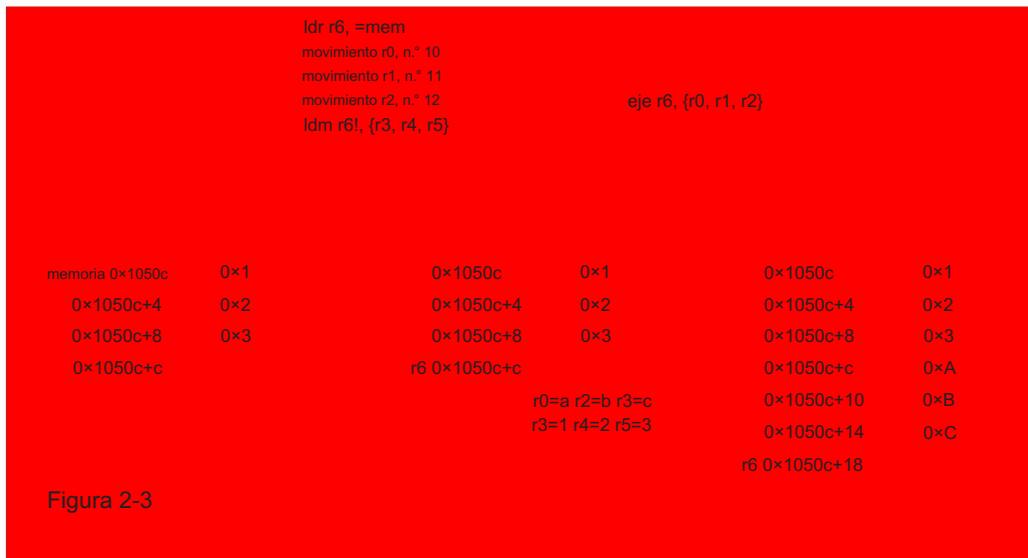


Figura 2-3

Debido a que LDM y STM pueden mover múltiples palabras a la vez, se utilizan normalmente en operaciones de copia en bloque o de movimiento. Por ejemplo, a veces se utilizan para insertar en línea memcpy cuando se conoce la longitud de la copia en el momento de la compilación. Son similares a la instrucción MOVS con el prefijo REP en x86. Considere los siguientes fragmentos de código generados por dos compiladores diferentes a partir del mismo archivo fuente:

Compilador A

01: A4 46	Movimiento	R12, R4
02: 35 46	Movimiento	R5, R6
03: BC E8 0F 00 LDMIA.W R12!, {R0-R3}		
04: 0F C5	STMIA	R5!, {R0-R3}
05: BC E8 0F 00 LDMIA.W R12!, {R0-R3}		
06: 0F C5	STMIA	R5!, {R0-R3}
07: 9C E8 0F 00 LDMIA.W R12, {R0-R3}		
08: 85 E8 0F 00 STMIA.W R5, {R0-R3}		

Compilador B

01: 30 22	Movimientos	R2, #0x30
02: 21 46	Movimiento	R1, R4
03: 30 46	Movimiento	R0, R6
04: 23 F0 17 FA BL		memoriacopy

Todo lo que hace es copiar 48 bytes de un búfer a otro; el primer compilador utiliza LDM/STM con escrituras diferidas para cargar/almacenar 16 bytes a la vez, mientras que el segundo simplemente llama a su implementación de memcpy. Al realizar ingeniería inversa del código, puede detectar la forma de memcpy en línea al reconocer que LDM/STM está utilizando los mismos punteros de origen y destino con el mismo conjunto de registros. Este es un buen truco a tener en cuenta porque lo verás a menudo.

Otro lugar común donde se pueden ver LDM/STM es al principio y al final de las funciones en el estado ARM. En este contexto, se utilizan como prólogo y epílogo. Por ejemplo:

01: F0 4F 2D E9 STMFD SP!, {R4-R11,LR} ; guardar regs + dirección de retorno
02: ...
03: F0 8F BD E8 LDMFD SP!, {R4-R11,PC} ; restaurar regs y regresar

STMFD y LDMFD son pseudoinstrucciones para STMDB y LMDIA/LDM, respectivamente.

NOTA: a menudo verá los sufijos FD, FA, ED o EA después de STM/LDM. Son simplemente pseudoinstrucciones para las instrucciones LDM/STM en diferentes modos (IA, IB, etc.). La asociación es STMFD/STMDB, STMFA/STMIB, STMED/STMDA, STMEA/STMIA, LDMFD/LDMIA, LDMA/LDMDA y LDMEA/LDMDB. Puede resultar un poco difícil memorizar estas asociaciones; la forma más eficaz es dibujar imágenes para cada una de ellas.

EMPUJAR y POP

El último conjunto de instrucciones de carga/almacenamiento es PUSH y POP. Son similares a LDM/STM excepto dos características:

- Utilizan implícitamente SP como dirección base.
- SP se actualiza automáticamente.

La pila crece hacia abajo hasta las direcciones inferiores, como ocurre en la arquitectura x86/x64. La sintaxis general es PUSH/POP {Rn}, donde Rn puede ser un rango de registros.

PUSH almacena los registros en la pila de modo que la última ubicación esté 4 bytes por debajo del puntero de pila actual y actualiza SP con la dirección de la primera ubicación. POP carga los registros comenzando desde el puntero de pila actual y actualiza SP con la dirección 4 bytes por encima de la última ubicación. PUSH/POP son en realidad lo mismo que STMDB/LDMIA con escritura diferida y SP como puntero base. A continuación se muestra un breve tutorial que demuestra las instrucciones:

```

01: (gdb) desas principal
02: Volcado de código ensamblador para la función principal:
03:     0x00008344 <+0>:           movimiento w r0, n.º 10
04:     0x00008348 <+4>:           movimiento w r1, n.º 11
05:     0x0000834c <+8>:           movimiento w r2, n.º 12
06:     0x00008350 <+12>:          empujar    {r0,r1,r2}
07:     0x00008352 <+14>:          estallido   {r3, r4, r5}
08: ...
09: (gdb) br principal
10: Punto de interrupción 1 en 0x8344
11: (gdb) r
12: Punto de interrupción 1, 0x00008344 en main()
13: (gdb) si
14: 0x00008348 en el servidor principal ()
15: ...
16: (bdf)
17: 0x00008350 en el servidor principal ()
18: (gdb) información reg sp           ; puntero de pila actual
19: sp 20:                         0xbbee56848          0xbbee56848
(gdb) si
21: 0x00008352 en el servidor principal ()
22: (gdb) x/3x $sp 23: 0xbbee5683c: ; sp se actualiza después del push
                                         0x0000000a          0x0000000b          0x0000000c
24: (gdb) si 25:                   ; entrar en los registros
0x00008354 en principal ()
26: (gdb) información reg r3 r4 r5 ; nuevos registros
27:r3                           0xa               10
28:r4                           0xb               11
29:r5                           0xc               12
30: (gdb) información reg sp       ; nuevo sp (4 bytes por encima de la última ubicación)
31: sp 32:                         0xbbee56848          0xbbee56848
(gdb) x/3x $sp 12
33:0xbbee5683c:                 0x0000000a          0x0000000b          0x0000000c

```

La figura 2-4 ilustra el fragmento anterior.

movimiento w r0, n. [#] 10				
movimiento w r1, n. [#] 11				
movimiento w r2, n. [#] 12			estallar {r3, r4, r5}	
empujar {r0, r1, r2}				
0×bee56848-c	sp0×bee56848-c	0xA	0×bee56848-c	0xA
0×bee56848-8	0×bee56848-8	0xB	0×bee56848-8	0xB
0×bee56848-4	0×bee56848-4	0xC	0×bee56848-4	0xC
es 0×bee56848	0×bee56848		es 0×bee56848	
0×abeja56848+4	0×abeja56848+4		0×abeja56848+4	
0×abeja56848+8	0×abeja56848+8		0×abeja56848+8	
0×bee56848+c	0×bee56848+c		0×bee56848+c	
		r0=a r1=b r2=c		r0=a r1=b r2=c
				r3=a r4=b r5=c

Figura 2-4

El lugar más común para PUSH/POP es al principio y al final de las funciones . En este contexto, se utilizan como prólogo y epílogo (como STMFD/ LDMFD en estado ARM). Por ejemplo:

```
01: 2D E9 F0 4F PUSH.W {R4-R11,LR} ; guardar registros + dirección de retorno
02: ...
03: BD E8 F0 8F POP.W {R4-R11,PC} ; restaurar registros y regresar
```

Algunos desensambladores realmente utilizan este patrón como heurística para determinar los límites de las funciones.

Funciones e invocación de funciones

A diferencia de x86/x64, que solo tiene una instrucción para la invocación de función (CALL) y ramificación (JMP), ARM ofrece varias dependiendo de cómo se codifique el destino . Cuando se llama a una función, el procesador necesita saber dónde reanudar la ejecución después de que la función regrese; esta ubicación se conoce normalmente como la dirección de retorno. En x86, la instrucción CALL empuja implícitamente la dirección de retorno en la pila antes de saltar a la función de destino; cuando termina de ejecutarse , la función de destino reanuda la ejecución en la dirección de retorno sacándola de la pila hacia EIP.

El mecanismo en ARM es esencialmente el mismo con algunas diferencias menores . Primero, la dirección de retorno se puede almacenar en la pila o en el registro de enlace (LR); para reanudar la ejecución después de la llamada, la dirección de retorno se extrae explícitamente de la pila en PC o habrá una bifurcación incondicional a LR. Segundo, una

La rama puede cambiar entre el estado ARM y el estado Thumb, dependiendo del LSB de la dirección de destino. En tercer lugar, ARM define una convención de llamada estándar: los primeros cuatro parámetros de 32 bits se pasan a través de registros (R0-R3) y el resto se almacena en la pila. El valor de retorno se almacena en R0.

Las instrucciones utilizadas para invocar funciones son B, BX, BL y BLX.

Aunque es raro ver B utilizado en el contexto de la invocación de una función, se puede utilizar para la transferencia de control. Es simplemente una bifurcación incondicional y es idéntica a la instrucción JMP en x86. Normalmente se utiliza dentro de bucles y condicionales para volver al principio o salir; también se puede utilizar para llamar a una función que nunca retorna. B solo puede utilizar desplazamientos de etiquetas como destino; no puede utilizar registros. En este contexto, la sintaxis de B es la siguiente: B imm, donde imm es un desplazamiento relativo a la instrucción actual. (Esto no tiene en cuenta los indicadores de ejecución condicional, que se analizan en la sección "Bifurcación y ejecución condicional"). Un hecho importante a tener en cuenta es que debido a que las instrucciones ARM y Thumb están alineadas en 4 y 2 bytes, el desplazamiento de destino debe ser un número par. Aquí hay un fragmento que muestra el uso de B:

01:0001C788B	ubicación_1C7A8
02:0001C78A	
03:0001C78A ubicación_1C78A	
04:0001C78A LDRB	R7, [R6,R2]
05: ...	
06:0001C7A4 ESTRELLA.W	R7, [R3,#-1]
07:0001C7A8	
08:0001C7A8 ubicación_1C7A8	
09:0001C7A8 MOVIMIENTO	R7, R3
10:0001C7AA AGREGA	R3, n.º 2
11:0001C7AC CMP	R2, R4
12:0001C7AE BLT	ubicación_1C78A

En la línea 1, ves que B se utiliza como un salto incondicional para iniciar un bucle. Puedes ignorar las demás instrucciones por ahora.

BX es Branch and Exchange. Es similar a B en que transfiere el control a un destino, pero tiene la capacidad de cambiar entre el estado ARM/Thumb, y la dirección del destino se almacena en un registro. Las instrucciones de ramificación que terminan con X indican que son capaces de cambiar entre estados. Si el LSB de la dirección del destino es 1, entonces el procesador cambia automáticamente al estado Thumb; de lo contrario, se ejecuta en estado ARM. El formato de instrucción es BX <registro>, donde registro contiene la dirección de destino. Los dos usos más comunes de esta instrucción son regresar de una función mediante la ramificación a LR (es decir, BX LR) y transferir el control al código en un modo diferente (es decir, pasar de ARM a Thumb o viceversa). En el código compilado, casi siempre verá BX LR al final de las funciones; es básicamente lo mismo que RET en x86.

BL es Branch with Link (rama con enlace). Es similar a B , excepto que también almacena la dirección de retorno en LR antes de transferir el control al desplazamiento de destino. Esta es probablemente la equivalencia más cercana a la instrucción CALL en x86 y la verás a menudo utilizada.

para invocar funciones. El formato de instrucción es el mismo que el de B (es decir, solo acepta desplazamientos). A continuación, se incluye un breve fragmento que muestra la invocación y el retorno de una función:

01:00014350 BL	foo; LR = 0x00014354
02:00014354 MOVIMIENTOS	R4, #0x15
03: ...	
04:0001B224 foo	
05:0001B224 EMPUJE	(R1-R3)
06:0001B226 MOVIMIENTO	R3, 0x61240
07: ...	
08:0001B24C BX	LR; volver a 0x00014354

La línea 1 llama a la función foo usando BL; antes de transferir el control al destino, BL almacena la dirección de retorno (0x00014354) en LR. foo hace algún trabajo y regresa al llamador (BX LR).

BLX es Branch with Link and Exchange. Es como BL con la opción de cambiar de estado. La principal diferencia es que BLX puede tomar un registro o un desplazamiento como su destino de ramificación; en el caso en que BLX utiliza un desplazamiento, el procesador siempre intercambia el estado (ARM a Thumb y viceversa). Debido a que comparte las mismas características que BL, también puede pensar en él como el equivalente de la instrucción CALL en x86. En la práctica, tanto BL como BLX se utilizan para llamar a funciones. BL se utiliza normalmente si la función está dentro de un rango de 32 MB, y BLX se utiliza siempre que el rango de destino sea indeterminado (como un puntero de función). Cuando se opera en estado Thumb, BLX se utiliza normalmente para llamar a rutinas de biblioteca; en estado ARM, se utiliza BL en su lugar.

Después de haber explorado todas las instrucciones relacionadas con la ramificación incondicional y la invocación directa de funciones, y cómo regresar desde una función (BX LR), puede consolidar su conocimiento observando una rutina completa:

01:0100C388	; void * __cdecl misterio(int)
02:0100C388	misterio
03: 0100C388 2D E9 30 48 EMPUJE.W [R4,R5,R11,LR]	
04: 0100C38C 0D F2 08 0B AGREGAR R11, SP, #8	
05:0100C3900C4B	LDR R3, =__imp_malloc
06:0100C392 C5 1D	AGREGA R5, R0, n.º 7
07:0100C394 6F F3 02 05 BFC.W R5, n.º 0, n.º 3	
08:0100C3981B68	LDR R3, [R3]
09:0100C39A 15 F1 08 00 AGREGA.W R0, R5, #8	
10:0100C39E 98 47	BLX R3
11:0100C3A0 04 46	Movimiento R4, R0
12:0100C3A224B1	ZBCC R4, ubicación_100C3AE
13:0100C3A4 EB17	----- R3, R5, #0x1F
14:0100C3A6 63 60	STR R3, [R4,#4]
15:0100C3A82560	STR R5, [R4]
16:0100C3AA 08 34	AGREGA R4, n.º 8
17:0100C3AC 04 E0	B ubicación_100C3B8
18:0100C3AE	ubicación_100C3AE
19:0100C3AE 04 49	LDR R1, =aFailed ; "falló..."
20:0100C3B02A46	Movimiento R2, R5
21:0100C3B2 07 20	Movimientos R0, n.º 7

60 Capítulo 2 ■ BRAZO

```

22:0100C3B4 01 F0 14 FC BL           comida
23:0100C3B8
24:0100C3B8           ubicación_100C3B8
25:0100C3B82046          Movimiento   R0, R4
26:0100C3BA BD E8 30 88 POBLACIÓN [R4,R5,R11,PC]
27:0100C3BA             ;Misterio del fin de la función

```

Esta función cubre varias de las ideas discutidas anteriormente (ignore las otras instrucciones por ahora):

- La línea 3 es el prólogo, utilizando la secuencia PUSH {..., LR} ; L26 es la epílogo.
- La línea 10 llama a malloc a través de BLX.
- La línea 22 llama a foo a través de BL.
- La línea 26 regresa, utilizando la secuencia POP {..., PC} .

Operaciones aritméticas

Después de cargar un valor de la memoria en un registro, el código puede moverlo y realizar operaciones sobre él. La operación más sencilla es moverlo a otro registro con la instrucción MOV . La fuente puede ser una constante, un registro o algo procesado por el desplazador de barril. A continuación se muestran algunos ejemplos de su uso:

```

01: 4F F0 0A 00 MOV.W R0, #0xA; r0 = 0xa
02: 38 46           Movimiento   R0, R7      ;r0 = r7
03: A4 4A A0 E1 MOV           R4, R4, LSR n° 21; r4 = (r4>>21)

```

La línea 3 muestra el operando de origen que está siendo procesado por el desplazador de barril antes de ser movido al destino. Las operaciones del desplazador de barril incluyen desplazamiento a la izquierda (LSL), desplazamiento a la derecha (LSR, ASR) y rotación (ROR, RRX). El desplazador de barril es útil porque permite que la instrucción trabaje con constantes que normalmente no se pueden codificar en forma inmediata. Las instrucciones ARM y Thumb pueden tener 16 o 32 bits de ancho, por lo que no pueden tener directamente constantes de 32 bits como parámetro; con el desplazador de barril, un inmediato se puede transformar en un valor mayor y mover a otro registro. Otra forma de mover una constante de 32 bits a un registro es dividir la constante en dos mitades de 16 bits y moverlas una a la vez; esto normalmente se hace con las instrucciones MOVT y MOVT . MOVT establece los 16 bits superiores de un registro y MOVW establece los 16 bits inferiores.

Las operaciones aritméticas y lógicas básicas son ADD, SUB, MUL, AND, ORR y EOR. A continuación se muestran algunos ejemplos de su uso:

```

01: 4B44           AGREGAR        R3, R9          ;r3 = r3+r9
02: 0D F2 08 0B AGREGAR        R11, SP, n.º 8    ; r11 = sp+8
03: 04 EB 80 00 AGREGAR        R0, R4, R0,LSL#2; r0 = r4 + (r0<<2)
04: EA B0           SUB           SP, SP, #0x1A8; sp = sp-0x1a8

```

05:03 FB 05 F2 MUL.W	R2, R3, R5	; r2 = r3 * r5 (resultado de 32 bits)
06:14 F0 07 02 ANDS.W R2, R4, #7		; r2 = r4 y 7 (bandera)
07:83 EA C1 03 EOR.O	R3, R3, R1,LSL#3; r3 = r3 ^ r1	(r1 << 3)
08:53 40 EORS	R3, R2; r3 = r2 (bandera)	
09:43 EA 02 23 ORR.O	R3, R3, R2,LSL#8; r3 = r3 (r2 << 8)	
10:53 F0 02 03 ORRS.W R3, R3, #2		; r3 = r3 2 (bandera)
11:13 43 ORRS	R3, R2	; r3 = r3 r2 (bandera)

Tenga en cuenta la “S” después de algunas de estas instrucciones. A diferencia de x86, las instrucciones aritméticas ARM no establecen el indicador condicional de forma predeterminada. El sufijo “S” indica que la instrucción debe establecer indicadores condicionales aritméticos (cero, negativo, etc.) según su resultado. Tenga en cuenta que la instrucción MUL trunca el resultado de modo que solo los 32 bits inferiores se almacenan en el registro de destino; para la multiplicación completa de 64 bits , utilice las instrucciones SMULL y UMULL (consulte ARM TRM para obtener más detalles).

¿Dónde está la instrucción de división? ARM no tiene una instrucción de división nativa.

(Los núcleos ARMv7-R y ARMv7-M tienen SDIV y UDIV, pero no se tratan aquí). En la práctica, el entorno de ejecución tendrá una implementación de software para la división y el código simplemente lo llamará cuando sea necesario. Aquí hay un ejemplo con El entorno de ejecución de Windows C:

01:4146	Movimiento	R1, R8
02:30 46	Movimiento	R0, R6
03:35 F0 9E FF BL		__rt_udiv ; implementación de software de udiv

Ramificación y ejecución condicional

Todos los ejemplos analizados hasta ahora se han ejecutado de manera lineal. La mayoría de los programas tendrán condicionales y bucles. En el nivel de ensamblaje, estas construcciones se implementan utilizando indicadores condicionales, que se almacenan en el registro de estado del programa de aplicación (APSR). El APSR es un alias del CPSR y es similar al EFLAG en x86. La Figura 2-5 ilustra los indicadores relevantes, que se describen a continuación:

- N (bandera negativa): se establece cuando el resultado de una operación es negativo (el bit más significativo del resultado es 1).
- Z (bandera cero): se establece cuando el resultado de una operación es cero.
- C (Bandera de acarreo): se establece cuando el resultado de una operación entre dos Los valores sin signo se desbordan.
- V (bandera de desbordamiento): se establece cuando el resultado de una operación entre dos Los valores firmados se desbordan.
- IT (bits If-then): codifican varias condiciones para la instrucción Thumb IT . Se analizan más adelante.

Los bits N, Z, C y V son idénticos a los bits SF, ZF, CF y OF en el EFLAG

Se registran en x86. Se utilizan para implementar condicionales y bucles en lenguajes de nivel superior; también se utilizan para respaldar la ejecución condicional en el

Nivel de instrucción. La igualdad se describe en términos de estos indicadores. La Tabla 2-1 muestra las relaciones comunes y los indicadores correspondientes.

CPSR	banderas del cond.	ÉL	mi	yo	reservado
APSR	Código de resultado de Nuevo Zustand		Reservado		0

Figura 2-5

Tabla 2-1: Código condicional y significado

SIGNIFICADO DEL SUFIXO/CÓDIGO	BANDERAS
Igual	Z==1
No es igual	Z==0
MI	N==1
ES	N==0
HOLA	C==1 y Z==0
LS	C==0 o Z==1
EG	N==V
ES	N!=V
GTA	Z==0 y N==V
EL	Z==1 o N!=V

Las instrucciones se pueden ejecutar condicionalmente agregando uno de estos sufijos al final. Por ejemplo, BLT significa bifurcar si la condición LT es verdadera. (Esto es lo mismo que JL en x86). De manera predeterminada, las instrucciones no actualizan los indicadores condicionales a menos que se use el sufijo "S"; las instrucciones de comparación (CBZ, CMP, TST, CMN y TEQ) actualizan los indicadores automáticamente porque generalmente se usan antes de las instrucciones de bifurcación.

La instrucción de comparación más común es probablemente CMP. Su sintaxis es CMP Rn, X, donde Rn es un registro y X puede ser una operación inmediata, de registro o de desplazamiento de barril. Su semántica es idéntica a la de x86: realiza Rn - X, establece los indicadores apropiados y descarta el resultado. Suele ir seguida de una bifurcación condicional. A continuación se muestra un ejemplo de su uso y pseudocódigo:

BRAZO		
01: B3 EB E7 7F CMP.W	R3, R7, ASR n.º 31	
02: 05 db	BLT	Sin loc
03: 01 DC	BGT	loc_mayor
04: BD 42	CMP	R5, R7
05: 02 D9	BLS	Sin loc

```

06:           loc_mayor
07:07 3D       Subs      R5, n.o 7
08:6E F1 00 0E SBC.W    LR, LR, #0
09:           Sin loc
10:A5 FB 08 12 UMULL.W R1, R2, R5, R8
11:87 FB 08 04 SMULL.W R0, R4, R7, R8
12:0E FB 08 23 MLA.W     R3, izq., R8, R2

```

Pseudo C

```

if (r3 < r7) { ir a loc_less; }
de lo contrario si (r3 > r7) { ir a loc_greater; }
de lo contrario si (r3 == r7) { goto loc_equal; }

```

La siguiente instrucción de comparación más común es TST; su sintaxis es idéntica a la de CMP. Su semántica es idéntica a TEST en x86: realiza Rn & X, establece los indicadores apropiados y descarta el resultado. Se utiliza generalmente para comprobar si un valor es igual a otro o para comprobar si hay indicadores. Como la mayoría de las instrucciones de comparación, suele ir seguida de una bifurcación condicional. A continuación se muestra un ejemplo:

```

01: AB8A        LDH       R3, [R5,#0x14]
02:13 F0 02 0F TST.W   R3, n.o 2
03:09 D0        resultado   loc_10179DA
04: ...
05:           ubicación_10179BE
06: AA8A        LDH       R2, [R5,#0x14]
07:12 F0 04 0F TST.W   R2, n.o 4
08:02 D0        resultado   ubicación_10179E8

```

En el estado Thumb-2, hay dos instrucciones de comparación populares: CBZ y CBNZ. Su sintaxis es simple: CBZ/CBNZ Rn, etiqueta, donde Rn es un registro y etiqueta es un desplazamiento al que se debe realizar la bifurcación si la condición es verdadera. CBZ luego se bifurca a etiqueta Si el registro es cero, CBNZ es igual, excepto que verifica si hay una condición distinta de cero. Estas instrucciones se utilizan normalmente para determinar si un número es 0 o si un puntero es NULL. A continuación, se muestra un uso típico:

```

BRAZO
01:10 F0 48 FF BL          comida
                           ; foo devuelve un puntero en r0
02:28 B1       ZBCC      R0, ubicación_100BC8E
03: ...
04:           ubicación_100BC8E
05:01 20       Movimientos   R0, n.o 1
06:28 E0       B          locreto_100 a. C.4
07: ...
08:           locreto_100 a. C.4
09: BD E8 F8 89 POP.W     {R3-R8,R11,PC}

```

Pseudo C

```

tipo *a;
a = foo(...);
si (a == NULL) { devolver 1; }

```

Las otras instrucciones de comparación son CMN/TEQ, que realiza la suma/O exclusivo en los operandos. Como no se utilizan comúnmente, no se tratan aquí.

Has visto que la instrucción de bifurcación (B) puede ejecutar bifurcaciones condicionales añadiendo un sufijo (BEQ, BLE, BLT, BLS, etc.). De hecho, la mayoría de las instrucciones ARM pueden ejecutarse condicionalmente de la misma manera. Si no se cumple la condición, la instrucción puede considerarse una operación no válida. La ejecución condicional a nivel de instrucción puede reducir las bifurcaciones, lo que puede acelerar el tiempo de ejecución. A continuación se muestra un ejemplo:

```
BRAZO
01:00 00 50 E3 CMP           R0, #0
02:01 00 A0 03 MOVEQ R0, #1
03:68 00 D0 15 LDRNEB R0, [R0,#0x68]
04: 1E FF 2F E1 BX          LR
```

Pseudo C

```
tipo desconocido *a = ...;
si (a == NULL) { devolver 1; }
de lo contrario { devolver a->off_48; }
```

Sabes inmediatamente que R0 es un puntero debido a la instrucción LDR en la línea 3. La línea 1 verifica si R0 es NULL. Si es verdadero (EQ), entonces la línea 2 establece R0 en 1; de lo contrario, NEQ carga el valor en R0+0x68 en R0 (línea 3) y luego retorna. Como EQ y NEQ no pueden ser verdaderos al mismo tiempo, solo se ejecutará una de las instrucciones. Tenga en cuenta que no hay instrucciones de bifurcación.

Estado del pulgar

A diferencia de la mayoría de las instrucciones ARM, las instrucciones Thumb no se pueden ejecutar condicionalmente (con excepción de B) sin la instrucción IT (si-entonces). Esta es una instrucción específica de Thumb-2 que permite ejecutar condicionalmente hasta cuatro instrucciones posteriores. La sintaxis general es la siguiente: ITxyz cc, donde cc es el código condicional para la primera instrucción; x, y y z describen la condición para la segunda, tercera y cuarta instrucción, respectivamente. Las condiciones para las instrucciones posteriores a la primera se describen con una de dos letras: T o E. T significa que la condición debe coincidir con cc para ejecutarse; E significa ejecutarse solo si la condición es la inversa de cc. Considere el siguiente ejemplo:

```
BRAZO
01:00 2B             CMP      R3, #0
; comprobar y establecer condición
02:12 BF             ITEEE NE
; iniciar bloque de TI
03: BC FA 8C F0 CLZNE.W R0, R12
;primera instrucción
```

```

04: B6 FA 86 F0 CLZEQ.W R0, R6
      ; segunda instrucción
05:20 30          AGREGAR DEQ R0, #0x20
      ;tercera instrucción

Pseudo C

si (R3 != 0) {
    R0 = contarceros(R12);
} demás {
    R0 = contarceros(R6);
    R0 += 0x20
}

```

La línea 1 realiza una comparación y establece un indicador condicional. La línea 2 especifica las condiciones e inicia el bloque if-then. NE es la condición de ejecución para la primera instrucción; la primera E (después de IT) indica que la condición de ejecución para la segunda instrucción es la inversa de la primera. (EQ es la inversa de NE). La segunda E indica lo mismo para la tercera instrucción. Las líneas 3 a 5 son instrucciones dentro del bloque IT .

Debido a su flexibilidad, la instrucción de TI se puede utilizar para reducir el número de instrucciones necesarias para implementar condicionales cortos en el estado Thumb.

Caja del interruptor

Las sentencias switch-case pueden entenderse como muchas sentencias if-else agrupadas . Debido a que la expresión de prueba y la etiqueta de destino se conocen en tiempo de compilación, los compiladores generalmente construyen una tabla de saltos para almacenar direcciones (ARM) o desplazamientos (Thumb) para cada controlador de caso. Después de determinar el índice en la tabla de saltos, el compilador se ramifica indirectamente al destino cargando la dirección de destino en PC. En el estado ARM, esto normalmente lo hace LDR con PC como destino y registro base. Considere el siguiente ejemplo:

```

01: ; R1 es el caso
02:0B 00 51 E3 CMP          R1, #0xB; ¿está dentro del alcance?
03:01 F1 9F 97 LDRLS        PC, [PC,R1,LSL#2] ; si, cambiar por
                                         ; indexando en la tabla
04:14 00 00 EA B           loc_DD10 ; no, romper
05: 3C DD 00 00+ DCD loc_DD3C ; inicio de la tabla de saltos
06:4C DD 00 00+ DCD ubicación_DD4C
07:68 DD 00 00+ DCD ubicación_DD68
08:8C DD 00 00+ DCD ubicación_DD8C
09: BC DD 00 00+ DCD ubicación_DDBC
10: F0 DD 00 00+ DCD ubicación_DDF0
11:38 DE 00 00+ DCD loc_DE38
12:38 DE 00 00+ DCD loc_DE38
13: EC DC 00 00+ DCD loc_DCEC ; caso/índice 8
14: EC DC 00 00+ DCD loc_DCEC ; caso/índice 9
15:3C DD 00 00+ DCD ubicación_DD3C

```

66 Capítulo 2 ■ ARM

```

16:3C DD 00 00 DCD ubicación_DD3C
17:           loc_DCEC ; manejador para el caso 8,9
18:00 00 A0 E3 MOVIMIENTO      R0, #0
19:08 10 41 E2 SUB            R1, R1, #8
20:04 30 A0 E3 MOVIMIENTO      R3, n.º 4
21:14 00 82 E5 STR           R0, [R2,#0x14]
22: BC 31 C2 E1 STR           R3, [R2,#0x1C]
23:101082E5ESTRATE          R1, [R2,#0x10]

```

La línea 2 verifica si el caso está dentro del rango; si no, entonces ejecuta el controlador predeterminado (línea 4). La línea 3 se ejecuta condicionalmente si R1 está dentro del rango; se ramifica al controlador de caso indexando en la tabla de saltos y carga la dirección de destino en PC. Recuerde que PC está 8 bytes después de la instrucción actual (en estado ARM), por lo que la tabla de saltos generalmente se almacena a 8 bytes de la instrucción LDR .

En el modo Thumb, se aplica el mismo concepto, excepto que la tabla de saltos contiene desplazamientos en lugar de direcciones. ARM agregó nuevas instrucciones para admitir la ramificación de tablas con desplazamientos de bytes o medias palabras: TBB y TBH. Para TBB, las entradas de la tabla son valores de bytes; para TBH, son medias palabras. Las entradas de la tabla se deben multiplicar por dos y agregar a PC para obtener el destino de la ramificación final. Aquí está el ejemplo anterior utilizando TBB:

01:0101E6000B29	CMP	R1, #0xB; ¿está dentro del alcance?
02:0101E60276D8	BHI	loc_101E6F2 ; no, romper
03:0101E604 04 26	Movimientos	R6, n.º 4
04: 0101E606 DF E8 01 F0 TBB W [PC,R1] ; rama utilizando desplazamiento de tabla		
05: 0101E60A 06 jpt_101E606	DCB 6 ; inicio de la tabla de saltos	
06:0101E60B 09	DCB9	
07:0101E60C 0F	DCB-0xF	
08:0101E60D 18	DCB0x18	
09:0101E60E 24	DCB0x24	
10:0101E60F 32	DCB0x32	
11:0101E610 45	Código de barras 0x45	
12:0101E611 45	Código de barras 0x45	
13:0101E612 6D	DCB 0x6D; desplazamiento para 8	
14:0101E613 6D	DCB 0x6D; desplazamiento para 9	
15:0101E614 06	DCB6	
16:0101E615 06	DCB6	
17: ...		
18:0101E6E4	loc_101E6E4 ; manejador para el caso 8,9	
19:0101E6E4 B1 F1 08 03 SUBS.W R3, R1, #8		
20:0101E6E8 00 20	Movimientos	R0, #0
21:0101E6EA 60 61	STR	R0, [R4,#0x14]

Debido a que se encuentra en estado Thumb, PC se encuentra 4 bytes después de la instrucción actual; por lo tanto, para el caso 8, la entrada de la tabla estaría en la dirección 0x0101E612 (=0x0101E60A+8), que es 0x6d, y el controlador está en 0x101E6E4 (=PC+(0x6d*2)). De manera similar al ejemplo anterior, la tabla de saltos generalmente se ubica después de la instrucción TBB/TBH . Tenga en cuenta que TBB/TBH se utilizan solo en estado Thumb.

Misceláneas

En esta sección se analizan brevemente conceptos que no están directamente relacionados con el proceso de ingeniería inversa. Sin embargo, en la práctica, es importante conocerlos porque pueden contribuir a su conocimiento general. Siempre es bueno tener más conocimientos. Puede omitir esta sección en una primera lectura.

Código automodificable y justo a tiempo

ARM admite el concepto de código justo a tiempo (JIT) y código automodificable (SMC). El código JIT es código nativo generado dinámicamente por un compilador JIT; por ejemplo, los lenguajes Microsoft .NET se compilan en un lenguaje intermedio (MSIL) que se convierte en código de máquina nativo (x86, x64, ARM, etc.) para su ejecución en el núcleo de la CPU. El SMC es el código que se genera o modifica mediante el flujo de instrucciones actual. Un ejemplo común de SMC es el código shell codificado que se decodifica y se ejecuta en tiempo de ejecución. Tanto el código JIT como el SMC requieren escribir en la memoria datos nuevos que luego se recuperan mediante la ejecución.

El núcleo ARM tiene dos líneas de caché independientes para instrucciones (caché i) y datos (caché d); las instrucciones se ejecutan desde el caché i y el acceso a la memoria se realiza a través del caché d. No se garantiza que estas líneas de caché sean coherentes, lo que significa que los datos escritos en una caché pueden no ser visibles inmediatamente para la otra. Por ejemplo, supongamos que el i-cache contiene cuatro instrucciones del flujo de instrucciones y el usuario genera instrucciones nuevas o modificadas en el mismo lugar (lo que actualiza el d-cache). Como no son coherentes, el i-cache puede no saber nada de la modificación reciente, por lo que ejecuta instrucciones obsoletas (lo que puede provocar fallos misteriosos o resultados incorrectos). Si está escribiendo sistemas JIT o shellcode, claramente no es una situación deseable. La solución es forzar explícitamente la actualización del i-cache (también conocido como vaciar el caché). En ARM, esto se hace actualizando un registro en el coprocesador de control del sistema (CP15):

01:4F F0 00 00 MOVIMIENTO.W	R0, #0
02:07 EE 15 0F MCR	p15, 0, R0,c7,c5, 0

La mayoría de los sistemas operativos proporcionan una interfaz para esta operación, por lo que no es necesario que la escriba usted mismo. En Linux, utilice `__clear_cache`; en Windows, utilice `Limpia caché de instrucciones`.

Primitivas de sincronización

ARM no tiene una instrucción similar a `cmpxchg` (comparar e intercambiar) en x86; en su lugar, se utilizan dos instrucciones: LDREX y STREX. Estas instrucciones son iguales que LDR/STR, excepto que adquieren acceso exclusivo a la memoria.

68 Capítulo 2 ■ BRAZO

dirección antes de cargar o almacenar. Juntos, se utilizan normalmente para implementar funciones intrínsecas de comparación e intercambio. Por ejemplo:

```
BRAZO

01:01 21           Movimientos      R1, n.º 1
02:                 ubicación_100C4B0
03:54 E8 00 2F LDREX.W R2, [R4]
04:1A B9           CBNZ          R2, ubicación_100C4BE
05: 44 E8 00 13 STREX.W R3, R1, [R4] ; r3 es el resultado
06:00 2B           CMP           R3, #
07: F8 D1           BNE          ubicación_100C4B0
```

Pseudo C

```
si (InterlockedCompareExchange(&r4, 1, 0) == 0) { hacer cosas; }
```

La línea 3 realiza una carga atómica en R2 y la compara con 0; si es cero, se intercambia con cero y el resultado se devuelve en R3. Esta es, en realidad, la implementación de InterlockedCompareExchange en Windows.

De vez en cuando, se encontrará con código que utiliza las instrucciones DMB, DSB e ISB . Se trata de instrucciones de barrera que garantizan que el acceso a la memoria y la obtención de instrucciones estén sincronizados antes de ejecutar las instrucciones posteriores. Esto es necesario en algunos casos porque el acceso a la memoria y las instrucciones pueden ejecutarse fuera de orden (es decir, la CPU puede ejecutar las instrucciones en un orden diferente al que aparece en el código ensamblador) y otros subprocesos en ejecución pueden no ver el resultado actualizado y, en consecuencia, tener una visión inconsistente de los datos. Por este motivo, a menudo verá que estas instrucciones se utilizan en código que implementa bloqueos.

Servicios y mecanismos del sistema

Cuando se inicia un núcleo ARM, comienza a ejecutar código en el estado ARM en la dirección de memoria 0x00000000 o 0xFFFF0000, dependiendo de una configuración en el coprocesador 15. Esto se determina por el bit de vector (V) en el registro de control del sistema (CP15, C1/C0). Si es 0, entonces el vector de excepción está en 0x00000000; de lo contrario, está en 0xFFFF0000. Esta dirección generalmente está en la memoria flash (la RAM aún no se ha inicializado, por lo que no se puede utilizar), y el contenido de la misma se conoce comúnmente como vectores de excepción. ARM tiene una lista de vectores predefinidos que comienzan en la dirección base. El controlador de excepción RESET es el primero en la tabla, por lo que se ejecuta después de un evento de reinicio. Debido a que es el primer código que se ejecuta, generalmente comienza realizando una configuración básica del hardware e inicia el proceso de arranque. Aquí hay un vector de excepción tomado de un dispositivo real:

```
01: 00000000 1A 00 00 EA B vect_RESET
02: 00000004 12 00 00 EA B vect_INSTRUCCIÓN_INDEFINIDA
```

```
03: 00000008 12 00 00 EA B vect_SUPERVISOR_CALL ; (para SWI/SVC)
04: 0000000C 12 00 00 EA B vect_PREFETCHABORT
05: ...
06: 00000004           vect_INSTRUCCIÓN_INDEFINIDA
07: 00000054 FE FF FF EA B vect_INSTRUCCIÓN_NO_DEFINIDA
08: 00000058           vect_LLAMADA_SUPERVISOR
09: 00000058 FE FF FF EA B vect_LLAMADA_SUPERVISOR
10: 0000005C vect_PREFETCHABORT
11: 0000005C FE FF FF EA B vect_PREFETCHABORT
12: ...
13: 00000070           vector_REINICIO
14: 00000070 1C F1 9F E5 LDR PC, =0x10000078
15:           ; el código ha sido mapeado en 0x10000078
16:           ;comenzar a ejecutar allí
17: ...
18: 10000078 18 01 9F E5 LDR R0, =0x2001
19: 1000007C 11 0F 0F EE MCR p15, 0, R0,c15,c1, 0
20:           ; inicializa un registro específico del proveedor
21: 10000080 00 00 A0 E1 NOP
22: 10000084 00 00 A0 E1 NOP
23: 10000088 00 00 A0 E1 NOP
24: 1000008C 78 00 A0 E3 MOV R0, #0x78
25: 10000090 10 0F 01 EE MCR p15, 0, R0,c1,c0, 0
26:           ; inicializa el registro de control del sistema
```

Después de inicializar el hardware, el código de excepción de reinicio salta a un cargador de arranque que normalmente se encuentra en la memoria flash, un medio extraíble (MMC, tarjeta SD, etc.) o alguna otra forma de almacenamiento. Algunos dispositivos utilizan U-Boot, un cargador de arranque de código abierto muy popular . El cargador de arranque realiza más inicialización del hardware, lee una imagen del sistema operativo desde el almacenamiento y la asigna a la memoria principal, y transfiere el control allí. Después de eso, el sistema operativo se inicia y el sistema está listo para su uso.

Un sistema operativo administra los recursos de hardware y proporciona servicios a los usuarios. Debido a que el código de usuario (normalmente en modo USR) se ejecuta con un privilegio menor que el código del núcleo/SO (normalmente en modo SVC), tiene que utilizar una interfaz para solicitar un servicio del SO. En la práctica, la interfaz se proporciona a través de una interrupción de software o una instrucción trap especial proporcionada por el procesador; el servicio se implementa habitualmente como llamadas al sistema. (Por ejemplo, en Linux x86, puede utilizar la interrupción 0x80 o la instrucción especial SYSENTER para emitir una llamada al sistema; en x64, esto se proporciona mediante la instrucción SYSCALL). En ARM, no hay una instrucción de llamada al sistema dedicada, por lo que se utiliza la interrupción de software para implementar llamadas al sistema. Cuando se produce una interrupción de software, el procesador cambia al modo supervisor para gestionar la interrupción. Las interrupciones de software pueden ser activadas por el SWI/SVC Instrucciones. (Estas instrucciones son idénticas excepto que tienen nombres diferentes).

70 Capítulo 2 ■ BRAZO

Ambas instrucciones toman un número inmediato como parámetro: algunos sistemas operativos usan este parámetro como índice en una tabla de llamadas del sistema, y otros no usan el parámetro pero requieren que el número de llamada del sistema esté en un registro (por ejemplo, Windows usa R12 para este propósito). En algunos sistemas Linux, el número de llamada al sistema se coloca en R7 y los argumentos se pasan a través de R0-R2. Por ejemplo:

Linux (Ubuntu)

```

01:05 20 A0 E1 MOVIMIENTO      R2, R5      ; 3er argumento
02:06 10 A0 E1 MOVIMIENTO      R1, R6      ; 2do argumento
03:09 00 A0 E1 MOVIMIENTO      R0, R9      ; 1er argumento
04:92 70 A0 E3 MOVIMIENTO      R7, #0x92

; número de llamada al sistema
05:00 00 00 EF SVC            0 ; realizar la llamada al sistema
06:04 00 70 E3 CMN            R0, n.º 4

; comprobar el valor de retorno
07:00 30 A0 13 MOVIMIENTO R3, #0
; movimiento de condición basado en el valor de retorno

```

Ventanas RT

```

ZwCreateFile (en ntdll)
4F F0 53 0C MOVIMIENTO.W      R12, #0x53
01 DF          CVS           1
70 47          BX            LR

; Fin de la función ZwCreateFile

```

El SVC pasa al modo supervisor, copia los registros de usuario relevantes en su propio espacio, realiza la función solicitada y regresa cuando termina. ¿Cómo sabe el SVC a dónde regresar? Normalmente, regresa a la instrucción después del SVC. Antes de procesar la excepción, el modo SVC copia la dirección de retorno a R14_svc, que es un registro bancario en el modo SVC. Los registros bancarios son aquellos que tienen significado solo en el contexto de un modo de procesador particular. Por ejemplo, R13_svc y R14_svc son registros bancarios en el modo SVC, por lo que tendrán valores diferentes a los de R13–14 en el modo USR.

Si bien existe una instrucción dedicada para el punto de interrupción de software BKPT, existen algunas formas de implementarlo. La primera es a través de la instrucción BKPT, que activa el controlador de excepción de cancelación de precarga; el controlador puede luego pasar el control a un depurador. Otro método común es activar el controlador de excepción de instrucción no definida a través de una instrucción no definida. La codificación de instrucciones ARM tiene un rango reservado que se garantiza que no estará definido.

Instrucciones

Cada instrucción en estado ARM codifica una condición aritmética para admitir la ejecución condicional. De manera predeterminada, la condición es AL (ejecutar siempre).

La condición se codifica en los cuatro bits más significativos del código de operación (bits 28 a 31); AL se define como 0b1110, que es 0xE. Si presta mucha atención a los fragmentos de código de ensamblaje (en estado ARM), notará que el código de bytes generalmente tiene un 0xE*. patrón al final. De hecho, si observa las instrucciones en un editor hexadecimal, notará que 0xE* aparece comúnmente cada cuatro bytes. Por ejemplo:

```
FE FF FF EA FE FF FF EA FE FF FF EA FE FF FF EA  
FE FF FF EA 1C F1 9F E5 00 00 A0 E1 18 01 9F E5  
11 0F 0F EE 00 00 A0 E1 00 00 A0 E1 00 00 A0 E1  
78 00 A0 E3 10 0F 01 EE 00 00 A0 E1 00 00 A0 E1  
00 00 A0 E1 00 00 A0 E3 17 0F 08 EE 17 0F 07 EE
```

¿Por qué es importante conocer este patrón? Porque el código ARM a veces está incrustado en la memoria ROM o flash y puede no seguir un formato de archivo específico . En su recorrido de ingeniería inversa, a veces solo recibirá un volcado de memoria sin mucho contexto, por lo que puede ser útil adivinar la arquitectura mirando los códigos de operación. La otra razón está relacionada con los exploits.

El shellcode puede estar incrustado dentro de un exploit distribuido a través de la red o en un documento; para analizarlo, debes extraer el shellcode del resto del tráfico de la red. A veces es sencillo y el límite del shellcode es obvio, otras veces no lo es. Sin embargo, si puedes reconocer el patrón, puedes adivinar rápidamente el inicio/final del código. La capacidad de reconocer los límites de las instrucciones en un conjunto aparentemente aleatorio de datos es importante. Tal vez lo aprecies más adelante.

Tutorial paso a paso

Una vez que haya aprendido todos los conceptos básicos, podrá aplicarlos en esta sección descompilando por completo una función desconocida. Esta función abarca muchos conceptos y técnicas que se tratan en este capítulo, por lo que es una excelente manera de poner a prueba sus conocimientos. A lo largo del camino, también aprenderá nuevas habilidades que solo se insinuaron en las secciones anteriores. Debido a que la función es algo larga, la pusimos en forma de gráfico para ahorrar espacio y mejorar la legibilidad. El cuerpo de la función se muestra en la Figura 2-6, y todos los números de línea de código que se analizan en esta sección hacen referencia a esta figura.

A continuación el contexto en el que se le llama:

01:17 9B	LDR	R3, [SP,#0x5c]
02:16 9A	LDR	R2, [SP,#0x58]
03:51 46	Movimiento	R1, R10
04:20 46	Movimiento	R0, R4
05: FF F7 98 FF BL		función unk

72 Capítulo 2 ■ BRAZO

```

07:           función unk
08: 2D E9 78 48 EMPUJE.W (R3-R6,R11,LR)
09: 0D F2 10 0B AGREGAR R11, SP, #0x10
10:85 68 R5, [R0,#8]      LDR
11:8C 69 R4, [R1,#0x10]    LDR
12: 1E 46 R6, R3          Movimiento
13: A5 42                 CMP
14:01 D0                 R5, R4
                                         ubicación_103C4BE

18: loc_103C4BE
19:03 8A                 LDH
20:02 2B                 CMP
21: FA D1                 BNE
                                         R3, [R0,#0x10]
                                         R3, n.º 2
                                         ubicación_103C4BA

22:8369                 LDR
23:1A 40                 Y
24:C369 25:3340          LDR
26:1343                 Y
27: F4 D1                 ORRS
                                         R3, R6
                                         R3, R2
                                         ubicación_103C4BA

28: C3 68 29: 00          LDR
68 30: 03 EB 43          LDR
02 AGREGAR,W R2, R3, R3,LSL#1
31: CB 68 32: D0          LDR
68 33: 03 EB C2          LDR
03 AGREGAR,W R3, R3, R2, LSL # 3
34:93 F9 16 40 LDRSB,W R4, [R2,#0x16]
35: E9 F7 E9 F9 BL 36: 61 28 CMP 37.
04 D0 BEQ
                                         foo ; supongamos que esto toma el argumento I
                                         R0, #0x61
                                         ubicación_103C4FB

38:6228                 CMP
39:04 D0                 R0, #0x62
                                         Movimiento
                                         ubicación_103C4FA

43:           ubicación_103C4F6
44:81 2C                 CMP
45: DF D1                 BNE
                                         R4, #0x81
                                         ubicación_103C4BA
                                         40:63 2C
                                         41:02 DIA
                                         CMP
                                         BGE
                                         R4, #0x63
                                         ubicación_103C4FA

46:           ubicación_103C4FA
47:01 20                 Movimiento
                                         R0, n.º 1
                                         42:E1E7
                                         B
                                         ubicación_103C4BA

                                         15: loc_103C4BA
                                         16:00 20
                                         17:1E0
                                         Movimiento
                                         B
                                         R0, #
                                         ubicación_103C4FC

48: locrel_103C4FC
49: BD E8 78 68 POBLACIÓN 50;          (R3-R8,R11,PC)
                                         ; Fin de la función unk_function

```

Figura 2-6

Al abordar una función desconocida (o cualquier bloque de código), el primer paso es determinar qué se sabe con certeza sobre ella. La siguiente lista enumera estos hechos y cómo se sabe:

- El código es el estado Thumb y el conjunto de instrucciones es Thumb-2. Lo sabes porque:
 - 1) el prólogo y el epílogo (líneas 1 y 49) usan el comando PUSH/POP.
 - patrón;
 - 2) el tamaño de la instrucción es de 16 o 32 bits de ancho;
 - 3) el desensamblador muestra el prefijo .W para algunas instrucciones, lo que indica que están utilizando la codificación de 32 bits.

- La función conserva R3–R6 y R11. Esto se sabe porque se guardan y restauran en el prólogo (línea 1) y el epílogo (línea 49), respectivamente.
- La función toma como máximo cuatro argumentos (R0–R3) y devuelve un valor booleano (R0). Lo sabes porque según la ABI (interfaz binaria de aplicación) de ARM, los primeros cuatro parámetros se pasan en R0–R3 (el resto se coloca en la pila) y el valor de retorno está en R0. En este caso, son “como máximo cuatro” porque viste que antes de llamar a la función en la línea 5, R0–R3 se inicializan con algunos valores y no se ve ninguna otra instrucción que se escriba en la pila (para argumentos adicionales). En este punto, el prototipo de la función es el siguiente:
`BOOL unk_function(int, int, int, int)`
- El tipo de los dos primeros argumentos es “puntero a un objeto”. Esto se sabe porque R0 y R1 son la dirección base en una instrucción de carga (líneas 10 y 11). Lo más probable es que los tipos sean estructuras porque hay acceso al desplazamiento 0x10, 0x18, 0x1c, etc. (línea 10, 11, 19, 22, 24, 28, etc.). Puede estar casi seguro de que no son matrices porque el patrón de acceso/carga no es secuencial. No se sabe con certeza si R0 y R1 son punteros a uno o dos tipos de estructura diferentes sin más contexto. Por ahora, puede suponer que son dos tipos diferentes. Actualice el prototipo de la siguiente manera:
`BOOL unk_function(estructura1 *, estructura2 *, int, int)`
- loc_103C4BA es la ruta de salida para devolver 0; loc_103C4FA es la ruta de salida para devolver 1; y locret_103C4FC retorna desde la función. Por lo tanto, las ramificaciones a estas ubicaciones indican que ha terminado con la función.
- El tercer y cuarto argumento son de tipo entero. Esto lo sabes porque R2 y R3 se están utilizando en operaciones AND/ORR (líneas 23, 25 y 26). Si bien existe la posibilidad de que sean punteros, es poco probable que sea así a menos que fueran punteros de codificación/decodificación; e incluso si fueran punteros, deberías verlos en uso en operaciones de carga/almacenamiento, pero no lo ves.
- Aunque R11 está ajustado para que esté 0x10 bytes por encima del puntero de pila, nunca se utiliza después de esa instrucción. Por lo tanto, se puede ignorar.
- La función foo (línea 35) toma un argumento. Su cuerpo completo no se incluye aquí debido a limitaciones de espacio. Supongamos que esto es así por simplicidad.

Una vez enumerados los hechos conocidos, ahora debe utilizarlos para derivar lógicamente otros hechos útiles. La siguiente tarea importante es profundizar en las dos estructuras desconocidas identificadas. Obviamente, no puede recuperar su diseño completo porque solo algunos de sus elementos están referenciados en la función; sin embargo, aún puede inferir la información del tipo de campo.

R0 es del tipo struct1 *. En la línea 10, carga un miembro de campo en el desplazamiento 0x8 y luego lo compara con R4 (línea 13). R4 es un miembro de campo en el desplazamiento 0x18 en la estructura struct2 (R1). Debido a que se están comparando entre sí, sabes que

son del mismo tipo. La línea 13 compara estos dos campos. Si son iguales, la ejecución continúa hasta loc_103C4BE; de lo contrario, se devuelve 0 (línea 15). Debido a la comparación de igualdad, puede inferir que estos dos campos son números enteros.

La línea 19 carga otro miembro de campo de struct1 y lo compara con 2; si no es igual, se devuelve 0 (línea 21). Puede inferir que el tipo de campo es corto debido a la instrucción LDRH (carga media palabra).

Las líneas 22 y 23 cargan otro miembro de campo de struct1 y lo combinan con el tercer argumento (que se supone que es un entero). Las líneas 25 y 27 hacen algo similar con el cuarto argumento. Debido a estas operaciones, se puede inferir que los miembros de campo en los desplazamientos 0x18 y 0x1c son enteros.

Las definiciones de estructura hasta el momento son las siguientes:

```
estructura1
...
+0x008 campo08_i ; mismo tipo que struct2.field18_i
...
+0x010 campo10_s ; corto
...
+0x018 campo18_i ; entero
+0x01c campo1c_i ; entero

estructura2
...
+0x018 field18_i ; mismo tipo que struct1.field08_i
```

NOTA: Para los nombres de campos de estructura, puede seguir la costumbre de indicar el desplazamiento y el “tipo”. Por ejemplo, un sufijo “l” significa entero (o algún tipo genérico de 32 bits), “s” significa corto (16 bits), “c” significa carácter (1 byte) y “p” significa puntero de algún tipo. Esto le permite recordar rápidamente cuáles son sus tipos. Cuando determine su verdadero propósito, puede cambiarles el nombre por algo más significativo.

Dados estos tipos, ya puedes recuperar el pseudocódigo de todo.
De la línea 1 a la 27. Es como sigue:

```
estructura1 *arg1 = ...;
estructura1 *arg2 = ...;
int arg3 = ...;
int arg4 = ...;

BOOL resultado = unk_function(arg1, arg2, arg3, arg4);
si (arg1->campo08_i == arg2->campo18_i) {
    si (arg1->campo10_s != 2) devuelve 0;
    si ( ((arg1->campo18_i & arg3) |
          (arg1->campo1c_i & arg4))
        ) != 0
    ) devuelve 0;
    ...
} demás {
```

```
    devuelve 0;  
}
```

NOTA: Puede resultar un poco sospechoso que se esté utilizando la operación AND en dos campos enteros adyacentes. Esto suele significar que en realidad son enteros de 64 bits divididos en dos registros o ubicaciones de memoria. Este es un patrón común que se utiliza para acceder a constantes de 64 bits en arquitecturas de 32 bits.

Los lectores astutos notarán que las líneas 25 a 27 pueden parecer un poco redundantes. ANDS establece los indicadores de condición, ORRS los sobrescribe inmediatamente y BNE toma el indicador de ORRS; por lo tanto, las condiciones establecidas por ANDS realmente no son necesarias. El compilador genera esta redundancia porque está optimizando para la densidad del código: AND tendrá 4 bytes de longitud, pero ANDS solo tiene 2 bytes. MOV y MOVS también están sujetos a la misma optimización. A menudo verá este patrón en el código optimizado para Thumb.

La línea 28 carga otro campo desde struct1 en R3; la línea 29 carga desde el desplazamiento cero de la misma estructura en R0; y la línea 30 establece R2 en R3*3 (=R3+(R3<<1)).

La línea 31 carga un campo de struct2 en R3 y luego accede a otro campo usándolo como un puntero base. Esto implica que tienes un puntero a otra estructura dentro de struct2 en el desplazamiento 0xC. La línea 32 carga un campo de esa nueva estructura en R3; la línea 33 lo actualiza para que sea R3+R2*8; y la línea 34 lo usa como una dirección base y carga un valor corto con signo en el desplazamiento 0x16 de otra estructura en R4.

Actualicemos la definición de la estructura antes de continuar:

```
estructura1  
    +0x000 campo00_i ; entero  
    ...  
    +0x008 field08_i ; mismo tipo que struct2.field18_i  
    +0x00c campo0c_i ; entero  
    ...  
    +0x010 campo10_s ; corto  
    ...  
    +0x018 campo18_i ; entero  
    +0x01c campo1c_i ; entero  
    ...  
  
estructura2  
    ...  
    +0x00c campo0c_p ; estructura3 *  
    ...  
    +0x018 field18_i ; mismo tipo que struct1.field08_i  
    ...  
  
estructura3  
    ...  
    +0x00c campo0c_p ; estructura4 *  
    ...  
  
struct4 (tamaño=0x18=24) // ¿por qué?
```

76 Capítulo 2 ■ BRAZO

```
...
+0x016 campo16_c; carácter +0x017 fin
```

Podrías deducir que había una matriz involucrada debido al factor de multiplicación/escalamiento (líneas 30 y 33); no había dos matrices porque R2–R3 en la línea 30 no es una dirección base sino un índice. Además, no tiene sentido que una dirección base se multiplique por 3. La dirección base de la matriz es R3 en la línea 33 porque se está indexando con R2. Infiriste que cada elemento de la matriz debe ser 0x18 (24) porque después de la simplificación, era $R2^*3^*8$, donde R2 es el índice y 24 es la escala.

La figura 2-7 ilustra las relaciones entre las cuatro estructuras.

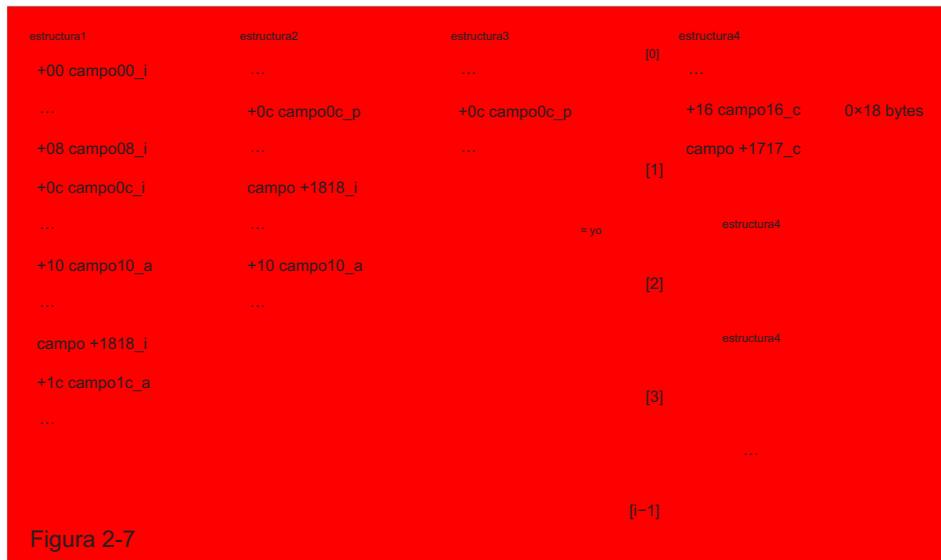


Figura 2-7

Aquí está el pseudocódigo para las líneas 28 a 35:

```
r3 = arg1->campo0c_i; r2 = r3 + r3<<1
= arg1->campo0c_i*3;
r3 = arg2->campo0c_p; r3 = arg2-
>campo0c_p->campo0c_p; r3 = arg2->campo0c_p-
>campo0c_p + r2*8 = arg2->campo0c_p->campo0c_p + arg1-
>campo0c_i*24; = arg2->campo0c_p->campo0c_p[arg1->campo0c_i]; r4 = arg2-
>campo0c_p->campo0c_p[arg1->campo0c_i].campo16_c; r0 = foo(arg1-
>campo00_i);
```

El resto de la función simplemente compara el valor de retorno de foo y r4. El pseudocódigo completo ahora luce así:

```
estructura1 *arg1 = ...;
estructura2 *arg2 = ...;
```

```
int arg3 = ...;
int arg4 = ...;

BOOL resultado = unk_function(arg1, arg2, arg3, arg4);

BOOL función_desconocida(estructura1 *arg1, estructura2 *arg2, entero arg3, entero arg4)
{
    carácter a;
    entero b;

    si (arg1->campo08_i == arg2->campo18_i) {
        si (arg1->campo10_s != 2) devuelve 0;
        si ( ((arg1->campo18_i & arg3) |
              (arg1->campo1c_i y arg4))
            ) != 0
            ) devuelve 0;
        b = foo(arg1->campo00_i);
        a = arg2->campo0c_p->campo0c_p[arg1->campo0c_i].campo16_c;
        si (b == 0x61 y a != 0x61) {
            devuelve 0;
        } de lo contrario { devolver 1;}
        si (b == 0x62 y a >= 0x63) {
            devuelve 1;
        } de lo contrario { devolver 0;}
    } demás {
        devuelve 0;
    }
}
```

Si bien esta función utilizó múltiples estructuras de datos interconectadas cuyo diseño completo no está claro, puede ver cómo pudo recuperar algunos de los tipos de campo y su relación con otros. También aprendió a reconocer el ancho y el carácter signado de un tipo al considerar la instrucción y el código condicional asociados con ellos.

Próximos pasos

Este capítulo proporciona las habilidades fundamentales necesarias para realizar ingeniería inversa estática del código ARM. Evitamos intencionalmente escribir un manual de instrucciones y omitimos muchos detalles; para mejorar sus habilidades, deberá realizar los ejercicios, practicar y leer los manuales de ARM (estas actividades van juntas). El manual de referencia técnica puede ser algo denso, pero el conocimiento adquirido en este capítulo hará que sea mucho más fácil de entender.

El siguiente paso debe ser comprar un dispositivo ARM y experimentar con él. Hay muchos dispositivos ARM entre los que elegir, pero quizás los dos más propicios para el aprendizaje sean BeagleBoard y PandaBoard. Se trata de placas de desarrollo destinadas a introducir a la gente en el desarrollo integrado en ARM.

Plataforma; son relativamente potentes, baratas (\$150–\$170), bien documentadas y tienen una gran comunidad de usuarios. (Puede que no te encuentres con muchas personas que entiendan el ensamblaje ARM, pero no te preocunes porque ya leíste este capítulo). Las áreas para las que puedes necesitar ayuda suelen estar relacionadas con los periféricos integrados y cómo se programan/controlan). Puedes instalar Linux con un entorno de desarrollo completo en estas placas, por lo que es muy sencillo poner a prueba tus conocimientos de ARM.

Ceremonias

Los ejercicios se incluyen para garantizar que comprenda bien los conceptos y aumentar su motivación. Algunos de los ejercicios se seleccionaron intencionalmente para incluir instrucciones que no se cubrieron en el capítulo para que se acostumbre a leer el manual (un hábito muy importante); también se omite el contexto de llamada para que piense más. Cada función es independiente para facilitar la descompilación completa; algunas se seleccionan de modo que pueda verificar su respuesta si ha realizado suficientes. Se recomienda que escriba comentarios y notas, y que establezca conexiones entre ramas/etiquetas, en los propios ejercicios.

Para el código de cada ejercicio, haga lo siguiente en orden (siempre que sea posible):

- Determinar si está en estado Thumb o ARM.
 - Explique la semántica de cada instrucción. Si la instrucción es LDR/STR, explique también el modo de direccionamiento.
 - Identificar los tipos (ancho y signo) de cada objeto posible. Para las estructuras, recuperar el tamaño del campo, el tipo y el nombre descriptivo siempre que sea posible. No todos los campos de estructura se podrán recuperar porque la función solo puede acceder a unos pocos campos. Para cada tipo recuperado, explíquese a sí mismo (o a otra persona) cómo lo dedujo.
 - Recuperar el prototipo de la función.
 - Identificar la función prólogo y epílogo.
 - Explique qué hace la función y luego escriba un pseudocódigo para ella.
 - Descompila la función nuevamente a C y dale un nombre significativo.
1. La figura 2-8 muestra una función que acepta dos argumentos. Puede parecer un poco complicada al principio, pero su funcionalidad es muy común. Tenga paciencia.
 2. La figura 2-9 muestra una función que se encontró en la tabla de exportación.

3. Aquí hay una función sencilla:

```
01:             misterio3
02:8368          LDR      R3, [R0,#8]
03:0B60          STR      R3, [R1]
04:C3 68         LDR      R3, [R0,#0xC]
05:00 20         Movimiento R0, #0
06:4B60          STR      R3, [R1,#4]
07:70 47         BX       LR
08:             ; Fin de la función misterio3
```

4. La figura 2-10 muestra otra función sencilla.
5. La figura 2-11 también es sencilla. Se han eliminado los nombres de las cadenas reales, por lo que no se puede hacer trampa buscando en Internet.
6. La figura 2-12 implica algunos cambios.
7. La figura 2-13 ilustra una rutina común, pero es posible que no la haya visto implementada de esta manera.
8. En la Figura 2-14, byteArray es una matriz de 256 caracteres cuyo contenido es byte-Array[] = {0, 1, ..., 0xff}.
9. ¿Qué hace la función que se muestra en la Figura 2-15?
10. La figura 2-16 es una función de Windows RT. Lea MSDN si es necesario. Ignore las rutinas de cookies de seguridad PUSH/POP .
11. En la Figura 2-17, sub_101651C toma tres argumentos y no devuelve nada.
Si logras completar este ejercicio, deberías darte una palmadita en la espalda.

80 Capítulo 2 ■ ARM

01:	misterio1		
02: F0 01 2D E9 ESTIMULACIÓN		;SPI, (R4-R8)	
03:00 30 D0 E5 LDRB		R3, [R0]	
04:2D 00 53 E3 CMP		R3, #0x2D	
05:29 00 00 0A ecualizador		ubicación_B348	
06:2B 00 53 E3 CMP	R3, #0x2B	57: ubicación_B348	
07:00 60 A0 E3 MOVIMIENTO	R6, #0	58:01 30 F0 E5 LDRB	(R3, [R0,#1])
08: 01 30 F0 05 LDREQB R3, [R0,#1]		59:01 60 A0 E3 MOVIMIENTO	R6, n.º 1
		60: D5 FF FF EA B	ubicación_B2Ac
		61: ; Fin de la función misterio1	
09:	ubicación_B2AC		
10:30 00 53 E3 CMP		R3, #0x30	
11:04 00 00 1A BNE		ubicación_B2C8	
12:01 30 80 E2 AGREGAR		R3, R0, n.º 1	
13:	ubicación_B2B8		
14:03 00 A0 E1 MOVIMIENTO	R0, R3		
15:01 20 D3 E4 LDRB	R2, [R3],#1		
16:30 00 52 E3 CMP	R2, #0x30		
17: FB FF FF 0A ECU		ubicación_B2B8	
18:	ubicación_B2C8		
19:00 C0 A0 E3 MOVIMIENTO		R12, #0	
20:00 40 A0 E3 MOVIMIENTO		R4, #0	
21:00 50 A0 E3 MOVIMIENTO		R5, #0	
22:0A 80A0E3 MOVIMIENTO		R8, #0xA	
23:01 00 00 EA B		ubicación_B2E4	
27: ubicación_B2E4			
28: 0C 70 D0 E7 LDRB 29: 01 c0 8C E2	R7, [R0,R12]		
ANADIR	R12, R12, #1		
30:942883E0UMULL	R2, R3, R4, R8		
31:307057E2 SUBTITULACIONES	R7, R7, #0x30		
32:07 00 00 4A IMC		ubicación_B316	
33:090057 E3 CMP	R7, n.º 9		
34:983523E0 MLA	R3, R8, R5, R3		
35:04 00 00 CA BGT		ubicación_B318	
36:0B 00 5C E3 CMP	R12, #0xB	43: ubicación_B318	
37: F3 FF FF 1A BNE	ubicación_B2DC	44:06 20 54 E0 SUBSRIETE	R2, R4, R6
		45: C6 3F C5 E0 SBC	R3, R5, R6, ASR n.º 31
		46:02 01 52 E3 CMP	R2, #0x80000000
		47:00 00 03 E2 SBCS	R0, R3, #0
		48: F7 FF FF AA BGE	ubicación_B30c
24:	ubicación_B2DC		
25:07 40 92 E0 AGREGA	R4, R2, R7	38:	ubicación_B30C
26: C7 5F A3 E0 CAD	R5, R3, R7, ASR n.º 31	39:00 00 A0 E3 MOVIMIENTO	R0, #0
		49:00 00 56 E3 CMP	R6, #0
		50:01000000A ecualizador	ubicación_B33C
		51:00 40 74 E2 RSBS	R4, R4, #0
		52:00 50 E5 E2 RSC	R5, R5, #0
		53: ubicación_B33C	
		54:00 40 81 E5 STR	R4, [R1]
		55:01 00 A0 E3 MOVIMIENTO	R0, n.º 1
		56: F1 FF FF EA B	ubicación_B310
		40: ubicación_B310	
		41: F0 01 BD E8 LDMPD	[SPI, (R4-R8)]
		42: 1E FF 2F E1 BX	LR

Figura 2-8

```

01:                         misterio2
02:28 B1                  ZBCC          R0, ubicación_C672

03: 90 F8 63 00 LDRB.W R0, [R0,#0x63]
04:00 38 SUBS              R0, #0
05:18 BF                  ES NECESARIO
06:01 20                  MOVERSE      R0, n.o 1
07:70 47                  BX           LR
                                         ; Fin de la función misterio2

```

Figura 2-9

```

01:                         misterio4
02:08 B9                  CBNZ          R0, ubicación_100C3DA

03:00 20                  Movimientos   R0, #0
04:70 47                  BX           LR
                                         ; Fin de la función misterio4

```

Figura 2-10

```

01: misterio5
02:03 46                  Movimiento R3, R0
03:06 2B                  CMP R3, n.o 6
04:00 D0                  Ubicación de la base de datos BEQ_1032596

22: loc_1032596
23:01 48 24:70             LDR R0, =aE; "E"
47:25:                   BX           LR
                                         ; Fin de la función misterio5

07:08 2B                  CMP R3, n.o 8
08:05 D0                  Ubicación de la base de datos BEQ_1032596
                                         ; Fin de la función misterio5

09:09 2B                  CMP R3, n.o 9
10:01 D0                  Ubicación de la base de datos BEQ_103258A
                                         ; Fin de la función misterio5

13: loc_103258A
14:07 48                  LDR R0, =aB; "B"
15:70 47                  BX           LR
                                         ; Fin de la función misterio5

16: loc_103258E
17:05 48                  LDR R0, =ac; "C"
18:70 47                  BX           LR
                                         ; Fin de la función misterio5

19: loc_1032592
20:03 48                  LDR R0, =aD; "D"
21:70 47                  BX           LR
                                         ; Fin de la función misterio5

11:09 48 12:70             LDR R0, =aA; "A"
47                           BX           LR
                                         ; Fin de la función misterio5

```

Figura 2-11

82 Capítulo 2 ■ ARM

```

01- misterio6
02: 2D E9 18 48 EMPUJE W {R3,R4,R11,LR}
03: 0D F2 08 0B AGREGAR 04: 04 R11, SP, n.o 8
08 05: 00 22 LDR R4, [R0]
06: 00 2C 07: Movimientos R2, #0
06 DD CMP R4, #0
BLE ubicación_103B3B6

08: ubicación_103B3A8
09:50 F8 04 3F LDR.O ;R3, [R0,#4]!
10:8B42 CMP R3, R1
11:06 D0 Movimientos ubicación_103B3BE

20: ubicación_103B3BE
21: B2 F1 20 03 SUBS.W R3, R2, #0X20
22:01 21 MOVIMIENTOS R1, R3 R1, n.o 1
23:99 40 LLS 12:01 32 AGREGA R2, n.o 1
24:01 23 Movimientos R3, n.o 1 13: A2 42 CMP R2, R4
25: 13 FA 02 F0 LSLS.W R0, R3, R2 14: F8 DB BLT ubicación_103B3A8
26: F5 E7 B, Fin de la función misterio6 locret_103B3BA
27:

15: ubicación_103B3B6
16:00 20 Movimientos R0, #0
17:00 21 Movimientos R1, #0

18: locret_103B3BA
19: BD E8 18 88 POBLACIÓN {R3,R4,R11,PC}

```

Figura 2-12

```

misterio7
01:02:02 46 Movimiento R2, R0
03:08 B9 CBNZ R0, ubicación_100E1D8

04:00 20 05:70 Movimientos R0, #0 06: loc_100E1D8
47 BX LR 07:90 F9 00 30 LDRSB.W R3, [R0]
08:02 E0 loc_100E1E4 B

12: ubicación_100E1E4
13:00 2B CMB R3, #0
14: FA D1 BNE ubicación_100E1DE

09: ubicación_100E1DE 15:10 1A Subs R0, R2, R0
10:01 32 AGREGA R2, n.o 1 16:6F F3 9F 70 BFC W R0, #0x1E, #2
11:92 F9 00 30 LDRSB.W R3, [R2] 17:70 47 BX LR
18: ; Fin de la función misterio7

```

Figura 2-13

```

01:                     misterio8
02: 2D E9 78 48 EMPIEJE.W (R3-R6,R11,LR)
03: 0D F2 10 08 AGREGAR 04: 0C 4E R6,          R11, SP, #0x10
=bytAray           LDR
05:09 E0             B      ubicación_100E34C

17:                     ubicación_100E34C
18:00 2A             CMP
19: F3 CC            BGT   ubicación_100E338

06:                     ubicación_100E338
20:01 3A           Subs    R2, n.o 1
07:05 78           LDRB   R5, [R0]
08:01 3A           Subs   R2, n.o 1
09:4D B1           ZBCC   R5, ubicación_100E352

10:0878           LDRB   R3, [R1]
11:9C 5D           LDRB   R4, [R3,R6]
12: AB 5D           LDRB   R3, [R5,R6]
13: A3 42           CMP    R3, R4
14:04 D1           BNE    ubicación_100E352

21:                     ubicación_100E352
22:00 2A           CMP    R2, #0
23:01 D1A          BGE    ubicación_100E35A      15:01 30      AGREGA   R0, n.o 1
                                         16:01 31      AGREGA   R1, n.o 1

26: loc_100E35A
27:0B78           LDRB   R3, [R1]
28:9 a 5 p. 29:03  LDRB   R2, [R3,R6]
29:               p.78   LDRB   R3, [R0]
30:9B 5D           LDRB   R3, [R3,R6]
31:98 1A           Subs   R0, R3, R2

32: locret_100E364
33: BD E8 78 88 POBLACIÓN          (R3-R6,R11,PC)
34: ; Fin de la función misterio8

```

Figura 2-14

01:	misterio9		
02: 2D E9 30 48 EMPUJE W {R4,R5,R11,LR}			
03: 0D F2 08 0B AGREGAR	R11, SP, n. ^o 8		
04: 09 4D	LDR	R5, = Matriz de bytes	
05: 0E E0	B	ubicación_100E312	
14: loc_100E312			
15: 04 78	LDRB	R4, [R0]	
16: 00 2C	CMP	R4, #0	
17: F5 D1	BNE	ubicación_100E304	
06:	ubicación_100E304		
07: 0B 78	LDRB	R3, [R1]	
08: 5 a 5 d	LDRB	R2, [R3,R5]	
09: 63 5D	LDRB	R3, [R4,R5]	
10: 93 42	CMP	R3, R2	
11: 04 D1	BNE	ubicación_100E318	
18: loc_100E318			
19: 0B 78 LDRB 20: 5A 5D 21: 03 78 22: 58			
5D 23: 98 1A 24:	LDRB	R3, [R1]	
AGREGA	R0, n. ^o 1	BD E8 30 88	R2, [R3,R5]
AGREGA	R1, n. ^o 1	POP.W 25: ; Fin	R3, [R3, R5]
12: 01 30		de la función	Subs
13: 01 31		misterio9	R0, R3, R2
			(R4,R5,R11,PC)

```

01:                         misterio10
02: 2D E9 70 48 EMPUJE W (R4-R6,R11,LR)
03: 0D F2 0C 0B AGREGAR           R11, SP, #0xC
04:37 F0 CC F9 BL              Cookie de seguridad push
05:84 B0                      SUB   SP, SP, #0x10
06:0D 46                      Movimiento R5, R1
07:00 24                      Movimiento R4, #0
08:10 2D                      CMP   R5, #0x10
09:16 46                      Movimiento R6, R2
10:0C D1a 3                   BCC   loc_1010786

11:1A 4B 12:68                LDR   R3, = __imp__ObtenerTiempoSistema
46                           Movimiento R0, SP
13:1B 68 14:98                LDR   R3, [R3]
47                           BLX   R3
15:00 9B 16:10                LDR   R3, [SP,#0x1C+var_1C]
24 17:33 60                  Movimiento R4, #0x10
STR                          R3, [R6]
18:01 9B 19:73                LDR   R3, [SP,#0x1C+var_18]
60                           STR   R3, [R6,#4]
20:02 9B                      LDR   R3, [SP,#0x1C+var_14]
21:B3 60                      STR   R3, [R6,#8]
22:03 9B                      LDR   R3, [SP,#0x1C+var_10]
23:F3 60                      STR   R3, [R6,#0xC]

24:                         loc_1010786
25:2B1B                     Subs  R3, R5, R4
26:04 2B                     CMP   R3, n.o 4
27:04 D3                   BCC   loc_1010796

28:11 4B                      LDR   R3, = __imp__ObtenerIDDeProcesoActual
29:1B 68 30:98                LDR   R3, [R3]
47                           BLX   R3
31:30 51                      STR   R0, [R6,R4]
32:0434                     AGREGA R4, n.o 4

33:                         loc_1010796
34:2B1B                     Subs  R3, R5, R4
35:04 2B                     CMP   R3, n.o 4
36:04 D3                   BCC   loc_10107A6

37:0C 4B                      LDR   R3, = __imp__ObtenerConteoDeTicks
38: 1B 68 39:                 LDR   R3, [R3]
98:47 40: 30                  BLX   R3
51:41: 04 34                 STR   R0, [R6,R4]
AGREGA                      R4, n.o 4

42: loc_10107A6
43:2B1B                     Subs  R3, R5, R4
44:08 2B                     CMP   R3, n.o 8
45:09 D3                   BCC   ubicación_10107C0

46:07 4B 47:68                LDR   R3, = __imp__Contador de rendimiento de consultas
46                           Movimiento R0, SP
48:1B 68 49:98                LDR   R3, [R3]
47                           BLX   R3
50:00 9B 51:32                LDR   R3, [SP,#0x1C+var_1C]
19:52:33 51                 AGREGA R2, R6, R4
STR                          R3, [R6,R4]
53:01 9B 54:08                LDR   R3, [SP,#0x1C+var_18]
34:55:53 60                 AGREGA R4, n.o 8
STR                          R3, [R2,#4]

58:                         ubicación_10107C0
57:20 46                      Movimiento R0, R4
58:04 B0                      AGREGA SP, SP, #0x10
59:37 F0 A4 F9 BL              __cookies de seguridad pop
60: 8D E8 70 88 POBLACIÓN    (R4-R6,R11,PC)
61: ; Fin de la función misterio10

```

Figura 2-16

86 Capítulo 2 ■ ARM

```

01:010185B0                         misterio11
02: 010185B0 2D E9 F8 4F EMPUJE.W [R3-R11 LR]
03: 010185B4 0D F2 20 0B AGREGAR R11, SP, #0x20
04: 010185B8 B0 F9 5A 30 LDRSH.W R3, [R0,#0xE8]
05: 010185BC 07 46 R7, R0               Movimiento
06: 010185B8 90 46 R2, R2             Movimiento
07: 010185C0 00 EB 83 03 AGREGAR.W R3, R0, R3, LSL #2
08: 010185C4 D3 F8 84 A0 LDR.W R10, [R3,#0x84]
09: 010185C8 7B 8F R3, [R7,#0x3A]      LDH
10: 010185CA B9 46 R9, R1             Movimiento
11. 010185CC C8 B9 R3, loc_1018602   CBNZ

12:010185CE B0 F9 5A 40 LDRSH.W R4, [R0,#0x5A]
13: 010185D2 17 F1 20 02 AGREGA.W R2, R7, #0x20
14: 010185D6 00 EB 44 03 AÑadir.W R3, R0, R4,LSL#1
15: 010185DA B3 F6 5C 60 LDRH.W R5, [R3,#0x5C]
16:010185DE 00EB 84 03 AGREGAR.W R3, R0, R4,LSL#2
17:010185E2 D3 F8 84 00 LDR.W R0, [R3,#0x84]
18:010185E6B6369                      LDH     R3, [R0,#0xC]
19:010185E8 08 6C                     LDR     R6, [R0,#0x40]
20: 010185EA 03 EB 45 03 AGREGAR.W R3, R3, R5,LSL#1
21:010185EE 98 19                     AGREGA   R3, R3, R6
22:010185F01C78                      LDRB    R4, [R2]
23:010185F2 59 78                     LDRB    R3, [R3,#1]
24:010185F4 43 EA 04 24 ORR.W R4, R3, R4, LSL#8
25:010185F8438A                      LDH     R3, [R0,#0x12]
26:010185FA2340                      Y       R3, R4
27:010185FC 99 19                     AGREGA   R1, R3, R6
28:010185FE FD F7 8D FF BL          sub_101861C

29:01018602 loc_1018602
30:01018602BA 8E                     LDH     R2, [R7,#0x34]
31:01018604 BB6A                     LDR     R3, [R7,#0x28]
32:01018606 D0 18                     AGREGA   R0, R2, R3
33:01018608 9A F8 02 30 LDRB.W R3, [R10,#2]
34:0101860C 08 B1                     ZBCC   R3, ubicación_1018612

35:0101860E 00 22
36:01018610 00 E0
                           Movimiento   R2, #0           37:01018612
                           B            loc_1018614        38:01018612 3A 6A
                                         LDR     R2, [R7,#0x20]

39:01018614 loc_1018614
40: 01018614 FB 8E LDRH 41: 01018616 B8 F1 00 0F
CMP.W 42: 0101861A 01 D0
                           AGREGA   R0, R0, R2
                           Subs    R3, R3, R2
                           Movimiento
                           loc_1018620

43:0101861C 80 18
44:0101861E9B1A
                           AGREGA   R0, R0, R2
                           Subs    R3, R3, R2
                           Movimiento
                           loc_1018620

45: 01018620 46:                         loc_1018620
01018620 C9 F8 00 30 FUERZA.W R3, [R9]
47: 01018624 BD EB F8 8F POP.W 48: 01018624 ; Fin de la
función misterio11
(R3-R11,PC)

```

Figura 2-17

CAPÍTULO

3

El núcleo de Windows

En este capítulo se tratan los principios y las técnicas necesarias para analizar el código de los controladores en modo kernel, como los rootkits, en la plataforma Windows. Dado que los controladores interactúan con el sistema operativo a través de interfaces bien definidas, la tarea analítica se puede descomponer en los siguientes objetivos generales:

- Comprender cómo se implementan los componentes principales del sistema operativo
- Comprender la estructura de un controlador
- Comprender las interfaces usuario-controlador y controlador-SO y cómo Windows los implementa
- Comprender cómo se manifiestan ciertas construcciones de software del controlador en forma binaria
- Aplicar sistemáticamente los conocimientos de los pasos anteriores en la práctica general. proceso de ingeniería inversa

Si el proceso de ingeniería inversa de los controladores de Windows pudiera modelarse como una tarea discreta, el 90 % comprendería cómo funciona Windows y el 10 % comprendería el código ensamblador. Por lo tanto, el capítulo está escrito como una introducción al núcleo de Windows para ingenieros inversos. Comienza con un análisis de las interfaces de usuario-núcleo y su implementación. A continuación, analiza las listas enlazadas y cómo se utilizan en Windows. A continuación, explica conceptos como subprocesos, procesos, memoria, interrupciones y cómo se utilizan en el núcleo y los controladores. Después, se analiza la arquitectura de un controlador en modo kernel y la interfaz de programación entre el controlador y el kernel, y se concluye aplicando estos conceptos a la ingeniería inversa de un rootkit.

A menos que se especifique lo contrario, todos los ejemplos de este capítulo se toman de Versión 8.0.

Fundamentos de Windows

Comenzamos con un análisis de los conceptos básicos del kernel de Windows, incluidas las estructuras de datos fundamentales y los objetos del kernel relevantes para la programación de controladores y la ingeniería inversa.

Disposición de la memoria

Al igual que muchos sistemas operativos, Windows divide el espacio de direcciones virtuales en dos partes: el espacio del núcleo y el espacio del usuario. En x86 y ARM, los 2 GB superiores están reservados para el núcleo y los 2 GB inferiores para los procesos del usuario. Por lo tanto, las direcciones virtuales de 0 a 0x7fffffff están en el espacio del usuario, 0x80000000 y superiores están en el espacio del núcleo. En x64, se aplica el mismo concepto, excepto que el espacio del usuario es de 0. a 0x000007ff ffffff y el espacio del kernel es 0xfffff0800'00000000 y superior.

La Figura 3-1 ilustra el diseño general en x86 y x64. El espacio de memoria del núcleo es prácticamente el mismo en todos los procesos. Sin embargo, los procesos en ejecución sólo tienen acceso a su espacio de direcciones de usuario; el código en modo núcleo puede acceder a ambos. (Algunos rangos de direcciones del núcleo, como los de sesión e hiperespacio, varían de un proceso a otro). Este es un hecho importante que hay que tener en cuenta porque volveremos a él más adelante cuando analicemos el contexto de ejecución. Las páginas en modo núcleo y en modo usuario se distinguen por un bit especial en su entrada de la tabla de páginas.

Cuando se programa la ejecución de un hilo en un proceso, el sistema operativo cambia un registro específico del procesador para que apunte al directorio de páginas de ese proceso en particular. Esto es para que todas las traducciones de direcciones virtuales a físicas sean específicas del proceso y no de otros. Así es como el sistema operativo puede tener múltiples procesos y cada uno de ellos tiene la ilusión de que posee todo el espacio de direcciones del modo usuario. En las arquitecturas x86 y x64, el registro base del directorio de páginas es CR3; en ARM es el registro base de la tabla de traducción (TTBR).



NOTA: Es posible cambiar este comportamiento predeterminado especificando el parámetro /3GB en las opciones de arranque. Con /3GB, el espacio de dirección del usuario aumenta a 3 GB y el 1 GB restante es para el núcleo.

Los rangos de direcciones de usuario/núcleo se almacenan en dos símbolos en el núcleo: MmSystemRangeStart (núcleo) y MmHighestUserAddress (usuario). Estos símbolos se pueden ver con un depurador de núcleo. Es posible que notes que hay un espacio de 64 KB entre el espacio de usuario y el de núcleo en x86/ARM. Esta región, normalmente denominada región sin acceso, está ahí para que el núcleo no cruce accidentalmente el límite de direcciones y corrompa la memoria en modo usuario. En x64, el lector astuto puede notar que 0xfffff8000'00000000 es una dirección no canónica y, por lo tanto, no puede ser utilizada por el sistema operativo. Esta dirección realmente solo se utiliza como separador entre el espacio de usuario y el de núcleo. La primera dirección utilizable en el espacio de núcleo comienza en 0xfffff8000'00000000.

Inicialización del procesador

Cuando el núcleo arranca, realiza una inicialización básica para cada procesador. La mayoría de los detalles de inicialización no son vitales para las tareas diarias de ingeniería inversa, pero es importante conocer algunas de las estructuras principales.

La región de control del procesador (PCR) es una estructura por procesador que almacena información crítica y el estado de la CPU. Por ejemplo, en x86 contiene la dirección base del IDT y el IRQL actual. Dentro del PCR hay otra estructura de datos llamada bloque de control de la región del procesador (PRCB). Es una estructura por procesador que contiene información sobre el procesador, es decir, tipo de CPU, modelo, velocidad, subproceso actual, etc.

90 Capítulo 3 ■ El núcleo de Windows

que se está ejecutando, próximo hilo a ejecutar, cola de DPC a ejecutar, etc. Al igual que el PCR, esta estructura no está documentada, pero aún puede ver su definición con el depurador del núcleo:

x64 (x86 es similar)

```
PCR

0: kd>dt nt!_KPCR
+0x000 NT_Tib : _NT_TIB
+0x000 Base de datos Gdt :Ptr64_KGDTENTRY64
+0x008 Base de datos de transacciones :Ptr64_KTSS64
+0x010 UsuarioRsp +0x018 :Uint8B
Yo :Ptr64_KPCR
+0x020 ActualPrcb :Ptr64_KPRCB
...
+0x180 Prcb : _KPRCB
```

```
Braza de la Procesadora

0: kd>dt nt!_KPRCB
+0x000 MxCsr :Uint4B
+0x004 Número heredado :UChar
+0x005 ReservadoDebe ser cero: UChar
+0x006 Solicitud de interrupción: UChar
+0x007 Inactividad detenida :UChar
+0x008 Hilo actual : Ptr64 _K_HILO
+0x010 Siguiente hilo : Ptr64 _K_HILO
+0x018 Hilo inactivo : Ptr64 _K_HILO
...
+0x040 Estado del procesador: _KPROCESSOR_STATE
+0x5f0 TipoCpu +0x5f1 :Caracter
IdCpu +0x5f2 PasoCpu :Caracter
+0x5f2 PasoCpu +0x5f3 :Uint2B
ModeloCpu +0x5f4 MHz :UChar
:Uint4B
...
+0x2d80 DpcData : [2] _DATOS_KDPC
+0x2dc0 DpcStack :Ptr64 Vacío
+0x2dc8 Profundidad máxima de cola de Dpc: Int4B
...
```

```
BRAZO

PCR

0: kd>dt nt!_KPCR
+0x000 NT_Tib : _NT_TIB
+0x000 TibPad0 : [2] Uint4B
+0x008 Repuesto01 : Ptr32 Vacío
+0x00c Yo mismo :Ptr32_KPCR
+0x010 ActualPrcb :Ptr32_KPRCB
...
```

```

Entrada de la República

0: kd>dt ntl_KPCR
+0x000 NtTib : _NT_TIB
+0x000TibPad0 : [2] Uint4B
+0x008 Repuesto1 : Ptr32 Vacío
+0x00c Yo mismo :Ptr32_KPCR
+0x010 ActualPrcb :Ptr32_KPRCB
...
0: kd>dt ntl_KPRCB
+0x000 Número heredado :UCar
+0x001 ReservadoMustBeZero; UChar
+0x002 Inactividad detenida :UCar
+0x004 Hilo actual : Ptr32 _K-HILO
+0x008 Siguiente hilo : Ptr32 _K-HILO
+0x00c Hilo inactivo : Ptr32 _K-HILO
...
+0x020 Estado del procesador: _KPROCESSOR_STATE
+0x3c0 Modelo de procesador: UInt2B
+0x3c2 Revisión del procesador: UInt2B
+0x3c4 MHz :UInt4B
...
+0x690 Datos de Dpc : [2] _DATOS_KDPC
+0x6b8 Pila de Dpc : Ptr32 Vacío
...
+0x900 Número de interrupciones: UInt4B
+0x904 Tiempo del núcleo :UInt4B
+0x908 Hora del usuario :UInt4B
+0x90c Tiempo de Dpc :UInt4B
+0x910 Tiempo de interrupción :UInt4B
...

```

El PCR de un procesador actual siempre es accesible desde el modo kernel a través de registros especiales. Se almacena en el segmento FS (x86), el segmento GS (x64) o uno de los registros del coprocesador del sistema (ARM). Por ejemplo, el kernel de Windows exporta dos rutinas para obtener el EPROCESS y el ETHREAD actuales: PsGetCurrentProcess y PsGetCurrentThread. Estas rutinas funcionan consultando el PCR/PRCB:

```

PsGetCurrentThread procedimiento cerca
movimiento      rax, gs:188h; gs[0] es el PCR, el desplazamiento 0x180 es el PRCB,
                ; el desplazamiento 0x8 en el PRCB es el subproceso actual
campo
Retirada
Fin del hilo PsGetCurrentThread

PsGetCurrentProcess procedimiento cerca
movimiento      rax, gs:188h ; obtener el hilo actual (ver arriba)
movimiento      rax, [rax+0B8h] ; el desplazamiento 0x70 en ETHREAD es el asociado
                ; proceso (en realidad ETHREAD.ApcState.Process)
Retirada
Fin del proceso PsGetCurrentProcess

```

Llamadas al sistema

Un sistema operativo administra los recursos de hardware y proporciona interfaces a través de las cuales los usuarios pueden solicitarlos. La interfaz más comúnmente utilizada es la llamada del sistema. Una llamada del sistema es típicamente una función en el núcleo que atiende las solicitudes de E/S de los usuarios; se implementa en el núcleo porque solo el código con privilegios altos puede administrar dichos recursos. Por ejemplo, cuando un procesador de textos guarda un archivo en el disco, primero necesita solicitar un identificador de archivo al núcleo, escribe en el archivo y luego confirma el contenido del archivo en el disco duro; el SO proporciona llamadas del sistema para adquirir un identificador de archivo y escribir bytes en él. Si bien estas parecen ser operaciones simples, las llamadas del sistema deben realizar muchas tareas importantes en el núcleo para atender la solicitud. Por ejemplo, para obtener un identificador de archivo, debe interactuar con el sistema de archivos (para determinar si la ruta es válida o no) y luego solicitar al administrador de seguridad que determine si el usuario tiene derechos suficientes para acceder al archivo; Para escribir bytes en el archivo, el núcleo necesita averiguar en qué volumen del disco duro se encuentra el archivo, enviar la solicitud al volumen y empaquetar los datos en una estructura entendida por el controlador del disco duro subyacente. Todas estas operaciones se realizan con total transparencia para el usuario.

Los detalles de implementación de las llamadas al sistema de Windows no están documentados oficialmente, por lo que vale la pena explorarlos por razones intelectuales y pedagógicas. Si bien la implementación varía entre procesadores, los conceptos siguen siendo los mismos. Primero explicaremos los conceptos y luego analizaremos los detalles de implementación en x86, x64 y ARM.

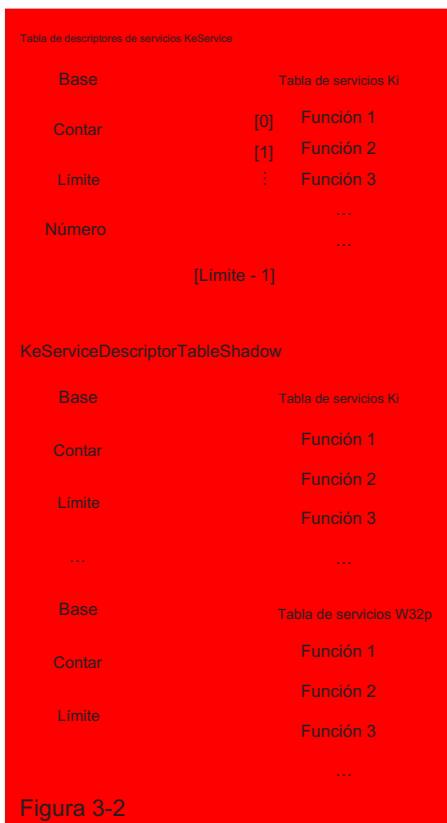
Windows describe y almacena información de llamadas del sistema con dos estructuras de datos: un descriptor de tabla de servicios y una matriz de punteros de función/desplazamientos. El descriptor de tabla de servicios es una estructura que contiene metadatos sobre llamadas del sistema admitidas por el SO; su definición no está documentada oficialmente, pero muchas personas han realizado ingeniería inversa de sus miembros de campo importantes de la siguiente manera. (También puede averiguar estos campos analizando KiSystemCall64 o KiSystemService rutinas.)

```
estructura de tipo definido _KSERVICE_TABLE_DESCRIPTOR
{
    PULONG Base; // matriz de direcciones o desplazamientos
    PULONG Conde;
    Límite ULONG; // tamaño de la matriz
    Número PUCHAR;
    ...
} DESCRIPTOR_TABLA_KSERVICE, *DESCRIPTOR_TABLA_PKSERVICE;
```

Base es un puntero a una matriz de punteros de función o desplazamientos (según el procesador); un número de llamada del sistema es un índice en esta matriz. El límite es el número de entradas en la matriz. El núcleo mantiene dos matrices globales de KSERVICE_DESCRIPTOR_DESCRIPTOR: KeServiceDescriptorTable y KeServiceDescriptorTableShadow.

El primero contiene la tabla de llamadas al sistema nativas; el segundo contiene los mismos datos, además de la tabla de llamadas al sistema para subprocessos de la GUI. El núcleo también mantiene dos tablas de llamadas al sistema globales.

punteros a las matrices de direcciones/desplazamientos: KiServiceTable apunta a la tabla de llamadas al sistema que no es de GUI y W32pServiceTable apunta a la de GUI. La Figura 3-2 ilustra cómo estas estructuras de datos se relacionan entre sí en x86.



En x86, el campo Base es una matriz de punteros de función para las llamadas al sistema:

```

0: kd> dps nt!KeServiceDescriptorTable 81472400 813564d0 nt!
KiServiceTable ; Base
81472404 00000000
81472408 000001ad
8147240c 81356b88 nt!KiArgumentTable 0: kd> dd nt!KiServiceTable

813564d0 81330901 812cf1e2 81581540 816090af
813564e0 815be478 814b048f 8164e434 8164e3cb
813564f0 812dfa09 814e303f 814a0830 81613a9f
81356500 814e5b65 815b9e3a 815e0c4e 8158ce33
...
0: kd> dps nt!KiServiceTable
813564d0 81330901 nt!NIWorkerFactoryWorkerReady 813564d4 812cf1e2 nt!NtYieldExecution

813564d8 81581540 nt!NtWriteVirtualMemory 813564dc 816090af nt!
NtWriteRequestData
    
```

94 Capítulo 3 ■ El núcleo de Windows

```
813564e0 815be478 nt!NtWriteFileGather
813564e4 814b048f nt!NtWriteFile
```

Sin embargo, en x64 y ARM, es una matriz de números enteros de 32 bits que codifica el desplazamiento de la llamada del sistema y la cantidad de argumentos que se pasan a la pila. El desplazamiento está contenido en los 20 bits superiores y la cantidad de argumentos en la pila está contenida en los 4 bits inferiores. El desplazamiento se agrega a la base de KiServiceTable.

Para obtener la dirección real de la llamada al sistema. Por ejemplo:

```
0: kd> dps nt!KeServiceDescriptorTable
ffff803'955cd900 fffff803'952ed200 nt!KiServiceTable ; Base
ffff803'955cd908 00000000'00000000
ffff803'955cd910 00000000'0000001ad
ffff803'955cd918 fffff803'952edf6c nt!KiTablaArgumentos
0: kd> u ntdll!NtCreateFile
ntdll!NtCreateFile:
000007f8`34f23130 movimiento      R10, RCX
000007f8`34f23133 movimiento      eax,53h ; número de llamada al sistema
Llamada al sistema 000007f8`34f23138
...
0: kd> x nt!KiServiceTable
ffff803'952ed200 nt!KiServiceTable (<sin información de parámetros>)
0: kd> dd nt!KiServiceTable+0x53^4L1
ffff803'952ed34c 03ea2c07          ; desplazamiento codificado y número de argumentos
0: kd> u nt!KiServiceTable + (0x03ea2c07>>4) ; obtener el desplazamiento y agregarlo a
Base
nt!NtCrearArchivo:
subtítulo fffff803'956d74c0      rsp,88h
ffff803'956d74c7 o más          Ea, Ea
movimiento fffff803'956d74c9      qword ptr [rsp+78h],rax
movimiento fffff803'956d74ce      palabra clave d [rsp+70h],20h
0: kd> ?0x03ea2c07 y 0xf        ; número de argumentos
Evaluar expresión: 7 = 00000000'00000007
.NtCreateFile toma 11 argumentos. Los primeros 4 se pasan a través de registros y
los últimos 7 se pasan a la pila
```

Como se ha demostrado, cada llamada del sistema se identifica mediante un número que es un índice en KiServiceTable o W32pServiceTable. En el nivel más bajo, las API del modo usuario se descomponen en una o más llamadas del sistema.

En términos conceptuales, así es como funcionan las llamadas del sistema en Windows. Los detalles de implementación varían según la arquitectura y la plataforma del procesador. Las llamadas del sistema se implementan normalmente a través de interrupciones de software o instrucciones específicas de la arquitectura , cuyos detalles se tratan en las siguientes secciones.

Fallas, trampas e interrupciones

Para preparar las siguientes secciones, necesitamos introducir algo de terminología básica para explicar cómo los dispositivos periféricos y el software interactúan con el procesador.

En los sistemas informáticos contemporáneos, el procesador normalmente está conectado a

dispositivos periféricos a través de un bus de datos como PCI Express, FireWire o USB. Cuando un dispositivo requiere la atención del procesador, provoca una interrupción que obliga al procesador a pausar lo que esté haciendo y a procesar la solicitud del dispositivo. ¿Cómo sabe el procesador cómo manejar la solicitud? En el nivel más alto, se puede pensar en una interrupción como algo asociado a un número que luego se utiliza para indexar en una matriz de punteros de función. Cuando el procesador recibe la interrupción, ejecuta la función en el índice asociado con la solicitud y reanuda la ejecución donde estaba antes de que se produjera la interrupción. Estas se denominan interrupciones de hardware porque las generan los dispositivos de hardware.

Son asíncronos por naturaleza.

Cuando el procesador está ejecutando una instrucción, puede encontrarse con excepciones. Por ejemplo, la instrucción provoca un error de división por cero, hace referencia a una dirección no válida o activa una transición de nivel de privilegio. Para los fines de este análisis, las excepciones se pueden clasificar en dos categorías: fallos y trampas. Un fallo es una excepción corregible. Por ejemplo, cuando el procesador ejecuta una instrucción que hace referencia a una dirección de memoria válida pero los datos no están presentes en la memoria principal (se han paginado), se genera una excepción de fallo de página. El procesador gestiona esto guardando el estado de ejecución actual, llama al controlador de fallos de página para corregir esta excepción (mediante la paginación de los datos) y vuelve a ejecutar la misma instrucción (que ya no debería provocar un fallo de página). Una trampa es una excepción provocada por la ejecución de tipos especiales de instrucciones. Por ejemplo, en x64, la instrucción SYSCALL hace que el procesador comience a ejecutarse en una dirección especificada por un MSR; una vez que el controlador termina, la ejecución se reanuda en la instrucción inmediatamente posterior a SYSCALL. Por lo tanto, la principal diferencia entre una falla y una trampa es dónde se reanuda la ejecución. Las llamadas al sistema se implementan comúnmente a través de excepciones especiales o instrucciones de trampa.

Interrupciones

La arquitectura Intel define una tabla de descripción de interrupciones (IDT) con 256 entradas; cada entrada es una estructura con información que define el controlador de interrupciones. La dirección base de la IDT se almacena en un registro especial llamado IDTR. Una interrupción se asocia con un índice en esta tabla. Existen interrupciones predefinidas reservadas por la arquitectura. Por ejemplo, 0x0 es para excepción de división, 0x3 es para punto de interrupción de software y 0xe es para fallas de página. Las interrupciones 32 a 255 están definidas por el usuario.

En x86, cada entrada de la tabla IDT es una estructura de 8 bytes definida de la siguiente manera:

1: kd> dt ntl!_KIDTENTRY	
+0x000 Desplazamiento	:Uint2B
Selector +0x002	:Uint2B
+0x004 Acceso	:Uint2B
+0x006 Desplazamiento extendido: Uint2B	

(En x64, la estructura de entrada de IDT es prácticamente la misma, excepto que la dirección del controlador de interrupciones se divide en tres miembros. Puede verlo haciendo un dump

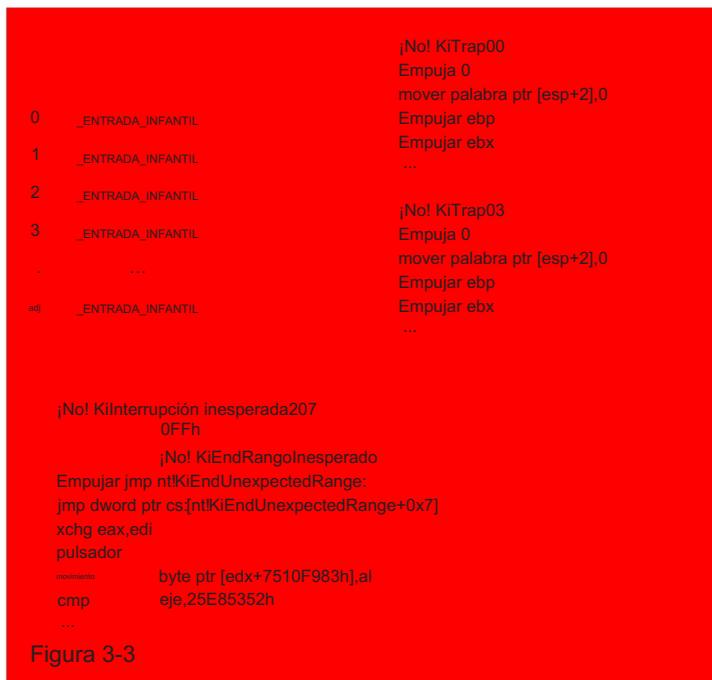
96 Capítulo 3 ■ El núcleo de Windows

La estructura nt!_KIDTENTRY64 . Observe también que el IDTR tiene 48 bits de ancho y está dividido en dos partes: dirección base y límite del IDT. WinDBG muestra solo la dirección base.)

La dirección del controlador de interrupciones se divide entre Offset y ExtendedOffset campos. A continuación se muestra un ejemplo de decodificación del IDT y de desensamblaje del controlador de interrupción de división por cero (0x0):

```
1: kd>r @idtr
Identificación = 8b409d50
1: kd> dt nt!_KIDTENTRY 8b409d50
+0x000 Desplazamiento :0xa284
 Selector +0x002 :8
+0x004 Acceso :0x8e00
+0x006 Desplazamiento extendido: 0x813c
1: kd>u0x813ca284
iNo! KiTrap00:
813ca284 empujar 0
813ca286movimiento palabra ptr [esp+2],0
813ca28d empujar -----
813ca28e empujar ebx
```

La figura 3-3 ilustra el IDT en x86.



En los procesadores anteriores a Pentium 2, Windows utiliza la interrupción 0x2e para implementar llamadas del sistema. Los programas en modo usuario llaman a las API en kernel32.dll (o kernelbase.dll),

que eventualmente se resuelven en trozos cortos en ntdll.dll que activan la interrupción 0x2e.

Para ilustrarlo, considere el siguiente fragmento de la rutina API kernelbase!CreateFileW en Windows 7:

```
[dentro de kernelbase!CreateFileW]
...
.text:0DCE9C87 movimiento ecx, [ebp+dwFlagsAndAttributes]
.text:0DCE9C8A push [ebp+lpSecurityAttributes]
.text:0DCE9C8D mov eax, [ebp+dwAcceso deseado]
.text:0DCE9C90 push [nombre_archivo_ebp+lp]
.text:0DCE9C93 mov esi, ds:_imp__NtCreateFile@44 .text:0DCE9C99 push
[ebp+var_4] .text:0DCE9C9C y ecx, 7FA7h

.text:0DCE9CA2 push [ebp+dwShareMode]
.text:0DCE9CA5 mov [ebp+dwBanderasYATributos], ecx
.text:0DCE9CA8 ecx de inserción
.text:0DCE9CA9 mensaje push ebx
.text:0DCE9CAA lea ecx, [ebp+var_20] .text:0DCE9CAD
push ecx
.text:0DCE9CAE o eax, 100080h
.text:0DCE9CB3 lea ecx, [ebp+var_64] .text:0DCE9CB6
push ecx
.text:0DCE9CB7 enviar y recibir
.text:0DCE9CB8 mov [ebp+dwAcceso deseado], eax
.text:0DCE9CBB leer eax, [ebp+var_8] .text:0DCE9CBE
enviar eax
.text:0DCE9CBF llamar a esi; NtCreateFile(...)
```

Realiza una validación preliminar (no se muestra aquí) y luego llama ntdll!NtCreateFile. La implementación es la siguiente:

```
[ntdll!NtCreateFile]
.text:77F04A10 _NtCreateFile@44 proc near .text:77F04A10 mov
eax, 42h ; syscall # .text:77F04A15 mov edx, 7FFE0300h ;
KUSER_SHARED_DATA SystemCall ; el símbolo para 0x7ffe0300 es SharedUserData
SystemCallStub .text:77F04A1A call dword ptr [edx] ; controlador de llamadas

.text:77F04A1C retn 2Ch ; volver al llamador
.text:77F04A1C _NtCreateFile@44 final
```

NtCreateFile establece EAX en 0x42 porque ese es el número de llamada del sistema para NtCreateFile en el núcleo. A continuación, lee un puntero en 0x7ffe0300 y lo llama. ¿Qué tiene de especial 0x7ffe0300? En todas las arquitecturas, existe una estructura por proceso denominada KUSER_SHARED_DATA que siempre se asigna a 0x7ffe0000. Contiene información genérica sobre el sistema y un campo denominado SystemCall:

```
0:000> dt ntdll!_DATOS_COMPARTIDOS_DE_KUSER
+0x000 TickCountLowObsoleto: UInt4B
+0x004 Multiplicador de recuento de ticks: UInt4B
+0x008 Tiempo de interrupción: _KSYSTEM_TIME +0x014
Tiempo del sistema: _KSYSTEM_TIME
```

98 Capítulo 3 ■ El núcleo de Windows

```
+0x020 Sesgo de zona horaria: _KSYSTEM_TIME
...
+0x2f8 Instrucción de prueba: UInt8B
+0x300 SystemCall: UInt4B; controlador de llamada al sistema
+0x304 SystemCallReturn: UInt4B
...
```

Al desmontar el stub de llamada del sistema, verá esto:

```
0:000> u poi(SharedUserData\SystemCallStub) ntdll!KiIntSystemCall: 76e46500
lea edx,[esp+8] 76e46504 int 2Eh
```

```
76e46506 retractado
```

```
76e46507 no
```

Al volcar la entrada IDT en el índice 0x2e se muestra que KiSystemService es el despachador de llamadas del sistema:

```
0:kd>idt0x2e
Volcado de IDT: ... 2e:
8284b22e nt!KiSystemService 0: kd> u nt!
KiSystemService nt!KiSystemService:
8284b22e push 0 8284b230
push ebp 8284b231 push
ebx 8284b232 push esi
8284b233 push edi 8284b234
push fs 8284b236 mov
ebx,30h
```

Los detalles del despachador de llamadas del sistema se tratan en la siguiente sección.

Trampas

En la sección anterior se explica cómo se implementan las llamadas del sistema con el mecanismo de procesamiento de interrupciones integrado. En esta sección se explica cómo se implementan mediante instrucciones trap en x64, x86 y ARM.

Comenzando con la implementación en x64, considere el stub de llamada del sistema ntdll! NtCreateFile:

```
01: .text:00000001800030F0 público ZwCreateFile 02: .text:00000001800030F0 ZwCreateFile proc
cerca de 03: .text:00000001800030F0 mov r10, rcx
...
04: .text:00000001800030F3 mov eax, 53h
05: .text:00000001800030F8 llamada al sistema
06: .text:00000001800030FA retn
07: .text:00000001800030FA ZwCreateFile fin
```

La línea 3 guarda el primer argumento en R10; debe hacerlo porque la semántica de SYSCALL dicta que la dirección de retorno (línea 6) debe almacenarse en RCX. La línea 4 guarda el número de llamada del sistema en EAX; una vez que SYSCALL pasa al modo kernel, lo utilizará como índice en la matriz KiServiceTable . La línea 5 ejecuta SYSCALL que pasa al modo kernel. ¿Cómo lo hace? La documentación de SYSCALL especifica que RIP se cargará con un valor definido por el MSR IA32_LSTAR (0xc0000082), y puede observarlo en el depurador:

```
1: kd> rdmsr 0xC0000082  
msr[0c0000082] = fffff800'89e96dc0  
1: kd> u fffff800'89e96dc0  
;No!KiSystemCall64:  
fffff800'89e96dc0 intercambios  
movimiento fffff800'89e96dc3  
movimiento fffff800'89e96dcc  
fffff800'89e96dd5 empujar  
fffff800'89e96dd7 empujar  
fffff800'89e96ddf empujar  
qword ptr gs:[10h],rsp  
rsp.qpalabra ptr gs:[1A8h]  
2Bh  
qword ptr gs:[10h]  
r11
```

Esta salida del depurador del núcleo indica que SYSCALL siempre terminará ejecutando KiSystemCall64 en el núcleo. De hecho, KiSystemCall64 es el principal despachador de llamadas del sistema en Windows x64. Windows configura el MSR LSTAR IA32 en KiSystemCall64, al principio del proceso de inicialización del procesador (ver KilnitializeBootStructures). Es el principal responsable de guardar el contexto del modo de usuario, configurar una pila de kernel, copiar los argumentos del modo de usuario a la pila de kernel, determinar la llamada del sistema en KiServiceTable (o W32pServiceTable) utilizando el índice pasado desde EAX, invocar la llamada del sistema y regresar al modo de usuario. ¿Cómo sabe el despachador de llamadas al sistema dónde regresar en el modo de usuario? Recuerde que SYSCALL Guarda la dirección de retorno en RCX. Una vez que la llamada del sistema termina su trabajo y regresa, el despachador de llamadas del sistema utiliza la instrucción SYSRET , que establece RIP en RCX para que vuelva al modo de usuario.

Si bien KiSystemCall64 admite muchas funcionalidades (creación de perfiles de llamadas al sistema, programación en modo usuario, depuración, etc.), su responsabilidad principal es enviar solicitudes de llamadas al sistema. En la sección anterior, indicamos que cada valor de la matriz KiServiceTable codifica un desplazamiento de la llamada al sistema y la cantidad de argumentos que se pasan a la pila. Esto se puede observar en el siguiente fragmento de código de KiSystemCall64:

100 Capítulo 3 ■ El núcleo de Windows

```

09: empujar          2Bh
10: pulsar qword ptr gs:10h ; KPCR->UserRsp
11: empujar          r11
12:
13: todavía          ; habilitar interrupciones
14: movimiento        [rbx+88h], rcx ; KTHREAD->Primer argumento
15: movimiento        [rbx+80h], eax ; KTHREAD->Número de llamada del sistema
16: KiSystemServiceStart proc cerca de 17: mov
                           [rbx+90h], rsp: KTHREAD->TrapFrame
18: movimiento        edi, eax           ; eax = llamada al sistema #
19: shr               edición,          ; determinar qué tabla de llamadas al sistema
20: y                 7 edición, 20h
21: y                 EAX, 0FFFh          ; índice en la tabla (recuperar llamada al sistema de 64 bits)
codificación)
22: KiSystemServiceRepeat proc cerca
23: lea               r10, Tabla de descriptores de servicios Ke
24: lea               r11, KeServiceDescriptorTableShadow
25: prueba            dword ptr [rbx+78h], 40h ; determina si es un hilo GUI
26: cmovnz r10, r11      ; ¿Qué tabla utilizar?
27: cmp               eax, [rdi+r10+10h]; ¿Es esa tabla de llamadas al sistema dentro de la tabla?
¿Límite?
28:                   ; es decir, KSERVICE_TABLE_DESCRIPTOR.Límite
29: jnb               caso_númerodellamadainválido
30: movimiento        r10, [rdi+r10]       ; seleccione la tabla correcta
31: movsxd r11, dword ptr [r10+rax*4]; obtener el desplazamiento de la llamada al sistema
32: movimiento        Rax, r11
33: sar               r11, 4
34: añadir            r10, r11          ; agréguelo a la base de la tabla para obtener syscall VA
35: cmp               edi, 20h; edi determina qué tabla. Aquí se utiliza para
                           ,determinado si es una GUI
36: jnz               caso corto_nonguirequest
37: movimiento        r11, [rbx+0F0h]
38:
39: Se acerca el procedimiento KiSystemServiceCopyEnd
40: prueba            cs:dword_140356088, 40 horas
41: jnz               registro_de_casos habilitado
42: llamada           r10              ; invocar la llamada del sistema

```

Recorrer KiSystemCall64 puede ser una experiencia instructiva y es
Lo dejé como ejercicio.

En x86, Windows utiliza la instrucción SYSENTER para implementar llamadas del sistema.
La mecánica es similar a la de SYSCALL en procesadores x64. Antes de pasar a la
implementación, veamos el código auxiliar de la llamada al sistema para ntdll!NtQuery
Proceso de información:

```

01: _ZwQueryInformationProcess@20 procedimiento cerca
02: mov ; número de llamada 5A0E 8800ma
03: llamada          sub_6A214FC0 14          ; trozo
04: retomado          horas             ; limpia la pila y regresa. NtQueryInformationProcess
acepta
05:                   ; 5 parámetros y se pasan a la pila
                           ; SYSENTER regresará aquí (ver siguiente)
ejemplo)

```

```

06: _ZwQueryInformationProcess@20 fin
07:
08: sub_6A214FCD proc cerca
09: movimiento      edx, esp
10: sistema de información
11: retornar
12: sub_6A214FCD fin

```

ntdll!NtCreateFile establece el número de llamada del sistema y llama a otra rutina que guarda el puntero de pila en EDX, seguida de la instrucción SYSENTER . La documentación de Intel indica que SYSENTER establece EIP en el valor almacenado en MSR 0x176:

```

0: kd> rdmsr 0x176
msr[176] = 00000000'80f7d1d0
0: kd>u 00000000'80f7d1d0

nt!KiFastCallEntrada:
80f7d1d0 movimiento      ecx,23h
80f7d1d5 empuje          30 horas
80f7d1d7 estallido        por
80f7d1d9 movimiento        ds,cx
movimiento 80f7d1db       es,cx
movimiento 80f7d1dd       ecx,dword ptr fs:[40h]

```

La salida del depurador muestra que cuando se ejecuta la instrucción SYSENTER , pasa al modo kernel y comienza a ejecutar KiFastCallEntry . es el principal despachador de llamadas del sistema en Windows x86 que utiliza SYSENTER (piénselo como KiSystemCall64 en x64). Una característica peculiar de SYSENTER es que no guarda la dirección de retorno en un registro como lo hace SYSCALL . Una vez que se completa la llamada del sistema, ¿cómo sabe el núcleo dónde regresar? La respuesta consta de dos partes. Usando NtQueryInformationProcess nuevamente como ejemplo, antes de llamar a SYSENTER para ingresar al modo núcleo, primero la secuencia de llamadas se ve así:

```

kernel32!ObtenerUnidadesLógicas ->
ntdll!NtQueryInformationProcess ->
Talón -> SYSENTER

```

Esto significa que la dirección de retorno ya está configurada en la pila antes de SYSENTER. Se ejecuta. Inmediatamente antes de SYSENTER, KiFastSystemCall guarda el puntero de pila en EDX. En segundo lugar, después de SYSENTER, el código pasa a KiFastCallEntry, que guarda este puntero de pila. Una vez que se completa la llamada del sistema, el despachador de llamadas del sistema ejecuta la instrucción SYSEXIT . Por definición, SYSEXIT establece EIP en EDX y ESP en ECX; en la práctica, el núcleo establece EDX en ntdll!KiSystemCallRet y ECX en el puntero de pila antes de entrar en el núcleo. Puede observar esto en acción estableciendo un punto de interrupción en la instrucción SYSEXIT dentro de KiSystemC y luego ver la pila desde allí:

```

1: kd>r
eax=00000000 ebx=00000000 ecx=029af304 edx=77586954 esi=029af3c0 edi=029afa04
esp=a08f7c8c ebp=029af3a8 iopl=0 cs=0008 ss=0010 ds=0023 es=0023
fs=0030 gs=0000 nv up ei ng nz na pe cy
EFL=00000287

```

```

nt!KiSystemCallExit2+0x18:
815d0458 salida del sistema
1: kd> dps @ecx L5 # SYSEXIT establecerá ESP en ECX (tenga en cuenta la dirección de retorno)
029af304 77584fca ntdll!NtQueryInformationProcess+0xa # dirección de retorno
029af308 775a9628 nt!Excepción Rt!Dispatch+0x7c
029af30c ffffffff
029af310 00000022
029af314 029af348
1: kd>u77584fca

ntdll!NtQueryInformationProcess+0xa:
77584fca retirado 14h          # Esta es la línea 4 del último fragmento.
1: kd> u @edx                # SYSEXIT establecerá EIP en EDX
ntdll!KiFastSystemCallRet:
77586954 retirado            # volver a 77584fca

```

Después de ejecutar KiFastSystemCallRet (que solo tiene una instrucción: RET), regresa a NtQueryInformationProcess.

Es instructivo comparar la implementación de SYSENTER en Windows 7 y 8. Se le pedirá que haga esto como ejercicio.

Windows en ARM utiliza la instrucción SVC para implementar llamadas del sistema. En documentación anterior, se puede hacer referencia a SVC como SWI, pero son el mismo código de operación. Recuerde que ARM no tiene un IDT como x86/x64 pero su tabla de vectores de excepciones tiene una funcionalidad similar:

.text:004D0E00 Vectores de excepción KiArm	
.text:004D0E00 LDR W	PC, =0xFFFFFFFF
.text:004D0E04 LDR W	PC, =(KiUndefinedInstructionException+1)
.text:004D0E08 LDR W	PC, =(Excepción KisWI+1)
.text:004D0E0C LDR.W	PC, =(KiPrefetchAbortException+1)
.text:004D0E10 LDR W	PC, =(Excepción de cancelación de KiData+1)
.text:004D0E14 LDR W	PC, =0xFFFFFFFF
.text:004D0E18 LDR W	PC, =(KiInterruptException+1)
.text:004D0E1C LDR.W	PC, =(Excepción KiFIQ+1)

Siempre que se ejecuta la instrucción SVC , el procesador cambia al modo supervisor y llama a KisWIException para manejar la excepción. Esta función puede considerarse como la equivalencia ARM de KiSystemCall64 en x64. Nuevamente, para comprender todo el proceso de llamada del sistema en ARM, considere la función de modo usuario ntdll!NtQueryInformationProcess:

01: Proceso de información de NtQuery		
02: MOVIMIENTO.W	R12, #0x17	; NtQueryInformationProcess
03: SVC	1	
04: BX	LR	

El número de llamada del sistema se coloca primero en R12 y luego en SVC. Cuando se coloca SVC ejecutado, entra en el manejador KisWIException:

01: KisWIException 02: marco	
trampa = -0x1A0	
03: SUB	SP, SP, #0x1A0
04: STR W	R0, R1, [SP,#0x1A0+trampa_R0]

```

05: STR.W          R2, R3, [SP,#0x1A0+trampa_R2]
06: STR.W          R12, [SP,#0x1A0+trampa_R12]
07: STR.W          R11, [SP,#0x1A0+trampa_R11]
08: ORR.W
09: MOVIMIENTOS
10: MRC
11: STR.W          LR, R0, [SP,#0x1A0+trapframe_Pc] ; LR es la dirección de retorno después de la

; Instrucción SVC.
; guardado aquí para que
; el sistema sabe dónde
; volver después de la
; la llamada al sistema está hecha

12: LDRB.W         R1, [R12,#_ETHREAD Tcb.Header.___u0.__s3.DebugActive]
13: MOVIMIENTOS
14: STR             R3, n.º 2
15: AGREGAR.W      R3, [SP,#0x1A0+trapframe.Excepción activa]
16: CMP
17: BNE             R1, #0
18: ubicación_4D00D0
19: MRC
20: MOVIMIENTOS
21: TST.W          R0, #0xF00000
22: ecualizador
23: AGREGAR
24: VMRS
25: AGREGAR
26: STR             R1, SP, #tamaño(_KTRAP_FRAME)
27: VSTMIA
28: BIC.W          R2, R2, #0x370000
29: VMSR
30:
31: loc_4D00F2
32: STR             R1, [SP,#0x1A0+trapframe.VfpState]
33: LDR             R0, [SP,#0x1A0+trapframe_R12] ; recuperar guardado

llamada al sistema
; desde la línea 6

34: LDR             R1, [SP,#0x1A0+trampa_R0]
35: movimiento
36: CPS.W           R2, SP
37: FUERZA
38: FUERZA
39: CPSIE.W         Yo, #0x13
40: STR.W           R0, R1, [R12,#_ETHREAD.Tcb.NúmeroDeLlamadaDelSistema]
; escribe syscall# en el
; hilo

41: MRC
42: BFC.W           p15, 0, R0, c13, c0, 4
43: LDR.W           R0, #0xC
44: movimiento
45: movimiento
46: CMP             R1, n.º 4

```

```

47: C.B.S.           ubicación_4D0178
48:
49:
50: ubicación_4D0178
51: MRC
52: LDR.W          p15, 0, R12, c13, c0, 3
53: BL             R0, [R12,#_ETHREAD.Tcb.NúmeroDeLlamadaDelSistema]
54: B              KiSystemService; envía la llamada del sistema
                  KiSystemServiceSalir ; volver al modo usuario

```

Esta función hace muchas cosas, pero los puntos principales son que construye un marco de trampa (nt!_KTRAP_FRAME) para guardar algunos registros, guarda la dirección de retorno del modo usuario (SVC coloca automáticamente la dirección de retorno en LR), guarda el número de llamada del sistema en el objeto de hilo actual y envía la llamada del sistema (el mismo mecanismo que x64). El regreso al modo usuario se realiza a través de KiSystemServiceExit:

```

01: KiSystemServiceSalir
02: ...
03: BIC.W          R0, R0, #1
04: MOV            R3, SP
05: AGREGAR        SP, SP, #0x1A0
06: CPS.W          #0x1F
07: LDR.W          SP, [R3,#_MARCO_TRAMPA_K_.Sp]
08: LDRD.W         LR, R11, [R3,#_MARCO_TRAMPA_K_.Lr]
09: CPS.W          #0x12
10: STR.W          R0, R1, [SP]
11: LDR            R0, [R3,#_MARCO_TRAMPA_K_.R0]
12: MOVIMIENTOS   R1, #0
13: MOVIMIENTOS   R2, #0
14: MOVIMIENTOS   R3, #0
15: movimiento     R12, R1
16: RFEFD.W        ES      ; volver al modo usuario

```

Nivel de solicitud de interrupción

El núcleo de Windows utiliza un concepto abstracto llamado nivel de solicitud de interrupción (IRQL) para gestionar la capacidad de interrupción del sistema. Las interrupciones se pueden dividir en dos categorías generales: software y hardware. Las interrupciones de software son eventos sincrónicos que se activan por condiciones en el código en ejecución (división por 0, ejecución de una instrucción INT , fallo de página, etc.); las interrupciones de hardware son eventos asincrónicos que se activan por dispositivos conectados a la CPU. Las interrupciones de hardware son asincrónicas porque pueden ocurrir en cualquier momento; se utilizan normalmente para indicar operaciones de E/S al procesador. Los detalles de cómo funcionan las interrupciones de hardware son específicos del hardware y, por tanto, se abstraen mediante el componente de capa de abstracción de hardware (HAL) de Windows.

En términos concretos, un IRQL es simplemente un número (definido por el tipo KIRQL, que en realidad es un UCHAR) asignado a un procesador. Windows asocia un IRQL

con una interrupción y define el orden en el que se maneja. El número exacto asociado con cada IRQL puede variar de una plataforma a otra, por lo que haremos referencia a ellos solo por su nombre. La regla general es que las interrupciones en IRQL X enmascararán todas las interrupciones que sean menores que X. Una vez que se maneja la interrupción, el núcleo reduce el IRQL para que pueda ejecutar otras tareas. Debido a que IRQL es un valor por procesador, varios procesadores pueden operar simultáneamente en diferentes IRQL.

Existen varios IRQL diferentes, pero los más importantes a recordar son los siguientes:

- **NIVEL PASIVO (0):** este es el IRQL más bajo del sistema. Todos los usuarios El código de modo y la mayor parte del código del kernel se ejecutan en este IRQL.
- **NIVEL APC (1) :** este es el IRQL en el que se ejecutan las llamadas a procedimientos asíncronos (APC). (Consulte la sección “Llamadas a procedimientos asíncronos”).
- **NIVEL DE DESPACHO (2) —**Este es el IRQL de software más alto del sistema. El despachador de subprocessos y las llamadas a procedimientos diferidos (DPC) se ejecutan en este IRQL. (Consulte la sección “Llamadas a procedimientos diferidos”). El código en este IRQL no puede esperar.

NOTA: Los IRQL superiores a DISPATCH_LEVEL suelen estar asociados con interrupciones de hardware reales o mecanismos de sincronización de nivel extremadamente bajo. Por ejemplo, IPI_LEVEL se utiliza para la comunicación entre procesadores.

Aunque parezca que IRQL es una propiedad de programación de subprocessos, no lo es. Es una propiedad del procesador, mientras que la prioridad del hilo es una propiedad por hilo.

Debido a que IRQL es una abstracción de software de la prioridad de las interrupciones, la implementación subyacente tiene una correlación directa con el hardware. Por ejemplo, en x86/x64, el controlador de interrupciones local (LAPIC) en el procesador tiene un registro de prioridad de tareas programable (TPR) y un registro de prioridad de procesador de solo lectura (PPR). El TPR determina la prioridad de las interrupciones; el PPR representa la prioridad de las interrupciones actual. El procesador entregará solo interrupciones cuya prioridad sea mayor que el PPR. En términos prácticos, cuando Windows necesita cambiar la prioridad de las interrupciones, llama a las funciones del núcleo KeRaiseIrql/KeLowerIrql, que programan el TPR en el APIC local. Esto se puede observar en la definición en x64 (en x64, CR8 es un registro de sombra que permite un acceso rápido al TPR de LAPIC; los sistemas x86 deben programar el LAPIC para establecer el TPR):

```
KeRaiseIrql
01: KzRaiseIrql proc cerca
02: movimiento      Rax, CR8
03: movzx ecx, c
04: movimiento      CR8, RCX
05: retornado
06: Fin de KzRaiseIrql
```

KeLowerIrql

```
01: procedimiento KzLowerIrql cerca
02: movzx eax, c
03: movimiento      CR8, rax
04: retornado
05: Fin de KzLowerIrql
```

Los conceptos anteriores explican por qué el código que se ejecuta en IRQL alto no se puede preemtpado por código en IRQL inferior.

Memoria de piscina

De manera similar a las aplicaciones en modo usuario, el código en modo kernel puede asignar memoria en tiempo de ejecución. El nombre general para esto es memoria de pool; uno puede pensar en ella como el montón en modo usuario. La memoria de pool generalmente se divide en dos tipos: pool paginado y pool no paginado. La memoria de pool paginado es memoria que puede ser paginada en cualquier momento por el administrador de memoria. Cuando el código en modo kernel toca un buffer que está paginado, desencadena una excepción de fallo de página que hace que el administrador de memoria pagina ese buffer desde el disco. La memoria de pool no paginado es memoria que nunca puede ser paginada; en otras palabras, acceder a dicha memoria nunca desencadena un fallo de página.

Esta distinción es importante porque tiene consecuencias para el código que se ejecuta en niveles IRQL altos. Supongamos que un subproceso del núcleo se está ejecutando actualmente en DISPATCH_LEVEL y hace referencia a la memoria que se ha paginado y que necesita ser manejada por el manejador de errores de página; debido a que el manejador de errores de página (ver MmAccessFault) necesita emitir una solicitud para traer la página desde el disco y el despachador de subprocesos se ejecuta en DISPATCH_LEVEL, no puede resolver la excepción y da como resultado una comprobación de errores. Esta es una de las razones por las que el código que se ejecuta en DISPATCH_LEVEL solo debe residir y acceder a la memoria del grupo no paginado.

La memoria del pool se asigna y libera mediante ExAllocatePool* y ExFreePool* Familia de funciones. De manera predeterminada, la memoria de grupo no paginado (tipo NonPagedPool) se asigna con permisos de lectura, escritura y ejecución en x86/x64, pero no ejecutable en ARM; en Windows 8, se puede solicitar memoria de grupo no paginado y no ejecutable especificando el tipo de grupo NonPagedPoolNX . La memoria de grupo paginado se asigna con permisos de lectura, escritura y ejecución en x86, pero no ejecutable en x64/ARM.

Listas de descriptores de memoria

Una lista de descriptores de memoria (MDL) es una estructura de datos que se utiliza para describir un conjunto de páginas físicas asignadas a una dirección virtual. Cada entrada de la MDL describe un búfer contiguo y se pueden vincular varias entradas entre sí. Una vez que se crea una MDL para un búfer existente, las páginas físicas se pueden bloquear en la memoria (es decir, no se reutilizarán) y se pueden asignar a otra dirección virtual.

Para que sean útiles, los MDL deben inicializarse, probarse, bloquearse y luego asignarse.

Para comprender mejor el concepto, consideremos algunos de los usos prácticos de los MDL.

Supongamos que un controlador necesita asignar una parte de la memoria en el espacio del núcleo al espacio de direcciones del modo de usuario de un proceso o viceversa. Para lograrlo, primero inicializaría un MDL para describir el búfer de memoria (`IoAllocateMdl`), se aseguraría de que el subproceso actual tenga acceso a esas páginas y las bloquearía (`MmProbeAndLockPages`) y, luego, asignaría esas páginas en la memoria (`MmMapLockedPagesSpecifyCache`) en ese proceso.

Otro escenario es cuando un controlador necesita escribir en algunas páginas de solo lectura (como las de la sección de código). Una forma de lograrlo es a través de MDL.

El controlador inicializaría el MDL, lo bloquearía y luego lo asignaría a otra dirección virtual con permiso de escritura. En este escenario, el controlador puede usar MDL para implementar una función similar a `VirtualProtect` en modo kernel.

Procesos y subprocesos

Un hilo se define mediante dos estructuras de datos del núcleo: ETHREAD y KTHREAD. Una estructura ETHREAD contiene información de mantenimiento sobre el hilo (es decir, identificación del hilo, proceso asociado, depuración habilitada/deshabilitada, etc.). Una estructura KTHREAD almacena información de programación para el despachador de subprocesos, como información de la pila de subprocesos, procesador en el que ejecutar, estado de alerta, etc. Un ETHREAD contiene un KTHREAD.

El programador de Windows opera en subprocesos.

Un proceso contiene al menos un subproceso y está definido por dos estructuras de datos del núcleo: EPROCESS y KPROCESS. Una estructura EPROCESS almacena información básica sobre el proceso (es decir, identificación del proceso, token de seguridad, lista de subprocesos, etc.). Una estructura KPROCESS almacena información de programación para el proceso (es decir, tabla de directorio de páginas, procesador ideal, tiempo de sistema/usuario, etc.). Un EPROCESS contiene un KPROCESS. Al igual que ETHREAD y KTHREAD, estas estructuras de datos también son opacas y solo se debe acceder a ellas con rutinas de kernel documentadas. Sin embargo, puede ver sus miembros de campo a través del depurador de kernel, de la siguiente manera:

Procesos

```
kd> dt nt!_EPROCESSO
+0x000 Placa de circuito impreso : _KPROCESO
+0x2c8 Bloqueo de proceso : _EX_PUSH_LOCK
+0x2d0 Hora de creación : _ENTERO_GRANDE
+0x2d8 Protección de ejecución: _EX_RUNDOWN_REF
+0x2e0 UniqueProcessId : Ptr64 Vacío
+0x2e8 Enlaces de proceso activos: _LIST_ENTRY
+0x2f0 Banderas2 : Uint4B
+0x2f8 TrabajoNoRealmenteActivo: Pos 0, 1 Bit
+0x2f8 Contabilidad doblada: Pos 1, 1 bit
+0x2f8 Nuevo proceso informado: Pos 2, 1 bit
...
```

```
+0x3d0 InheritedFromUniqueProcessId : Ptr64 Vacío
+0x3d8 LdtInformación: Ptr64 Vacío
+0x3e0 Proceso de creación: Ptr64 _EPROCESS
+0x3e0 Proceso del host de consola: UInt8B
+0x3e8 Peb :Ptr64 _PEB
+0x3f0 Sesión :Ptr64 Vacío
...
0: kd> dt ntl!KPROCESS
Encabezado +0x000 :_ENCABEZADO_DE_DESPACHO
+0x018 Encabezado de lista de perfiles: _LIST_ENTRY
+0x028 DirectorioTableBase : UInt8B
+0x030 Encabezado de lista de subprocessos: _LIST_ENTRY
+0x040 Bloqueo de proceso :UInt4B
...
+0x0f0 Cabeza de lista lista :_ENTRADA_DE_LISTA
+0x100 Entrada de lista de intercambio :_ENTRADA_DE_LISTA_ÚNICA
+0x108 Procesadores activos: _KAFFINITY_EX
...
```

Trapos

```
0: kd> dt ntl!HILO
+0x000 Por confirmar :_HILO
+0x348 Hora de creación :_ENTERO_GRANDE
+0x350 Hora de salida :_ENTERO_GRANDE
...
+0x380 Bloqueo de lista de temporizadores activos: UInt8B
+0x388 Encabezado de lista del temporizador activo: _LIST_ENTRY
+0x398 Cid :_ID_CLIENTE
...
0: kd> dt ntl!KTHREAD
Encabezado +0x000 :_ENCABEZADO_DE_DESPACHO
+0x018 SListFaultAddress: Ptr64 nulo
+0x020 Objetivo cuántico :UInt8B
+0x028 Pila inicial :Ptr64 Vacío
+0x030 Límite de pila :Ptr64 Vacío
+0x038 Base de pila :Ptr64 Vacío
+0x040 Bloqueo de hilo :UInt8B
...
+0xd8 Entrada de lista de espera :_ENTRADA_DE_LISTA
+0xd8 Entrada de lista de intercambio :_ENTRADA_DE_LISTA_ÚNICA
+0xe8 Cola :Ptr64 _QUEUE
+0xf0 Teb :Ptr64 Vacío
```

NOTA: Aunque decimos que sólo se debe acceder a estos procesos con rutinas de kernel documentadas, los rootkits del mundo real modifican campos semidocumentados o completamente indocumentados en estas estructuras para lograr sus objetivos. Por ejemplo, una forma de ocultar un proceso es eliminarlo del campo ActiveProcessLinks en la estructura EPROCESS. Sin embargo, debido a que son opacos y no documentados, los desplazamientos de campo pueden cambiar (y lo hacen) de una versión a otra.

También existen estructuras de datos de modo de usuario análogas que almacenan información sobre procesos e hilos. Para los procesos, existe el bloque de entorno de proceso (PEB/ntdll!_PEB), que almacena información básica como la dirección de carga base, los módulos cargados, los montones de procesos, etc. Para los subprocesos, existe el bloque de entorno de subprocesos (TEB/ntdll!_TEB), que almacena datos de programación de subprocesos e información para el proceso asociado. El código de modo de usuario siempre puede acceder al TEB a través del segmento FS (x86), GS (x64) o coprocesador 15 (ARM). Con frecuencia verá que el código del sistema accede a estos objetos, por lo que se enumeran aquí:

Hilo actual (modo kernel)

x86	
movimiento	eax, fs grande: 124 h
x64	
movimiento	rax, gs:188h
BRAZO	
CMR	p15, 0, R3, c13, c0, 3
BICS.W	R0, R3, #0x3F
TEB (Modo de usuario)	
x86	
movimiento	edx, gran fs:18h
x64	
movimiento	rax, gs:30h
BRAZO	
CMR	p15, 0, R4, c13, c0, 2

Contexto de ejecución

Cada subproceso en ejecución tiene un contexto de ejecución. Un contexto de ejecución contiene el espacio de direcciones, el token de seguridad y otras propiedades importantes del subproceso en ejecución. En cualquier momento, Windows tiene cientos de subprocesos ejecutándose en diferentes contextos de ejecución. Desde una perspectiva del núcleo, se pueden definir tres contextos de ejecución generales :

- **Contexto del hilo** : contexto de un hilo específico (o, por lo general, el hilo solicitante en el caso de un hilo de modo usuario que solicita un servicio del núcleo)
- **Contexto del sistema** : contexto de un hilo que se ejecuta en el proceso del sistema
- **Contexto arbitrario** : contexto de cualquier hilo que se estuviera ejecutando antes de que el programador tomara el control

Recuerde que cada proceso tiene su propio espacio de direcciones. Mientras está en modo kernel, es importante saber en qué contexto se está ejecutando su código porque eso determina el espacio de direcciones en el que se encuentra y los privilegios de seguridad que posee.

No existe una lista de reglas para determinar con precisión el contexto de ejecución en un escenario determinado, pero los siguientes consejos generales pueden ayudar:

- Cuando se carga un controlador, su punto de entrada (DriverEntry) se ejecuta en el sistema. contexto.
- Cuando una aplicación en modo usuario envía una solicitud (IOCTL) a un controlador, el controlador IOCTL del controlador se ejecuta en el contexto del hilo (es decir, el contexto del hilo en modo usuario que inició la solicitud).
- Los APC se ejecutan en el contexto del hilo (es decir, el contexto del hilo en el que se encuentran). APC estaba en cola).
- Los DPC y los temporizadores se ejecutan en un contexto arbitrario.
- Los elementos de trabajo se ejecutan en el contexto del sistema.
- Los subprocessos del sistema se ejecutan en el contexto del sistema si el parámetro ProcessHandle es NULL (caso común).

Por ejemplo, el punto de entrada de un controlador solo tiene acceso al espacio de direcciones del proceso del sistema y, por lo tanto, no puede acceder a ningún otro espacio de proceso sin provocar una violación de acceso. Si un subprocesso en modo kernel desea cambiar su contexto de ejecución a otro proceso, puede utilizar la API documentada KeStackAttachProcess.

Esto es útil cuando un controlador necesita leer/escribir la memoria de un proceso específico.

Primitivas de sincronización del núcleo

El núcleo proporciona primitivas de sincronización comunes que pueden utilizar otros componentes. Las más comunes son eventos, bloqueos de giro, mutexes, bloqueos de recursos y temporizadores. En esta sección se explica su interfaz y se analiza su uso.

Los objetos de evento se utilizan para indicar el estado de una operación. Por ejemplo, cuando el sistema se está quedando sin memoria de grupo no paginado, el núcleo puede notificar a un controlador a través de eventos. Un evento puede estar en uno de dos estados: señalado o no señalado. El significado de señalado y no señalado depende del escenario de uso. Internamente, un evento es un objeto definido por la estructura KEVENT e inicializado por la API KeInitializeEvent . Después de inicializar el evento, un subprocesso puede esperarlo con KeWaitForSingleObject o KeWaitForMultipleObjects.

Los eventos se utilizan comúnmente en los controladores para notificar a otros subprocessos que algo terminó de procesarse o que se cumplió una condición particular.

Los temporizadores se utilizan para indicar que ha transcurrido un cierto intervalo de tiempo. Por ejemplo, cada vez que entramos en un nuevo siglo, el núcleo ejecuta un código para actualizar la hora; el mecanismo subyacente para esto son los temporizadores. Internamente, los objetos temporizadores están definidos por la estructura KTIMER y se inicializan mediante KeInitializeTimer/Ex

Rutina. Al inicializar los temporizadores, se puede especificar una rutina DPC opcional que se ejecutará cuando expiren. Por definición, cada procesador tiene su propio temporizador.

cola; específicamente, el campo TimerTable en el PRCB es una lista de temporizadores para ese procesador en particular. Los temporizadores se utilizan comúnmente para hacer algo de manera periódica o específica en el tiempo. Tanto los temporizadores como los DPC se tratan con más detalle más adelante en este capítulo.

Los mutex se utilizan para el acceso exclusivo a un recurso compartido. Por ejemplo, si dos subprocessos modifican simultáneamente una lista enlazada compartida sin un mutex, pueden dañar la lista; la solución es acceder únicamente a la lista enlazada mientras se mantiene un mutex. Si bien la semántica básica de un mutex no cambia, el núcleo de Windows ofrece dos tipos diferentes de mutex: mutex protegido y mutex rápido.

Los mutex protegidos son más rápidos que los mutex rápidos, pero solo están disponibles en Windows 2003 y versiones posteriores. Internamente, un mutex se define mediante un **FAST_MUTEX** o estructura **GUARDED_MUTEX** e inicializada por **ExInitialize{Fast,Guarded}Mutex**. Después de la inicialización, se pueden adquirir y liberar a través de diferentes API; consulte la documentación del kit de controladores de Windows para obtener más información.

Los bloqueos de giro también se utilizan para el acceso exclusivo a un recurso compartido. Si bien son conceptualmente similares a los mutex, se utilizan para proteger recursos compartidos a los que se accede en DISPATCH_LEVEL o IRQL superior. Por ejemplo, el núcleo adquiere un bloqueo de giro antes de modificar estructuras de datos globales críticas, como la lista de procesos activos; debe hacerlo porque en un sistema multiprocesador, varios subprocessos pueden acceder y modificar la lista al mismo tiempo. Internamente, los bloqueos de giro se definen mediante la estructura **KSPIN_LOCK** y se inicializan con **KInitializeSpinLock**. Después de la inicialización, se pueden adquirir o liberar a través de varias API documentadas; consulte la documentación de WDK para obtener más información. Tenga en cuenta que el código que retiene un bloqueo de giro se ejecuta en DISPATCH_LEVEL o superior; por lo tanto, el código que se ejecuta y la memoria que toca siempre deben ser residentes.

Liza

Las listas enlazadas son los bloques de construcción fundamentales de las estructuras de datos dinámicas en el núcleo y los controladores. Muchas estructuras de datos importantes del núcleo (como las relacionadas con procesos e hilos) se construyen sobre listas. De hecho, las listas se usan tan comúnmente que el WDK proporciona un conjunto de funciones para crearlas y manipularlas de una manera genérica. Aunque las listas son conceptualmente simples y no tienen una relación directa con la comprensión de los conceptos del núcleo o la práctica de la ingeniería inversa, se introducen aquí por dos razones importantes. Primero, se usan en prácticamente todas las estructuras de datos del núcleo de Windows analizadas en este capítulo. El núcleo opera comúnmente sobre entradas de varias listas (es decir, lista de módulos cargados, lista de procesos activos, lista de hilos en espera, etc.) contenidas en estas estructuras, por lo que es importante comprender la mecánica de tales operaciones. Segundo, mientras que las funciones que operan sobre listas, por ejemplo, **InsertHeadList**,

InsertTailList, RemoveHeadList, RemoveEntryList, etc., aparecen en formato fuente en los encabezados de WDK, siempre están insertadas en línea por el compilador y, en consecuencia, nunca aparecerán como "funciones" en el nivel de ensamblaje en binarios de la vida real; en otras palabras, nunca aparecerán como destino de llamada o rama . Por lo tanto, debe comprender los detalles de implementación y los patrones de uso para poder reconocerlas en el nivel de ensamblaje.

Detalles de implementación

El WDK proporciona funciones que admiten tres tipos de listas:

- Lista enlazada simplemente: una lista cuyas entradas están enlazadas entre sí con un puntero (Siguiente).
- Lista enlazada simple secuenciada: una lista enlazada simple con soporte para operaciones atómicas. Por ejemplo, puede eliminar la primera entrada de la lista sin preocuparse por adquirir un bloqueo. ■
- Lista enlazada doble circular: una lista cuyas entradas están enlazadas entre sí con dos punteros, uno que apunta a la siguiente entrada (Flink) y otro que apunta a la entrada anterior (Blink).

Los tres son conceptualmente idénticos en términos de uso a nivel de código fuente. Este capítulo cubre sólo las listas doblemente enlazadas porque son las más comunes. En uno de los ejercicios, se le pedirá que revise la documentación de WDK sobre operaciones de lista y escriba un controlador que utilice los tres tipos de listas.

La implementación se construye sobre una estructura:

```
estructura de tipo definido _LIST_ENTRY {
    estructura _LIST_ENTRY *Flink;
    estructura _LIST_ENTRY *Parpadeo;
} LISTA_ENTRADA, *PLIST_ENTRADA;
```

Una LIST_ENTRY puede representar un encabezado de lista o una entrada de lista. Un encabezado de lista representa el "encabezado" de la lista y normalmente no almacena ningún dato excepto el LIST_ENTRY en sí misma; todas las funciones de lista requieren un puntero al encabezado de la lista. Una entrada de lista es la entrada real que almacena los datos; en la vida real, es una estructura LIST_ENTRY incrustada dentro de una estructura más grande.

Las listas deben inicializarse con InitializeListHead antes de su uso. Esta función simplemente establece los campos Flink y Blink para que apunten al encabezado de la lista. Su código se muestra a continuación y se ilustra en la Figura 3-4:

```
VOID InitializeListHead(PLIST_ENTRY ListHead) {
    Cabeza de lista->Flink = Cabeza de lista->Blink = Cabeza de lista;
    devolver;
}
```

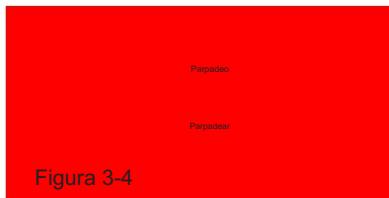


Figura 3-4

En formato de ensamblado, esto se traduciría en tres instrucciones: una para recuperar ListHead y dos para completar los punteros Flink y Blink . Consideré cómo se manifiesta InitializeListHead en el ensamblado x86, x64 y ARM:

x86	
pasto	eax, [esi+2Ch]
movimiento	[eax+4], eax
movimiento	[eax], eax
x64	
pasto	r11, [rbx+48h]
movimiento	[r11+8], r11
movimiento	[r11], r11
BRAZO	
AGREGA W	R3, R4, #0x2C
STR	R3, [R3,#4]
STR	R3, [R3]

En los tres casos, se utilizan el mismo puntero y registro en operaciones de solo escritura . Otra observación clave es que las escrituras se realizan en los desplazamientos +0 y +4/8 desde el registro base; estos desplazamientos corresponden a los punteros Flink y Blink en la estructura. Siempre que vea este patrón de código, debe pensar en listas.

Después de inicializar la lista, se pueden insertar entradas al principio o al final. Como se mencionó anteriormente, una entrada de lista es simplemente una LIST_ENTRY dentro de una estructura más grande ; por ejemplo, la estructura del núcleo KDPC (que se analiza más adelante en este capítulo) tiene un campo DpcListEntry :

C Definición

```
estructura de tipo definido _KDPC {
    Tipo UCHAR;
    Importancia de UCHAR;
    Número USHORT volátil;
    ENTRADA_LISTA DpcListEntry;
    PKDEFERRED_ROUTINE Rutina diferida;
    PVOID Contexto diferido;
    PVOID Argumento del sistema1;
    PVOID Argumento del sistema2;
    __volátil PVOID DpcData;
}KDPC, *PKDPC, *PRKDPC;
```

```
x64

0: kd>dt nt!_KDPC
+0x000 Tipo :UCar
+0x001 Importancia :UCar
+0x002 Número :UInt2B
+0x008 Entrada de lista Dpc :_ENTRADA_DE_LISTA
+0x018 Rutina diferida: Ptr64 vacío
+0x020 DeferredContext : Ptr64 Vacío
+0x028 Argumento del sistema1: Ptr64 nulo
+0x030 Argumento del sistema2: Ptr64 nulo
+0x038 Datos de Dpc :Ptr64 Vacío
```

Supongamos que tiene una lista con una entrada KDPC, como se muestra en la Figura 3-5.

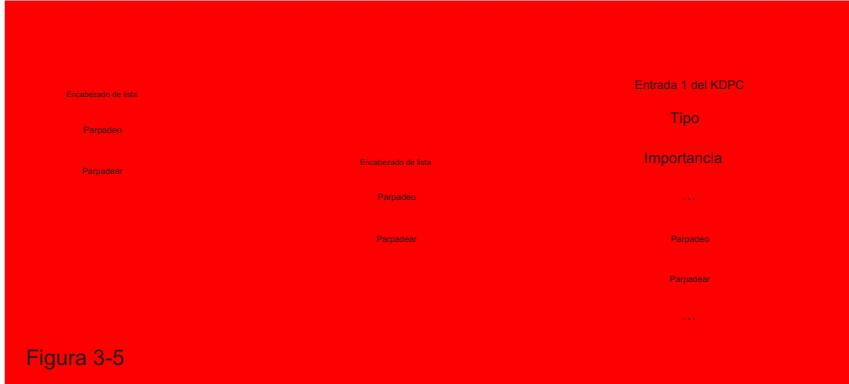


Figura 3-5

La inserción se realiza con InsertHeadList e InsertTailList. Tenga en cuenta lo siguiente: inserción de una entrada en el encabezado, como se muestra en la Figura 3-6.



El código fuente de estas rutinas y cómo pueden manifestarse en formato de ensamblaje se muestran aquí:

NOTA: Estos fragmentos se extraen de la función del núcleo KeInsertQueueDpc en Windows 8, con un par de líneas eliminadas para mayor claridad. El objetivo aquí es observar cómo se inserta la nueva entrada en la lista. La programación de instrucciones puede cambiar el orden de algunas instrucciones, pero serán en su mayoría las mismas.

```
InsertarListaDeEncabezados
do

VOID InsertarEncabezadoLista(PLIST_ENTRY EncabezadoLista, PLIST_ENTRY Entrada) {
    PLIST_ENTRY Enlace;
    Flink = ListaEncabezado->Flink;
    Entrada->Flink = Flink;
    Entrada->Blink = ListHead;
    Parpadeo->Parpadeo = Entrada;
    ListHead->Flink = Entrada;
    devolver;
}

BRAZO
LDR      R1, [R5]
STR      R5, [R2,#4]
STR      R1, [R2]
STR      R2, [R1,#4]
STR      R2, [R5]

x86
movimiento edx, [ebx]
movimiento [ecx], edx
movimiento [ecx+4], ebx
movimiento [edx+4], ecx
movimiento [ebx], ecx

x64
movimiento rcx, [rdi]
movimiento [rax+8], rdi
movimiento [rax], rcx
movimiento [rcx+8], rax
movimiento [rdi], rax
```

```
InsertarListaDeCola
do

VOID InsertTailList(PLIST_ENTRY ListHead, PLIST_ENTRY Entrada) {
    PLIST_ENTRY Parpadeo;
    Parpadeo = ListHead->Parpadeo;
    Entrada->Flink = ListHead;
    Entrada->Parpadeo = Parpadeo;
    Parpadeo->Flink = Entrada;
    ListHead->Blink = Entrada;
    devolver;
}
```

```

BRAZO

LDR      R1, [R5,#4]
STR      R5, [R2]
STR      R1, [R2,#4]
STR      R2, [R1]
STR      R2, [R5,#4]

x86

movimiento    ecx, [ebx+4]
movimiento    [eax], ebx
movimiento    [eax+4], ecx
movimiento    [ecx], eax
movimiento    [ebx+4], eje

x64

movimiento    rcx, [rdi+8]
movimiento    [rax], rdi
movimiento    [rax+8], rcx
movimiento    [rcx], rax
movimiento    [rdi+8], rax

```

En los fragmentos anteriores, R5/EBX/RDI apuntan a ListHead y R2/ECX/RAX

apunta a la entrada.

La eliminación se realiza con RemoveHeadList, RemoveTailList y RemoveEntryList. Estas rutinas suelen estar precedidas por la función IsListEmpty , que simplemente verifica si el Flink del encabezado de la lista apunta a sí mismo:

```

EstáListaVacia

do

    BOOLEAN IsListEmpty(PLIST_ENTRY Encabezado de lista) {
        devuelve (BOOLEAN)(ListHead->Flink == ListHead);
    }

BRAZO

LDR2, [R4]
CMP-R2, R4

x86

movimiento eax, [esi]
cmp eax, esi

x64

movimiento rax, [rbx]
cmp rax, rbx

```

Eliminar lista de encabezados

```

do

    PLIST_ENTRY RemoveHeadList(PLIST_ENTRY ListaEncabezado) {
        PLIST_ENTRY Enlace;

```

```
PLIST_ENTRY Entrada;
Entrada = ListHead->Flink;
Flink = Entrada->Flink;
ListaEncabezado->Flink = Flink;
Parpadeo->Parpadeo = ListHead;
retorno Entrada;
}

BRAZO

LDR2, [R4]
LDR-R1, [R2]
Fuerza R1, [R4]
Fuerza R4, [R1,#4]

x86

movimiento eax, [esi]
mov ecx, [eax] mov
[esi], ecx mov [ecx+4],
esi

x64

mov rax, [rbx] mov
rcx, [rax] mov [rbx],
rcx mov [rcx+8], rbx
```

```
Eliminar lista de cola

do

PLIST_ENTRY RemoveTailList(PLIST_ENTRY Encabezado de lista) {
    PLIST_ENTRY Parpadeo;
    PLIST_ENTRY Entrada;
    Entrada = ListHead->Blink;
    Parpadeo = Entrada->Parpadeo;
    ListHead->Blink = Parpadeo;
    Parpadeo->Flink = ListHead;
    retorno Entrada;
}

BRAZO

LDR R6, [R5,#4]
LDR2, [R6,#4]
Fuerza R2, [R5,#4]
Fuerza R5, [R2]

x86

movimiento ebx, [edi+4]
movimiento eax, [ebx+4]
movimiento [edi+4], eax
movimiento [eax], edi
```

x64

```
movimiento rsi, [rdi+8]
movimiento de rax, [rsi+8]
movimiento [rdi+8], rax
movimiento [rax], rdi
```

Eliminar lista de entradas

do

```
BOOLEAN RemoveEntryList(PLIST_ENTRY Entrada){
    PLIST_ENTRY Parpadeo;
    PLIST_ENTRY Enlace;
    Flink = Entrada->Flink;
    Parpadeo = Entrada->Parpadeo;
    Parpadeo->Parpadeo = Parpadeo;
    Parpadeo->Parpadeo = Parpadeo;
    devuelve (BOOLEAN)(Flink == Parpadeo);
}
```

BRAZO

```
Distancia de separación R1[R0]
Distancia de seguridad R2,[R0,#4]
Fuerza R1,[R2]
Fuerza R2,[R1,#4]
```

x86

```
movimiento edx, [ecx]
movimiento eax, [ecx+4]
movimiento [eax], edx
movimiento [edx+4], eax
```

x64

```
movimiento rdx, [rcx]
movimiento rax, [rcx+8]
movimiento [rax], rdx
movimiento [rdx+8], rax
```

Tenga en cuenta que todas las funciones de manipulación de listas operan únicamente en la estructura LIST_ENTRY . Para poder hacer cosas útiles con una entrada de lista, el código debe manipular los datos reales de la entrada. ¿Cómo acceden los programas a los campos de una entrada de lista? Esto se hace con la macro CONTAINING_RECORD :

```
#define CONTAINING_RECORD(dirección, tipo, campo) ((tipo *)( \
    (PCHAR)(dirección) - \
    (ULONG_PTR)&((tipo \
    *)0)->campo)))
```

CONTAINING_RECORD devuelve la dirección base de una estructura mediante el siguiente método: Calcula el desplazamiento de un campo en una estructura convirtiendo el puntero de la estructura en 0 y luego resta ese valor de la dirección real del campo. En la práctica, esta macro generalmente toma la dirección del campo LIST_ENTRY en la entrada de la lista.

el tipo de entrada de la lista y el nombre de ese campo. Por ejemplo, supongamos que tiene una lista de entradas de KDPC (consulte la definición anterior) y desea que una función acceda al campo DeferredRoutine ; el código sería el siguiente:

```
PKDEFERRED_ROUTINE Rutina diferida de entrada de lectura (entrada PLIST_ENTRY) {
    PKDPC pág.
    p = CONTAINING_RECORD(entrada, KDPC, DpcListEntry);
    devolver p->RutinaDiferida;
}
```

Esta macro se utiliza comúnmente inmediatamente después de llamar a una de las rutinas de eliminación de lista o durante la enumeración de entradas de lista.

Tutorial paso a paso

Después de haber analizado los conceptos y los detalles de implementación de las funciones de manipulación de listas en modo kernel, ahora los aplicaremos al análisis de la Muestra C. Este tutorial tiene tres objetivos:

- Mostrar un uso común de listas en un controlador/rootkit de la vida real
- Demostrar las incertidumbres a las que se enfrenta un ingeniero inverso en la práctica
- Analice los problemas de las estructuras no documentadas y las compensaciones codificadas.

Este controlador hace muchas cosas, pero solo nos interesan dos funciones: sub_11553 y sub_115DA. Considere el siguiente fragmento de sub_115DA:

01: .text:000115FF mov	eax, dword_1436C	
02: .text:00011604 mov	editar, ds:wcsncpy	
03: .text:0001160A movimiento	ebx, [eax]	
04: .text:0001160C mov	esi, ebx	
05: .text:0001160E inicio_de_bucle: dword ptr [esi+20h], 0		
06: .text:0001160E cmp 07: .text:00011612 jz		
short fallido		
08: .text:00011614 push 09: .text:00011617	dword ptr [esi+28h]	
llamada	ds:MmIsAddressValid	
10: .text:0001161D prueba	al, al	
11: .text:0001161F jz	corto fallido	
12: .text:00011621 mov	eax, [esi+28h]	
13: .text:00011624 prueba	Ea, Ea	
14: .text:00011626 jz 15: .text:00011628	corto fallido	
movzx ecx, palabra_ptr [esi+24h]		
16: .text:0001162C shr	ecx, 1	
17: .text:0001162E empujar	locals frame	; tamaño_t
18: .text:0001162F empujar 19: .text:00011630	fácil	:wchar_t*
lea	eax, [ebp+var_208]	
20: .text:00011636 push 21: .text:00011637	fácil	:wchar_t*
llamada	edición; wcsncpy	
22: .text:00011639 lea	eax, [ebp+var_208]	
23: .text:0001163F empujar	fácil	:wchar_t*
24: .text:00011640 llamada	ds:_wcslwr	
25: .text:00011646 lea	eax, [ebp+var_208]	

120 Capítulo 3 ■ El núcleo de Windows

26: .texto:0001164C empujar	Desplazamiento aKml	: "kml"
27: .text:00011651 push 28: .text:00011652 llamada	fácil	;wchar_t*
	ds:wcsstr	
29: .text:00011658 agregar	esp, 18h	
30: .text:0001165B prueba	Ea, Ea	
31: .text:0001165D jnz 32: .text:0001165F mov	corto coincidente_kml	
	esi, [esi]	
33: .text:00011661 cmp 34: .text:00011663 jz	esi, ebx	
35: .text:00011665 jmp 36: .text:00011667	Fin del bucle corto	
coincidente_kml:	bucle_corto_inicio	
37: .text:00011667 lea	eax, [ebp+var_208]	
38: .text:0001166D empujar	'\'	;char_t
39: .text:0001166F pulsar 40: .text:00011670 llamar	fácil	;wchar_t*
	ds:wcschr	
41: .text:00011676 pop 42: .text:00011677	eax	
prueba	Ea, Ea	

Las líneas 1 a 4 leen un puntero de una variable global en dword_1436C y lo guardan en EBX y ESI. El cuerpo del bucle hace referencia a este puntero en el desplazamiento 0x20 y 0x28; por lo tanto, puede deducir que es un puntero a una estructura de al menos 0x2c bytes de tamaño. Al final del bucle, lee otro puntero de la estructura y lo compara con el puntero original (guardado en la línea 3). Observe que el puntero se lee desde el desplazamiento 0. Por lo tanto, en este punto, puede suponer que este bucle está iterando sobre una lista en la que el puntero "siguiente" está en el desplazamiento 0. ¿Puede afirmar que esta estructura contiene un campo LIST_ENTRY en el desplazamiento 0? No, no hay suficientes datos concretos en este momento para respaldar eso. Averigüemos de dónde proviene la variable global dword_1436C .

sub_11553 utiliza la convención de llamada STDCALL y toma dos parámetros: un puntero a un DRIVER_OBJECT y un puntero a una variable global dword_1436C.

Tiene el siguiente fragmento de código interesante:

01: .texto:00011578mov	Ej., 0FFDFF034h	
02: .text:0001157D mov	Ea, [ea]	
03: .text:0001157F mov	eax, [eax+70h]	
04: ...		
05: .text:0001159E mov	ecx, [ebp+arg_4]; puntero a la variable global	
06: .text:000115A1 mov	[ecx], eax	

La línea 2 lee un puntero desde una dirección codificada, 0xFFDFF034. En Windows XP, hay una estructura de bloque de control del procesador (que se analiza más adelante en este capítulo) en 0xFFDFF000 y el desplazamiento 0x34 es el puntero KdVersionBlock . Las líneas 3 a 6 leen un valor de puntero en el desplazamiento 0x70 en KdVersionBlock y lo escriben de nuevo en la variable global; sabes que es un puntero porque se utiliza para iterar las entradas de la lista en sub_115DA. Para averiguar el tipo exacto de entrada de la lista, necesitas determinar qué hay en el desplazamiento 0x70 de la estructura KdVersionBlock . Debido a que se trata de una estructura específica del sistema operativo no documentada, tienes que hacer ingeniería inversa del núcleo de Windows XP o buscar en Internet para ver si otras personas ya lo han descubierto . Los resultados indican que en Windows XP, el desplazamiento 0x70 de KdVersionBlock

La estructura es un puntero a una lista global llamada PsLoadedModuleList. Cada entrada de esta lista es del tipo KLDR_DATA_TABLE_ENTRY y almacena información sobre los módulos del núcleo cargados actualmente (nombre, dirección base, tamaño, etc.); el primer miembro de esta estructura es del tipo LIST_ENTRY. Esto tiene sentido porque previamente dedujimos que en el desplazamiento 0 está el puntero “próximo” (Flink para ser precisos).

NOTA La estructura KLDR_DATA_TABLE_ENTRY no está documentada, pero es muy similar a LDR_DATA_TABLE_ENTRY, que se encuentra en los símbolos públicos. En Windows XP, los campos FullDllName y BaseDllName se encuentran en el mismo desplazamiento (0x24 y 0x2c).

Suponiendo que la información de Internet es correcta, estas dos funciones...
Las situaciones pueden resumirse de la siguiente manera:

- sub_11553 lee el puntero KdVersionBlock del bloque de control del procesador y recupera el puntero a PsLoadedModuleList desde allí; guarda este puntero en la variable global. PsLoadedModuleList es el encabezado de una lista cuyas entradas de lista son del tipo KLDR_DATA_TABLE_ENTRY. Esta función recibirá el nombre descriptivo GetLoadedModuleList.
- sub_115DA utiliza el puntero de la lista para iterar sobre todas las entradas buscando un nombre de módulo con la subcadena "krnl". El código busca la subcadena "krnl" porque el autor está buscando el nombre de la imagen del núcleo NT (normalmente "ntoskrnl.exe"). Esta función recibirá el nombre descriptivo nombre GetKernelName.

Puedes traducirlos brevemente de nuevo a C:

```
tipo de definición estructura _KLDR_DATA_TABLE_ENTRY {
    LIST_ENTRY Entrada de lista;
    ...
    UNICODE_STRING NombreDllCompleto;
    UNICODE_STRING NombreBaseDll;
    ...
} ENTRADA DE TABLA DE DATOS KLDR, *ENTRADA DE TABLA DE DATOS PKLDR;

BOOL ObtenerListaDeMódulosCargados(PDRIVER_OBJECT drvobj, PLIST_ENTRY g_modlist)
{
    ...
    g_modlist = (PCR->KdVersionBlock) + 0x70
    ...
}

BOOL ObtenerNombreNúcleo()
{
    WCHAR nombre[...];
    entrada PKLDR_DATA_TABLE_ENTRY;
    PLIST_ENTRADA p = g_modlist->Flink;
    mientras (p != g_modlist)
    {
        ...
    }
}
```

```
entrada = CONTAINING_RECORD(p, ENTRADA_TABLA_DATOS_KLDR, Entrada_Lista);
...
wcsncpy(fname, entrada->FullDIIName.Buffer, entrada->FullDIIName
Longitud * 2);
...
si (wcsstr(fname, L"krnl") != NULL) { ... }
p = p->Flink;
}
...
}
```

Aunque este controlador parezca funcionar en una versión específica de Windows, tiene varios problemas. En primer lugar, supone que el PCR siempre está ubicado en 0xFFDF000 y que KdVersionBlock siempre está en el desplazamiento 0x34; estas suposiciones no se cumplen para Windows Vista+. En segundo lugar, el controlador supone que KdVersionBlock siempre contiene un valor válido; esto no es cierto porque el valor es válido solo para el PCR del primer procesador. Por lo tanto, si este código se ejecutara en un sistema multiprocesador y el subproceso estuviera programado en otro procesador, este código fallaría. En tercer lugar, supone que hay una cadena UNICODE_STRING, en el desplazamiento 0x24 en la estructura KLDR_DATA_TABLE_ENTRY (que no está documentada en sí misma); esto puede no ser siempre cierto porque Microsoft puede agregar o quitar campos de la definición de la estructura, lo que hace que el desplazamiento cambie. En cuarto lugar, este código seguramente fallará en un kernel x64 porque los desplazamientos son todos diferentes. Finalmente, la lista de módulos cargados puede cambiar (es decir, los controladores se descargan) mientras el controlador está iterando la lista; por lo tanto, puede recibir resultados obsoletos o conducir a una violación de acceso como resultado de acceder a un módulo que ya no está allí. También tenga en cuenta que el controlador no utiliza ningún tipo de mecanismo de bloqueo mientras itera una lista global. A medida que analice más rootkits en modo kernel o controladores de terceros, con frecuencia encontrará código escrito con estas suposiciones prematuras.

En este ejemplo en particular, se puede decir que el desarrollador solo quiere obtener el nombre de la imagen del núcleo y la dirección base. Esto se podría haber logrado fácilmente utilizando la API del núcleo documentada AuxKlibQueryModuleInformation. (Consulte también el ejercicio sobre AuxKlibQueryModuleInformation).

Para concluir, nos gustaría discutir brevemente el proceso de pensamiento al analizar estas dos funciones. ¿Cómo pudimos pasar de valores aparentemente aleatorios como 0xFFDF034, 0x70 y 0x28 a PCR, KdVersionBlock, PsLoadedModuleList, KLDR_DATA_TABLE_ENTRY, etc.? La verdad es que ya teníamos conocimiento previo del kernel y experiencia analizando controladores en modo kernel, por lo que instintivamente pensamos en estas estructuras. Por ejemplo, comenzamos con un bucle que procesa cada entrada de la lista buscando la subcadena "krnl"; inmediatamente supusimos que estaban buscando el nombre de la imagen del kernel. Los desplazamientos de cadena y longitud (0x24 y 0x28) nos alertaron de una UNICODE_STRING; con nuestro conocimiento del kernel, supusimos que esta es la estructura KLDR_DATA_TABLE_ENTRY y verificamos que de hecho es el caso usando símbolos públicos. A continuación, sabemos que PsLoadedModuleList es el encabezado de lista global para la lista de mó-

PsLoadedModuleList no es un símbolo exportado, sabemos que el controlador debe recuperarlo de otra estructura. Yendo hacia atrás, vemos la dirección de memoria codificada 0xFFDF034 y pensamos inmediatamente en el PCR. Verificamos esto en el depurador:

```
0; kd>dt ntl_KPCR 0xffffdfff000
+0x000 NtTib : _NT_TIB
+0x01c Autoprueba de personalidad : 0xffffdff000_KPCR
+0x020 Prcb : 0xffffdff120_KPRCB
+0x024 Irql : 0 "
+0x028 TIR : 0
+0x02c IrrActivo : 0
+0x030 IDR : 0xffffffff
+0x034 KdVersionBlock: 0x8054d2b8 Nulo
...
...
```

Por experiencia, sabemos que KdVersionBlock es un puntero a una estructura grande que almacena información interesante, como la dirección base del kernel y los encabezados de listas.

En ese punto, tenemos toda la información y las estructuras de datos para entender el código.

Como puede ver, hay un proceso de pensamiento sistemático detrás del análisis; sin embargo, requiere una cantidad sustancial de conocimientos básicos sobre el sistema operativo y experiencia. Cuando recién comienza, es posible que no tenga todo el conocimiento y la intuición necesarios para comprender rápidamente los controladores en modo kernel. ¡No tema! Este libro intenta proporcionar una base sólida al explicar todos los conceptos y estructuras de datos principales del kernel. Con una base sólida y mucha práctica (vea los ejercicios), eventualmente podrá hacerlo con gran facilidad. Recuerde: conocimiento básico + intuición + experiencia + paciencia = habilidades.

Ceremonias

1. En Windows 8 x64, las siguientes funciones del kernel tienen InitializeListHead

Incluido al menos una vez:

- CcAllocateInitializeMbc
- Devoluciones de llamadas de CrmInit
- Devolución de llamada ExCreate
- ExplInitSystemPhase0
- ExplInitSystemPhase1
- ExpTimerInicialización
- Procesador de arranque inicial
- IoCreateDevice

- IoInitializeIrp
- KeInitHilo
- KeInitializeMutex
- KeInitializeProcess
- KeInitializeTimerEx
- KeInitializeTimerTable
- Procesador KilInitialize
- KilInitializeThread
- MiInitializeLoadedModuleList
- MiniInicializarPrefetchHead
- Proceso de asignación de Psp
- PspAllocateThread

Identifique dónde se incluye InitializeListHead en estas rutinas.

2. Repita el ejercicio anterior para InsertHeadList en las siguientes rutinas:

- CcSetVacblnFreeList
- Ordenar por cmp
- ExBurnMemory
- ExPoolLibreConEtiqueta
- Lectura de páginas de Io
- Controlador de llamadas Iovp1
- KeInitHilo
- KiInsertQueueApc
- KiInsertQueueDpc
- Hilo KiQueueReady
- MiInsertarEnEspacioDelSistema
- MiUpdateWsle
- ObpInsertCallbackByAltitude

3. Repita el ejercicio anterior para InsertTailList en las siguientes rutinas:

- AlpcpCreateClientPort
- AlpcpCreateSección
- AlpcpCreateView
- AuthzBasepAddSecurityAttributeToLists
- CcFlushCachePriv

- Administrador de caché CcInitialize
- CcInsertVacbArray
- CcSetFileSizesEx
- Tecla CmRename
- ExAllocatePoolWithTag
- ExPoolLibreConEtiqueta
- Elemento de trabajo ExQueue
- Devolución de llamada de ExRegister
- ExpSetTimer
- IoSetIoCompletionEx2
- KeInsertQueueDpc
- Hilo de inicio de Ke
- KiAddThreadToScbQueue
- KiInsertQueueApc
- Hilo KiQueueReady
- MiInsertarNuevoProceso
- PnpRequestDeviceAction
- Proceso de inserción de Psp
- PsplInsertar hilo

4. Repita el ejercicio anterior para RemoveHeadList en las siguientes rutinas:

- Puerto de recursos de Flush de Alpc
- CcEliminarMbcb
- CcGetVacbMiss
- Trabajador de confirmación perezosa de Cmp
- ExAllocatePoolWithTag
- FsRtlNotifyCompleteIrpList
- IoInitializeBootDrivers
- Lista de desconexión de KiProcess
- Solicitud de obtención de finalización de PnpDeviceCompletionQueue
- Tabla de átomos de destrucción Rt!
- Tabla de átomos vacía de Rt!
- RtpLiberarTodosLosÁtomos

5. Repita el ejercicio anterior para RemoveTailList en las siguientes rutinas:

- Proceso de datos persistente de la aplicación de arranque
- Devoluciones de llamadas de CmpCall
- Trabajador de cierre de retardo de cmp
- Devoluciones de llamadas posteriores a la operación de ObpCall
- Entrada de caché de RaspAdd

6. Repita el ejercicio anterior para RemoveEntryList en las siguientes rutinas:

- Procedimiento de eliminación de AlpcSection
- AlpcpDeletePort
- AlpcpUnregisterCompletionListDatabase
- AuthzBasepEliminar atributo de seguridad de las listas
- CcEliminarBcbs
- Entrada de cola de trabajo siguiente de CcFind
- CcLazyWriteScan
- CcSetFileSizesEx
- Sistema de apagado automático
- Devolución de llamada de CmUnRegister
- Devoluciones de llamadas de CmpCall
- CmpPostApc
- ExPoolLibreConEtiqueta
- Elemento de trabajo ExQueue
- Resumen del temporizador ExTimer
- ExpDeleteTimer
- ExpSetTimer
- Eliminar dispositivo Io
- IoUnregisterFsRegistrationChange
- Solicitud completa de lopf
- Devolución de llamada de comprobación de errores de KeDeregister
- Notificación de desregistro de objeto KeDeregister
- Notificación de objeto de registro de Ke
- KeRemoveQueueApc

- KeRemoveQueueDpc
 - Temporizador de cancelación de Ki
- KeTerminateThread
- KiDeliverApc
- KiExecuteAllDpcs
- Tabla de temporizadores KiExpire
- Hilo KiFindReady
- KiFlushQueueApc
 - Tabla de temporizadores de inserción Ki
- Lista de temporizadores expirados de KiProcess
- Eliminar direcciones virtuales
- Notificar cambios múltiples de teclas
- Devoluciones de llamadas de ObRegister
- Devoluciones de llamadas de ObUnRegister

7. Repita los ejercicios anteriores en Windows 8 x86/ARM y Windows 7 x86/x64.
¿Cuáles fueron las diferencias (si las hubo)?
8. Si realizó los ejercicios para InsertHeadList, InsertTailList, RemoveHeadList, RemoveTailList y RemoveEntryList en Windows 8, debería haber observado una construcción de código común a todas estas funciones. Esta construcción también debería permitirle detectar fácilmente las rutinas de inserción y eliminación de listas en línea. Explique esta construcción de código y por qué está allí. Sugerencia: Esta construcción existe solo en Windows 8 y requiere que observe el IDT.
9. En el tutorial, mencionamos que un controlador puede enumerar todos los módulos cargados con la API documentada AuxKlibQueryModuleInformation. ¿Esta API garantiza que la lista de módulos devuelta esté siempre actualizada? Explique su respuesta. A continuación, aplique ingeniería inversa a AuxKlibQueryModuleInformation en Windows 8 y explique cómo funciona. ¿Cómo maneja el caso en el que varios subprocesos solicitan acceso a la lista de módulos cargados? Nota: La función interna que maneja esta solicitud (y otras) es bastante grande, por lo que necesitará algo de paciencia. Alternativamente, puede usar un depurador para ayudarlo a rastrear el código interesante.
10. Explique cómo funcionan las siguientes funciones: KeInsertQueueDpc, KiRetireDpcList, KiExecuteDpc y KiExecuteAllDpcs. Si cree que es un gran alumno, descompila esas funciones de los ensamblados x86 y x64 y explica las diferencias.

Ejecución asincrónica y ad hoc

Durante la vida útil de un controlador, este puede crear subprocessos del sistema, registrar devoluciones de llamadas para determinados eventos, poner en cola una función para que se ejecute en el futuro, etc. Esta sección cubre una variedad de mecanismos que un controlador puede utilizar para lograr estas formas de ejecución asincrónica y ad hoc. Los mecanismos cubiertos incluyen subprocessos del sistema, elementos de trabajo, APC, DPC, temporizadores y devoluciones de llamadas de procesos y subprocessos.

Hilos del sistema

Un programa típico en modo usuario puede tener varios subprocessos que gestionen distintas solicitudes. De manera similar, un controlador puede crear varios subprocessos para gestionar solicitudes del núcleo o del usuario. Estos subprocessos se pueden crear con PsCreateSystemThread API:

```
NTSTATUS PsCreateSystemThread(  
    _Afuera_      Manejador de hilo PHANDLE,  
    _En_          ULONG Acceso deseado,  
    _En_opt_     POBJECT_ATTRIBUTES Atributos del objeto,  
    _En_opt_     MANEJAR ProcesoManejador,  
    _Out_opt_    PCLIENT_ID Id. de cliente,  
    _En_          PKSTART_ROUTINE Iniciar rutina,  
    Contexto de inicio de PVOID _In_opt_  
);
```

Si se llama con un parámetro ProcessHandle NULL , esta API creará un nuevo subprocesso en el proceso del sistema y establecerá su rutina de inicio en StartRoutine. El uso de subprocessos del sistema varía según el requisito del controlador. Por ejemplo, el controlador puede decidir crear un subprocesso durante la inicialización para manejar solicitudes de E/S posteriores o esperar algunos eventos. Un ejemplo concreto es el núcleo que crea un subprocesso del sistema para procesar DPC (consulte también la función KiStartDpcThread).

Ceremonias

1. Después de leer algunos foros en línea, observa que algunas personas sugieren que PsCreateSystemThread creará un hilo en el contexto del proceso de llamada. En otras palabras, sugieren que si llama a PsCreateSystemThread en un controlador IOCTL, el nuevo hilo estará en el contexto de la aplicación en modo usuario solicitante. Evalúe la validez de esta declaración escribiendo un controlador que llame a PsCreateSystemThread en el controlador IOCTL. A continuación, experimente con un ProcessHandle que no sea NULL .
y determinar si el contexto difiere.
2. Realice referencias cruzadas de tantas llamadas a PsCreateSystemThread como sea posible en la imagen del núcleo. Determine si alguna de ellas pasa un valor distinto de NULL .

Parámetro ProcessHandle . Explique el propósito de estas rutinas. Repita el ejercicio para tantas funciones como sea posible.

Elementos de trabajo

Los elementos de trabajo son similares a los subprocessos del sistema, excepto que no se crean objetos de subprocesso físicos para ellos. Un elemento de trabajo es simplemente un objeto en una cola procesado por un grupo de subprocessos del sistema. En términos concretos, un elemento de trabajo es una estructura definida de la siguiente manera:

```
0: kd> dt ntl!_IO_WORKITEM
+0x000 Elemento de trabajo :ELEMENTO_DE_COLA_DE_TRABAJO
+0x020 Rutina :Ptr64 vacío
+0x028 Objeto Io :Ptr64 Vacío
+0x030 Context :Ptr64 Vacío
+0x038 Tipo :UInt4B
+0x03c Id. de actividad :_GUID
0: kd> dt ntl!ELEMENTO_DE_COLA_DE_TRABAJO
+0x000 Lista :_ENTRADA_DE_LISTA
+0x010 Rutina de trabajo :Ptr64 vacío
Parámetro +0x018 :Ptr64 Vacío
```

Tenga en cuenta que su campo WorkItem es en realidad una entrada de lista que contiene la rutina de trabajo y el parámetro. Esta entrada se insertará en una cola más adelante. Un controlador llama a la función IoAllocateWorkItem para obtener un puntero a un IO_WORKITEM asignado en un grupo no paginado. A continuación, el controlador inicializa y pone en cola el elemento de trabajo llamando a IoQueueWorkItem:

```
PIO_WORKITEM IoAllocateWorkItem(
    _En_ PDEVICE_OBJECT Objeto de dispositivo
);

ANULAR IoQueueWorkItem(
    _En_ PIO_WORKITEM Elemento de trabajo de lo,
    _En_ PIO_WORKITEM_ROUTINE Rutina de trabajo,
    _En_ WORK_QUEUE_TYPE Tipo de cola,
    Contexto PVOID _In_opt_
);
```

La parte de inicialización simplemente completa la rutina del trabajador, el parámetro/contexto, y prioridad/tipo de cola:

```
IO_WORKITEM_ROUTINE Elemento de trabajo;
Elemento de trabajo nulo{
    _En_ PDEVICE_OBJECT Objeto del dispositivo,
    Contexto PVOID _In_opt_
}
{ ... }
```

```
tipo de enumeración de definición de tipo _WORK_QUEUE_TYPE {
    Cola de trabajo crítica = 0,
    Cola de trabajo retrasada = 1,
    Cola de trabajo hiper crítica = 2,
    Cola de trabajo máxima = 3
} TIPO_DE_COLA_DE_TRABAJO;
```

¿Dónde se pone en cola? Como se explicó anteriormente, cada procesador tiene un KPRCB asociado que contiene un campo llamado ParentNode, que es un puntero a un KNODE Estructura; cuando se inicializa el procesador, este puntero apunta a un ENODE Estructura que contiene la cola de elementos de trabajo:

```
Cola de elementos de trabajo

0: kd>dt nt!_KPRCB
...
+0x5338 Nodo principal : Ptr64 _NODO
0: kd> dt nt!_KNODE
+0x000 Conjunto inactivo profundo :Uint8B
+0x040 Id. de proximidad :Uint4B
+0x044 Número de nodo :Uint2B
0: kd>dt nt!_ENODE
+0x000 Ncb : _NODO
+0x0c0 ExWorkerQueues: [7] _EX_WORK_QUEUE
+0x2f0 ExpThreadSetManagerEvento: _KEVENT
+0x308 ExpWorkerThreadBalanceManagerPtr: Ptr64_ETHREAD
+0x310 Semilla de trabajador de Exp :Uint4B
+0x314 ExWorkerFullInit: posición 0, 1 bit
+0x314 ExWorkerStructInit: posición 1, 1 bit
+0x314 ExWorkerFlags :Uint4B
0: kd> dt nt!_EX_WORK_QUEUE
+0x000 Cola de trabajadores : _KQUEUE
+0x040 Elementos de trabajo procesados: Uint4B
+0x044 Elementos de trabajo procesados Último paso: Uint4B
+0x048 Número de subprocessos :Int4B
+0x04c Intento fallido :UCar
```

```
Elemento de trabajo de cola de espera

ExQueueWorkItemEx procedimiento cerca
...
movimiento rax, gs:20h
movimiento r8, [rax+5338h] ; enodo
movzx eax, palabra ptr [r8+44h]
movimento ecx, eax
pasto Rax, [rax+rax*2]
shl Rax, 6
agregar rax, rbp
...
movimiento edx, r9d ; tipo de cola
movimiento rcx, r11; elemento de trabajo pasado
llamar Nodo de elemento de trabajo de cola de espera
```

Lo que ocurre en realidad es que cada procesador tiene varias colas para almacenar los elementos de trabajo y hay un subproceso del sistema que va sacando de la cola un elemento a la vez para su ejecución. Este subproceso del sistema encargado de sacar de la cola es ExpWorkerThread.

Como se explicó anteriormente, los elementos de trabajo son livianos porque no requieren la creación de nuevos objetos de subproceso. También tienen dos propiedades importantes:

- Se ejecutan en el contexto del proceso del Sistema. La razón es porque ExpWorkerThread se ejecuta en el proceso del sistema.
- Se ejecutan en NIVEL_PASIVO.

Debido a su naturaleza liviana, es un patrón de programación de controladores común. para poner en cola elementos de trabajo dentro de un DPC.

Ceremonias

1. Explique cómo pudimos determinar que ExpWorkerThread es el hilo del sistema responsable de sacar de la cola los elementos de trabajo y ejecutarlos.
Sugerencia: La forma más rápida es escribir un controlador.
2. Explore IoAllocateWorkItem, IoInitializeWorkItem, IoQueueWorkItem, IoQueueWorkItemProlog y ExQueueWorkItem, y explique cómo funcionan.
3. Los elementos de trabajo y los subprocesos del sistema (es decir, aquellos creados por PsCreateSystemThread) son en su mayoría idénticos en términos de funcionalidad, así que explique por qué los DPC con frecuencia ponen en cola elementos de trabajo para manejar solicitudes pero nunca llaman a PsCreateSystemThread.
4. Escriba un controlador para enumerar todos los elementos de trabajo en el sistema y explicar los problemas que tuvo que superar en el proceso.

Llamadas a procedimientos asincrónicos

Las llamadas a procedimientos asincrónicos (APC) se utilizan para implementar muchas operaciones importantes, como la finalización asincrónica de E/S, la suspensión de subprocesos y el cierre de procesos. Lamentablemente, no están documentadas desde una perspectiva del núcleo. La documentación oficial de desarrollo de controladores incluye simplemente una breve sección que reconoce que existen los APC y que hay distintos tipos. Sin embargo, para las tareas de ingeniería inversa habituales, no es necesario comprender todos los detalles subyacentes. Esta sección explica qué son los APC y cómo se utilizan habitualmente.

Fundamentos de APC

En términos generales, las APC son funciones que se ejecutan en un contexto de hilo particular. Se pueden dividir en dos tipos: modo kernel y modo usuario. Modo kernel

132 Capítulo 3 ■ El núcleo de Windows

Las APC pueden ser normales o especiales; las normales se ejecutan en PASSIVE_LEVEL, mientras que las especiales se ejecutan en APC_LEVEL (ambas se ejecutan en modo kernel). Las APC de usuario se ejecutan en PASSIVE_LEVEL en modo usuario cuando el subproceso está en un estado de alerta . Debido a que las APC se ejecutan en el contexto del subproceso, siempre están asociadas con un objeto ETHREAD .

Concretamente hablando, un APC se define por la estructura KAPC :

```
1: kd> dt nt!_KAPC
+0x000 Tipo :UCar
+0x001 Byte de repuesto 0 :UCar
+0x002 Tamaño :UCar
+0x003 Byte de repuesto 1 :UCar
+0x004 RepuestoLong0 :UInt4B
+0x008 Hilo : Ptr32 _KHILO
+0x00c Entrada de lista de aplicaciones :_ENTRADA_DE_LISTA
+0x014 Rutina del núcleo :Ptr32 vacío
+0x018 Rutina de ejecución: Ptr32 vacío
+0x01c Rutina normal :Ptr32 vacío
+0x014 Reservado : [3] Ptr32 Vacío
+0x020 Contexto normal : Ptr32 Vacío
+0x024 Argumento del sistema1: Ptr32 nulo
+0x028 Argumento del sistema2: Ptr32 nulo
+0x02c Índice de estado de Apc :Carácter
+0x02d Modo Apc :Carácter
+0x02e Insertado :UCar
```

Esta estructura se inicializa mediante la API KeInitializeApc :

InicializarApc

```
NTKERNELAPI VOID KeInitializeApc(
    Agente de prioridad personal PRKAPC,
    Hilo PKTHREAD,
    KAPC_ENVIRONMENT Medio ambiente,
    PKKERNEL_ROUTINE Rutina del núcleo,
    PKRUNDOWN_ROUTINE Rutina de ejecución,
    PKNORMAL_ROUTINE Rutina Normal,
    KPROCESSOR_MODE Modo de procesador,
    PVOID Contexto normal
);
```

```
NTKERNELAPI BOOLEAN KeInsertQueueApc(
    PRKAPC Apc,
    PVOID Argumento del sistema1,
    PVOID Argumento del sistema2,
    Incremento de KPRIORITY
);
```

```
Prototipos de devolución de llamada

tipo definido VOID (*PKKERNEL_ROUTINE)(

    Apuntes de protección personal PkApc
    PKNORMAL_ROUTINE *Rutina Normal,
    PVOID *ContextoNormal,
    PVOID *Argumento del sistema1,
    PVOID *Argumento del sistema2
);

tipo definido VOID (*PKRUNDOWN_ROUTINE)(

    Apuntes de protección personal PkApc
);

tipo definido VOID (*PKNORMAL_ROUTINE)(

    PVOID Contexto normal,
    PVOID Argumento del sistema1,
    Argumento del sistema PVOID2
);

tipo de enumeración _KAPC_ENVIRONMENT {
    OriginalApcEnvironment,
    AdjuntoApcEnvironment,
    Entorno actual de Apc,
    InsertarApcEnvironment
} KAPC_ENVIRONMENT, *PKAPC_ENVIRONMENT;
```

NOTA Esta definición está tomada de http://forum.sysinternals.com/howto-capture-kernel-stack-traces_topic19356.html.

Si bien no podemos garantizar su exactitud, se sabe que funciona en experimentos.

Apc es un buffer asignado por el llamador de tipo KAPC. En la práctica, ExAllocatePool suele asignarlo en un grupo no paginado y liberarlo en el núcleo o en una rutina normal. El subproceso es el subproceso en el que se debe poner en cola este APC. El entorno determina el entorno en el que se ejecuta el APC; por ejemplo, OriginalApcEnvironment significa que la APC se ejecutará en el contexto del proceso del subproceso (si no se adjunta a otro proceso). KernelRoutine es una función que se ejecutará en APC_LEVEL en modo kernel; RundownRoutine es una función que se ejecutará cuando el subproceso esté terminando; y NormalRoutine es una función que se ejecutará en PASSIVE_LEVEL en ProcessorMode. Las APC en modo usuario son aquellas que tienen una NormalRoutine y un ProcessorMode establecidos en UserMode. NormalContext es el parámetro que se pasa a la NormalRoutine.

Una vez inicializado, un APC se pone en cola con la API KeInsertQueueApc . Apc es el APC inicializado por KeInitializeApc. SystemArgument1 y SystemArgument2 Son argumentos opcionales que se pueden pasar al kernel y a las rutinas normales. Incremento es el número para incrementar la prioridad de tiempo de ejecución; es similar al parámetro PriorityBoost en IoCompleteRequest. ¿Dónde se pone en cola el APC?

Recuerde que los APC siempre están asociados a un subproceso. La estructura KTHREAD tiene dos colas de APC:

```
0: kd> dt nt!_KTHREAD
  Encabezado +0x000 : _ENCABEZADO_DE_DESPACHO
  +0x018 SListFaultAddress: Ptr64 nulo
  +0x020 Objetivo cuántico : UInt8B
  ...
  +0x090 Marco de trampa : Ptr64 _KTRAP_MARCO
  +0x098 Estado de Apc : _ESTADO_KAPC
  +0x098 Estado de relleno de Apc : [43] UCar
  +0x0c3 Prioridad : Caracter
  +0x288 ProgramadorApc : _KAPC
  ...
  +0x2e0 SuspenderEvento : _EVENTO
0: kd> dt nt!_KAPC_STATE
  +0x000 Encabezado de lista de aplicaciones : [2] _ENTRADA_DE_LISTA
  +0x020 Proceso : Ptr64 _KPROCESO
  +0x028 KernelApcInProgress: UChar
  +0x029 KernelApcPendiente: UChar
  +0x02a UsuarioApcPending: UChar
```

El campo ApcState contiene una matriz de dos colas, que almacenan APC en modo kernel y en modo usuario, respectivamente.

Implementación de la suspensión de subprocessos con APC

Cuando un programa desea suspender un subprocesso, el núcleo pone en cola un APC del núcleo para el subprocesso. Este APC de suspensión es el campo SchedulerApc en el KTHREAD Estructura; se inicializa en KeInitThread con KiSchedulerApc como rutina normal . KiSchedulerApc simplemente mantiene el SuspendEvent del hilo. Cuando el programa desea reanudar el hilo, KeResumeThread libera este evento.

A menos que estés realizando ingeniería inversa en el kernel de Windows o en rootkits en modo kernel, es poco probable que te encuentres con código que utilice APC. Esto se debe principalmente a que no están documentados y, por lo tanto, no se utilizan comúnmente en controladores comerciales. Sin embargo, los APC se utilizan con frecuencia en rootkits porque ofrecen una forma limpia de inyectar código en modo usuario desde el modo kernel. Los rootkits logran esto al poner en cola un APC en modo usuario en un subprocesso en el proceso en el que desean inyectar código.

Ceremonias

1. Escriba un controlador utilizando APC en modo kernel y en modo usuario.
2. Escriba un controlador que enumere todos los APC en modo usuario y modo kernel para todos los subprocessos de un proceso. Sugerencia: debe tener en cuenta el nivel de IRQL al realizar la enumeración.

3. La función del núcleo KeSuspendThread es responsable de suspender un subproceso.
 Anteriormente aprendiste que los APC están involucrados en la suspensión de subprocesos en Windows 8. Explica cómo funciona esta función y cómo se utilizan los APC para implementar la funcionalidad en Windows 7. ¿En qué se diferencia de Windows 8?
4. Los APC también se utilizan en el cierre de procesos. El objeto KTHREAD tiene un indicador llamado ApcQueueable que determina si se puede poner en cola un APC.
 ¿Qué sucede cuando deshabilitas la cola de APC para un subproceso? Experimenta con esto iniciando notepad.exe y luego deshabilitando manualmente la cola de APC para uno de sus subprocesos (usa el depurador del núcleo para hacer esto).
5. Explique qué hacen las siguientes funciones:

- KilInsertQueueApc
- PsSalirApcEspecial
- Resumen de PspExitApc
- PspSalirNormalApc
- PspQueueApcApc especial
- KiDeliverApc

6. Explique cómo funciona la función KeEnumerateQueueApc y luego recupere su prototipo. Nota:
 Esta función solo está disponible en Windows 8.
7. Explique cómo el núcleo distribuye los APC. Escriba un controlador que utilice los diferentes tipos de APC y visualice la pila cuando se ejecutan. Nota: Utilizamos el mismo método para averiguar cómo funcionan los envíos del núcleo.
 elementos.

Llamadas a procedimientos diferidos

Las llamadas a procedimientos diferidos (DPC) son rutinas que se ejecutan en DISPATCH_LEVEL en un contexto de subproceso arbitrario en un procesador en particular. Los controladores de hardware las utilizan para procesar las interrupciones que provienen del dispositivo. Un patrón de uso típico es que la rutina de servicio de interrupciones (ISR) ponga en cola una DPC, que a su vez pone en cola un elemento de trabajo para realizar el procesamiento.

Los controladores de hardware hacen esto porque el ISR generalmente se ejecuta en IRQL altos (por encima de DISPATCH_LEVEL) y si tarda demasiado, podría reducir el rendimiento general del sistema. Por lo tanto, el ISR generalmente pone en cola un DPC y regresa inmediatamente para que el sistema pueda procesar otras interrupciones. Los controladores de software pueden usar DPC para ejecutar rápidamente tareas cortas.

Internamente, un DPC se define mediante la estructura KDPC :

```
0: kd>dt ntl!_KDPC
+0x000 Tipo :UCar
+0x001 Importancia :UCar
```

+0x002 Número	:Uint2B
+0x008 Entrada de lista Dpc	:_ENTRADA_DE_LISTA
+0x018 Rutina diferida: Ptr64	vacio
+0x020 DeferredContext : Ptr64 Vacio	
+0x028 Argumento del sistema1: Ptr64 nulo	
+0x030 Argumento del sistema2: Ptr64 nulo	
+0x038 Datos de Dpc	:Ptr64 Vacio

La semántica de cada campo es la siguiente:

- **Tipo:** tipo de objeto. Indica el tipo de objeto del núcleo para este objeto (es decir, proceso, subproceso, temporizador, DPC, eventos, etc.). Recuerde que los objetos del núcleo se definen mediante la enumeración nt!_KOBJECTS . En este caso, se trata de DPC, para los cuales hay dos tipos: normal y con subprocesos.
- **Importancia:** importancia de DPC. Determina dónde debe estar esta entrada de DPC en la cola de DPC. Consulte también KeSetImportanceDpc.
- **Número:** número de procesador en el que se debe poner en cola y ejecutar el DPC. Véase también KeSetTargetProcessorDpc.
- **DpcListEntry—LIST_ENTRY para la entrada DPC.** Internamente, la inserción /eliminación de DPC de la cola DPC opera en este campo. Consulte Cola de inserción KeInsertQueueDpc.
- **DeferredRoutine:** la función asociada con este DPC. Se ejecutará en un contexto de subproceso arbitrario y en DISPATCH_LEVEL. Se define de la siguiente manera:

```
KDEFERRED_ROUTINEDpc personalizado;
ANULAR CustomDpc(
    _En_           estructura_KDPC *Dpc,
    _En_opt_ PVOID DeferredContext,
    _En_opt_ PVOID Argumento del sistema1,
    _In_opt_ PVOID Argumento del sistema2
)
{ ... }
```

- **DeferredContext:** parámetro que se pasará a la función DPC.
- **SystemArgument1:** datos personalizados para almacenar en el DPC.
- **SystemArgument2:** datos personalizados para almacenar en el DPC.
- **DpcData:** un puntero a una estructura KDPC_DATA :

```
0: kd> dt nt!_KDPC_DATA
+0x000 DpcListHead +0x010      :_ENTRADA_DE_LISTA
DpcLock          :Uint8B
```

+0x018 Profundidad de cola de Dpc	:Int4B
+0x01c Conteo de Dpc	:Uint4B

Como puede ver, mantiene información de contabilidad sobre los DPC. Los datos se almacenan en el campo DpcData de la estructura KPRCB asociada con el DPC. DpcListHead es la entrada principal en la cola del DPC (se establece durante la inicialización de KPRCB) y DpcLock es el spinlock que protege esta estructura ; cada vez que se pone en cola un DPC, DpcCount y DpcQueueDepth se incrementan en uno. Consulte también KelInsertQueueDpc. Puede resultar instructivo analizar KelInsertQueueDpc en ensamblaje; preste atención al acceso a KPRCB y a la inserción de la lista principal/final.

El patrón de uso de DPC en el código es simple: inicialice el objeto KDPC con KelInitializeDpc y colóquelo en cola con KelInsertQueueDpc. Cuando el IRQL del procesador desciende a DISPATCH_LEVEL, el núcleo procesa todos los DPC en esa cola.

Como se mencionó anteriormente, cada núcleo de CPU mantiene su propia cola de DPC. Esta cola es monitoreada por la estructura KPRCB por núcleo:

0: kd>dt nt!_KPRCB	
+0x000 MxCsr	:Uint4B
+0x004 Número heredado	:UCar
+0x005 ReservadoDebe ser cero: UChar	
+0x006 Solicitud de interrupción: UChar	
...	
+0x2d80 DpcData	: [2] _DATOS_KDPC
+0x2dc0 DpcStack	:Ptr64 Vacío
+0x2dc8 Profundidad máxima de cola de Dpc: Int4B	
+0x2dcf Tasa de solicitud de Dpc: Uint4B	
+0x2dd0 TasaDpc mínima: Uint4B	
+0x2dd4 Último recuento de dpc	:Uint4B
+0x2dd8 ThreadDpcEnable: UChar	
+0x2dd9 Fin cuántico	:UCar
+0x2dda DpcRoutineActive: UChar	
0: kd> dt nt!_KDPC_DATA	
+0x000 DpcListHead +0x010	:_ENTRADA_DE_LISTA
DpcLock +0x018	:Uint8B
DpcQueueDepth +0x01c DpcCount	:Int4B
	:Uint4B

Los dos campos más importantes son DpcData y DpcStack. DpcData es una matriz de estructuras KDPC_DATA en la que cada elemento realiza un seguimiento de una cola de DPC; el primer elemento realiza un seguimiento de las DPC normales y el segundo de las DPC enhebradas. La función KelInsertQueueDpc simplemente inserta la DPC en una de estas dos colas. La relación se puede ilustrar como se muestra en la Figura 3-7.

138 Capítulo 3 ■ El núcleo de Windows



DpcStack es un puntero a un bloque de memoria que se utilizará como pila de la rutina DPC.

Windows tiene varios mecanismos para procesar la cola de DPC. El primer mecanismo es a través de KidleLoop. Mientras está “en reposo”, verifica el PRCB para determinar si hay DPC en espera y, en caso afirmativo, llama a KiRetireDpcList para procesar todos los DPC. Por eso, a veces, estas dos funciones aparecen en la pila mientras se ejecuta un DPC. Por ejemplo:

```
0: kd>kn
# Niño-SP          RetDirección      Sitio de llamada
00 fffff800`00b9cc88 fffff800`028db5dc PUERTO USB! PUERTO USB_IsrDpc
01 fffff800`00b9cc90 fffff800`028d86fa ;No! KiRetireDpcList+0x1bc
02 fffff800`00b9cd40 00000000 00000000 nt!KidleLoop+0x5a
```

El segundo mecanismo ocurre cuando la CPU está en DISPATCH_LEVEL. Considere La siguiente pila:

```
0: kd>kn
# Niño-SP          RetDirección      Sitio de llamada
00 fffff800`00ba2ef8 fffff800`028db5dc PUERTO USB! PUERTO USB_IsrDpc
01 fffff800`00ba2f00 fffff800`028d6065 ;No! KiRetireDpcList+0x1bc
02 fffff800`00ba2fb0 fffff800`028d5e7c ;No! Lista de retiros de KyDpc+0x5
03 fffff880`04ac67a0 fffff800`0291b793 nt!KiDispatchInterruptContinuar
04 fffff880`04ac67d0 fffff800`028cbda2 ;No! KiDpcInterruptBypass+0x13
05 fffff880`04ac67e0 fffff960`0002992c nt!KiInterruptDispatch+0x212
06 fffff880`04ac6978 fffff960`000363b3 win32k!AlphaPerPixelOnly+0x7c
07 fffff880`04ac6980 fffff960`00035fa4 win32k!AlphaScanLineBlend+0x303
08 fffff880`04ac6a40 fffff960`001fd4f9 win32k!EngAlphaBlend+0x4f4
09 fffff880`04ac6cf0 fffff960`001fdbaa win32k!NtGdiUpdateTransform+0x112d
0a fffff880`04ac6db0 fffff960`001fdd19 win32k!NtGdiUpdateTransform+0x17de
0b fffff880`04ac6ed0 fffff960`001fdded8 win32k!EngNineGrid+0xb1
0c fffff880`04ac6f70 fffff960`001fe395 win32k!EngDrawStream+0x1a0
```

```
0d fffff880`04ac7020 fffff960`001fce7 win32k!NgdiDrawStreamInternal+0x47d
0e fffff880`04ac70d0 fffff960`0021a480 win32k!GreDrawStream+0x917
0f fffff880`04ac72c0 fffff800`028cf153 win32k!NgdiDrawStream+0x9c
10 fffff880`04ac7420 0000007fe`fd762cda ;No! KiSystemServiceCopyEnd+0x13
```

Esta pila larga indica que win32k.sys estaba manejando alguna solicitud de operación gráfica del usuario y luego se ejecutó la rutina DPC del controlador del puerto USB (que no tiene nada que ver con win32k). Lo que probablemente sucedió es que mientras win32k.sys estaba manejando la solicitud, se produjo una interrupción del dispositivo que hizo que la CPU funcionara en el IRQL del dispositivo; y luego el IRQL finalmente se redujo a DISPATCH_LEVEL, lo que hace que se procese la cola DPC.

El tercer mecanismo es a través de un subprocesso del sistema creado durante la inicialización del procesador. KiStartDpcThread crea un subprocesso (KiExecuteDpc) para cada procesador, que procesa la cola DPC cada vez que se ejecuta. Por ejemplo:

```
0: kd>kn
# Niño-SP           RetDirección          Sitio de llamada
00 fffff880`03116be8 fffff800`028aadb0 ;No lo sé! Perro guardián KiDpc
01 fffff880`03116bf0 fffff800`028aac4b nt! KiExecuteAllDpcs+0x148
02 fffff880`03116cb0 fffff800`02b73166 ;No! KiExecuteDpc+0xcb
03 fffff880`03116d00 fffff800`028ae486 nt! PspSistemaHiloInicio+0x5a
04 fffff880`03116d40 00000000`00000000 nt! KiStartSystemThread+0x13
```

Recordemos que el despachador de subprocessos se ejecuta en DISPATCH_LEVEL, y el código que se ejecuta en este IRQL no puede ser interrumpido por otros IRQL de software (es decir, aquellos por debajo de DISPATCH_LEVEL). En otras palabras, si hay un bucle infinito en la rutina DPC, el procesador asociado a él girará eternamente y el sistema prácticamente se “congelará”; en un sistema multiprocesador, puede que no se congele, pero el procesador que ejecuta el DPC no será utilizable por el despachador de subprocessos. Además, la rutina DPC no puede esperar ningún tipo de objeto de despachador porque el propio despachador opera en DISPATCH_LEVEL; por eso existen funciones como KeWaitForSingleObject. y KeDelayExecutionThread no se puede llamar en rutinas DPC.

NOTA: Windows tiene una rutina de vigilancia de DPC que detecta los DPC que se ejecutan durante un período de tiempo determinado y comprueba errores con el código DPC_WATCHDOG_VIOLACIÓN (0x133). Puede consultar el valor del temporizador de vigilancia llamando Información del perro guardián de KeQueryDpc.

Algunos rootkits utilizan DPC para sincronizar el acceso a listas enlazadas globales. Por ejemplo, pueden eliminar una entrada de la lista ActiveProcessLinks para ocultar procesos; dado que esta lista puede ser modificada en cualquier momento por cualquier procesador, algunos autores de rootkits utilizan un DPC junto con otro mecanismo de sincronización para operar en ella de forma segura. En uno de los ejercicios, se le pedirá que explique por qué algunos autores tienen éxito en esto mientras que otros fallan (verificaciones de errores de la máquina).

Ceremonias

1. ¿Dónde y cuándo se inicializa el campo DpcData en KPRCB?
2. Escriba un controlador para enumerar todos los DPC en todo el sistema. ¡Asegúrese de que sea compatible con sistemas multiprocesador! Explique las dificultades y cómo las resolvió.
3. Explique cómo funciona la rutina KiDpcWatchdog .

Temporizadores

Los temporizadores se utilizan para señalar el vencimiento de un período de tiempo determinado, que puede ser periódico o en algún momento en el futuro. Opcionalmente, el temporizador también puede estar asociado a un DPC. Por ejemplo, si un controlador desea verificar el estado de un dispositivo cada cinco minutos o ejecutar una rutina dentro de 10 minutos, puede lograrlo mediante el uso de temporizadores.

Concretamente hablando, un temporizador se define mediante la estructura KTIMER :

```
Estructuras relacionadas con el temporizador

0: kd> dt ntl!_KPRCB
...
+0x2dfc Tasa de interrupción +0x2e00 :Uint4B
Tabla de temporizadores : _TABLA_DE_TIMER_K

0: kd> dt ntl!_TABLA_DE_TIMER_K
+0x000 Caducidad del temporizador : [64] Ptr64 _KTIMER
+0x200 Entradas del temporizador : [256] _ENTRADA_DE_TABLA_KTIMER

0: kd> dt ntl!_KTIMER
Encabezado +0x000 : _ENCABEZADO_DE_DESPACHO
+0x018 Hora de vencimiento : _NUMERO_UNIDAD
+0x020 Entrada de lista de temporizadores: _LIST_ENTRY
+0x030 Dpc :Ptr64 _KDPC
+0x038 Procesador :Uint4B
+0x03c Period :Uint4B

0: kd> dt ntl!_ENTRADA_TABLA_KTIMER
+0x000 Bloqueo :Uint8B
+0x008 Entrada : _ENTRADA_DE_LISTA
+0x018 Hora : _NUMERO_UNIDAD
```

```
Rutinas relacionadas con el temporizador

ANULAR KeInitializeTimer(
    Temporizador PKTIMER _Fuera_
);

BOOLEAN KeSetTimer(
    _Inout_ Temporizador PKTIMER,
```

```
_En_          LARGE_ENTER Vencimiento de la hora,
PKDPC Dpc en opción
);

BOOLEAN KeSetTimerEx(
    _Inout_ Temporizador PKTIMER,
    _En_      LARGE_ENTER Vencimiento de la hora,
    _En_      Largo periodo,
    PKDPC Dpc en opción
);
```

Se inicializa llamando a `KelInitializeTimer`, que simplemente completa algunos de los campos básicos. Después de la inicialización, el temporizador se puede configurar a través de `KeSetTimer` o `KeSetTimerEx`. La diferencia entre los dos es que `KeSetTimerEx` se puede utilizar para configurar un temporizador recurrente (es decir, que caduque cada X unidad de tiempo). Tenga en cuenta que estas funciones pueden tomar opcionalmente un objeto DPC, que se ejecuta cuando el temporizador caduque. Al llamar a estas rutinas, el temporizador se inserta en una tabla de temporizadores en el PRCB (`TimerTable->TimerListEntry`). Una vez configurado y en cola, un temporizador se puede cancelar y, por lo tanto, eliminar de la tabla de temporizadores. Esto se hace mediante el API `KeCancelTimer`.

¿Cómo sabe el sistema cuándo vence un temporizador? En cada interrupción del reloj, el sistema actualiza su tiempo de ejecución y verifica la lista de temporizadores para ver si hay entradas que vencen; si las hay, solicita una interrupción DPC que procesará las entradas. Por lo tanto, los temporizadores también se procesan en `DISPATCH_LEVEL`.

Hay muchos ejemplos que muestran cómo se utilizan los temporizadores en el sistema operativo. Por ejemplo, el sistema tiene un temporizador periódico que sincroniza la hora del sistema y verifica si la licencia está a punto de expirar (consulte `ExpTimeRefreshDpcRoutine`). Incluso hay un temporizador que expira al final de un siglo (consulte `ExpCenturyDpcRoutine`).

Ceremonias

1. Escriba un controlador para enumerar la lista de módulos cargados cada 10 minutos.
2. Escriba un controlador para enumerar todos los temporizadores del sistema. Asegúrese de que sea compatible con sistemas multinúcleo. Explique por qué los datos de DPC asociados con el temporizador no parecen tener sentido.
3. Explique el campo `DpcWatchDogTimer` en el PRCB.
4. Escriba un controlador que configure un temporizador con un DPC asociado. Explique la secuencia de llamadas que conducen a la ejecución del DPC. Es posible que le interesen las siguientes funciones: `KeUpdateRuntime`, `KeAccumulateTicks`, `KiTimerExpiration`, `KiRetireDpcList` y `KiExpireTimerTable`.
5. Explique cómo funciona la inserción del temporizador. Deberá observar la función

Tabla de temporizador de inserción `KiInsert`.

Devoluciones de llamadas de procesos y subprocesos

Un controlador puede registrar devoluciones de llamadas para una variedad de eventos. Dos de las devoluciones de llamadas más comunes están relacionadas con procesos e hilos, y se pueden registrar a través de API documentadas como PsSetCreateProcessNotifyRoutine, PsSetCreateThreadNotifyRoutine y PsSetLoadImageNotifyRoutine. ¿Cómo funcionan?

Durante la inicialización del sistema, el núcleo llama a la función PsplInitializeCallbacks para inicializar tres matrices globales: PspCreateThreadNotifyRoutine, PspCreateProcessNotifyRoutine y PspLoadImageNotifyRoutine. Cuando el controlador registra una devolución de llamada de proceso, subproceso o imagen, se almacena en una de estas matrices. Además, hay un indicador global, PspNotifyEnableMask, que determina qué tipos de notificaciones están habilitados o deshabilitados. En las rutas de inicialización y terminación de subprocesos (PsplInsertThread y PspExitThread, respectivamente), comprueba si el indicador PspNotifyEnableMask está presente e invoca las devoluciones de llamada en consecuencia.

Estas devoluciones de llamadas se proporcionan principalmente para los controladores y, por lo tanto, el núcleo no las utiliza explícitamente. Por ejemplo, muchos productos de software antivirus registran estas devoluciones de llamadas para supervisar el comportamiento del sistema. Los rootkits en modo núcleo a veces las utilizan junto con los APC para inyectar código en nuevos procesos.

Ceremonias

1. En esta sección se ofrece una explicación general de cómo se implementan las devoluciones de llamadas de notificación de procesos, subprocesos e imágenes. Investigue las siguientes funciones y explique cómo funcionan:
 - Rutina de notificación de subprocesos de PsSetCreate
 - Rutina de notificación de proceso de creación de PsSet
 - Rutina de notificación de carga de imagen PsSet
 - Devoluciones de llamadas de PsplInitialize
2. Si realizó el ejercicio 1, escriba un controlador que enumere todas las rutinas de notificación de procesos, subprocesos e imágenes en el sistema y elimínelas.
3. Si realizó el ejercicio 1, explique dos debilidades principales de estas devoluciones de llamadas de notificación. Por ejemplo, ¿puede crear nuevos procesos/subprocesos sin que estas devoluciones de llamadas lo detecten? Implemente su idea y evalúe su eficacia. Nota: es posible.
4. Si registra una devolución de llamada de carga de imagen con PsSetLoadImageNotifyRoutine, ¿bajo qué condición se la llama? Identifique una debilidad e implemente su idea. Sugerencia: es posible que deba consultar la especificación de PE.

5. Las API PsSetCreateThreadNotifyRoutine, PsSetCreateProcessNotifyRoutine y PsSetLoadImageNotifyRoutine están expuestas por el administrador de procesos . Sin embargo, los administradores de objetos y configuración también exponen sus propias devoluciones de llamadas a través de ObRegisterCallbacks y CmRegisterCallback, respectivamente. Investigue cómo se implementan estas devoluciones de llamadas.
6. Identifique otras devoluciones de llamadas similares documentadas en el WDK e investigue cómo funcionan (procesador, memoria, etc.).

Rutinas de finalización

El modelo de E/S de Windows es el de una pila de dispositivos, en la que los dispositivos se colocan uno sobre otro en capas, y cada capa implementa alguna función específica . Esto significa que los controladores de nivel superior pueden pasar solicitudes a las de nivel inferior para su procesamiento. La capa que complete las solicitudes la marca como completada llamando a IoCompleteRequest. Las rutinas de finalización se utilizan para notificar a los controladores que su solicitud de E/S se ha completado (o que se canceló o falló). Se ejecutan en un contexto de subproceso arbitrario y se pueden configurar a través de las API IoSetCompletionRoutine/Ex . IoSetCompletionRoutine está documentada en WDK, pero nunca aparecerá en una lista de ensamblajes ni en una tabla de importación porque está en línea de forma forzada; un método para identificar la rutina IoSetCompletion es ver el campo CompletionRoutine en una IO_STACK_LOCATION (consulte la siguiente sección) modificada:

Definición de estructura

```
0: kd> dt nt!_UBICACIÓN_DE_PILA_DE_IO
+0x000 Función principal +0x001 :UCar
Función secundaria :UCar
+0x002 Banderas :UCar
+0x003 Control :UCar
+0x008 Parámetros : <etiqueta sin nombre>
+0x028 Objeto de dispositivo : Ptr64 _OBJETO_DISPOSITIVO
+0x030 Objeto de archivo : Ptr64 _OBJETO_ARCHIVO
+0x038 Rutina de finalización: Ptr64 largo :Ptr64 Vacío
+0x040 Context :Ptr64 Vacío
```

Definición de función

```
VACÍO

Rutina de finalización de conjunto de IoT (
    _En_ PIRP Irp,
    _In_opt_ PIO_COMPLETION_ROUTINE Rutina de finalización,
    _In_opt_ __drv_aliasesMem Contexto PVOID,
    _En_ InvokeOnSuccess BOOLEAN,
    _En_ InvokeOnError BOOLEAN,
    _En_ BOOLEAN InvocarAlCancelar
)
{
    UBICACIÓN DE LA PILA PIO IrpSp;
```

```

irpSp = IoGetNextIrpStackLocation(Irp);
irpSp->RutinaDeCompletacion = RutinaDeCompletacion;
irpSp->Contexto = Contexto;
irpSp->Control = 0;
si (Invocar con éxito) {
    irpSp->Control = SL_INVOKE_ON_SUCCESS;
}
si (InvocarEnError) {
    irpSp->Control |= SL_INVOKE_ON_ERROR;
}
si (InvocarAlCancelar) {
    irpSp->Control |= SL_INVOKE_ON_CANCEL;
}
}

```

El administrador de E/S llama a la rutina de finalización registrada como parte de Solicitud completa de lopf.

Aunque el uso legítimo de las rutinas de finalización es obvio, los rootkits pueden usarlas con fines maliciosos. Por ejemplo, pueden configurar una rutina de finalización para modificar el búfer de retorno de un controlador inferior antes de que vuelva al modo de usuario.

Ejercicio

1. Escriba un controlador de prueba utilizando una rutina de finalización y determine dónde se encuentra llamado desde.

Paquetes de solicitud de E/S

Windows utiliza paquetes de solicitud de E/S (IRP) para describir las solicitudes de E/S a los componentes en modo kernel (como los controladores). Cuando una aplicación en modo usuario llama a una API para solicitar datos, el administrador de E/S crea un IRP para describir la solicitud y determina a qué dispositivo enviar el IRP para su procesamiento. Desde el momento en que se crea un IRP hasta que un controlador lo completa, puede haber pasado por varios dispositivos y se podrían haber creado IRP adicionales para satisfacer la solicitud. Se puede pensar en los IRP como la unidad fundamental de comunicación entre dispositivos para las solicitudes de E/S. Un

IRP se define en los encabezados de WDK por el IRP parcialmente opaco.

estructura, pero la mayoría de los campos no están documentados (por lo tanto son parcialmente opacos):

```

0: kd>dt nt!_IRP
+0x000 Tipo :Int2B
...

```

+0x042 Número de pilas	:Carácter
+0x043 Ubicación actual: Char	
...	
+0x058 Superposición	: <etiqueta sin nombre>
+0x068 Cancelar rutina	: Ptr64 vacío
+0x070 Buffer de usuario	:Ptr64 Vacío
+0x078 Cola	: <etiqueta sin nombre>

Desde una perspectiva de programación, un IRP se puede dividir en dos áreas: estática y dinámica. La parte estática es una estructura IRP con información básica sobre la solicitud, como quién solicitó la operación (kernel o usuario), el hilo solicitante y los datos pasados desde el usuario. Los campos Overlay y Tail son uniones que contienen metadatos sobre la solicitud. La parte dinámica está inmediatamente después del encabezado; es una matriz de estructuras IO_STACK_LOCATION que contienen información de solicitud específica del dispositivo. Una IO_STACK_LOCATION contiene la función principal y secundaria del IRP , los parámetros para la solicitud y una rutina de finalización opcional. Similar al IRP, es una estructura parcialmente opaca:

0: kd> dt nt!_UBICACIÓN_DE_PILA_DE_IO	
+0x000 Función principal +0x001	:UChar
Función secundaria	:UChar
+0x002 Banderas	:UChar
+0x003 Control	:UChar
+0x008 Parámetros	: <etiqueta sin nombre>
+0x028 Objeto de dispositivo	: Ptr64 _OBJETO_DISPOSITIVO
+0x030 Objeto de archivo	: Ptr64 _OBJETO_ARCHIVO
+0x038 Rutina de finalización: Ptr64 largo	
+0x040 Context	:Ptr64 Vacío

El campo Parámetros es una unión porque el parámetro depende del número de función principal y secundaria. Windows tiene una lista predefinida de funciones principales y secundarias genéricas para describir todos los tipos de solicitud. Por ejemplo, una solicitud de lectura de archivo conducirá a un IRP creado con la función principal IRP_MJ_READ; cuando Windows solicita la entrada del controlador de clase de teclado, también utiliza IRP_MJ_READ. Cuando el administrador de E/S crea un IRP, determina cuántas funciones IO_STACK_LOCATION estructuras para asignar en función de la cantidad de dispositivos que hay en la pila de dispositivos actual. Cada dispositivo es responsable de preparar la IO_STACK_LOCATION para el siguiente. Recuerde que un controlador puede configurar una rutina de finalización con la API IoSetCompletionRoutine ; en realidad, se trata de una rutina en línea que configura el campo CompletionRoutine en IO_STACK_LOCATION.

La figura 3-8 ilustra la relación entre estas dos estructuras en un IRP.



Figura 3-8

Tenga en cuenta que la ubicación de la pila “próxima” es el elemento inmediatamente superior al “actual” (no después). Es importante saber esto porque las rutinas de ubicación de pila como `IoGetCurrentIrpStackLocation`, `IoSkipCurrentIrpStackLocation`, `IoGetNextIrpStackLocation` y otros simplemente devuelven punteros a estos elementos de la matriz utilizando aritmética de punteros.

Aunque los IRP normalmente los genera el administrador de E/S en respuesta a solicitudes de los usuarios u otros dispositivos, también pueden crearse desde cero. y se envía a otros dispositivos para su procesamiento. Un controlador puede asignar un IRP con `IoAllocateIrp`, asociarlo con un hilo, completar el código principal y secundario del IRP, configurar el recuento/tamaño de `IO_STACK_LOCATION`, completar los parámetros y enviarlo a el dispositivo de destino para el procesamiento con `IoCallDriver`. Algunos rootkits utilizan Este mecanismo envía directamente solicitudes al controlador del sistema de archivos para evitar el enganche de llamadas del sistema. En el ejercicio analizarás uno de estos rootkits.

Estructura de un controlador

Un controlador es una pieza de software que interactúa con el núcleo y/o controla Recursos de hardware. Si bien existen muchos tipos diferentes de controladores, nos interesan principalmente los siguientes tipos de controladores en modo kernel:

- Controlador de software heredado : software que se ejecuta en el anillo 0 e interactúa con El núcleo a través de interfaces documentadas y no documentadas. La mayoría de los rootkits y controladores de seguridad son de este tipo.

- Controlador de filtro heredado : controladores que se conectan a un controlador existente y modifican su entrada.
- Controlador de minifiltro del sistema de archivos : controladores que interactúan con el sistema de archivos para interceptar las solicitudes de E/S de archivos. La mayoría del software antivirus utiliza este tipo de controlador para interceptar las escrituras y lecturas de archivos con fines de escaneo; el software de cifrado en disco se implementa normalmente a través de este mecanismo.

El modelo estándar para los controladores de Windows es el Modelo de controlador de Windows (WDM). WDM define tanto un conjunto de interfaces que los controladores deben implementar como reglas a seguir para interactuar de forma segura con el núcleo. Se ha definido desde Windows 2000 y todos los controladores que analiza se basan en él. Debido a que escribir controladores de hardware plug-and-play confiables con administración de energía completa y manejar todas las idiosincrasias de sincronización utilizando interfaces WDM puras es extremadamente difícil, Microsoft presentó el marco Windows Driver Foundation (WDF). WDF es básicamente un conjunto de bibliotecas construidas sobre WDM que simplifica el desarrollo de controladores al proteger a los desarrolladores de la interacción directa con WDM. WDF se divide en dos categorías: marco de controlador de modo kernel (KMDF) y marco de controlador de modo usuario (UMDF). KMDF está destinado a controladores de modo kernel (como teclados y dispositivos USB) y UMDF es para controladores de modo usuario (como controladores de impresora). Este libro trata solo con controladores basados en el modelo WDM.

Se puede pensar en un controlador como una DLL que se carga en la dirección del núcleo. El espacio de ejecución del controlador es limitado y se ejecuta con los mismos privilegios que el núcleo. Tiene un punto de entrada bien definido y puede registrar rutinas de despacho para atender solicitudes de usuarios u otros controladores. Tenga en cuenta que un controlador no tiene un hilo de ejecución principal; simplemente contiene código que puede ser llamado por el núcleo en determinadas circunstancias . Por eso, los controladores suelen tener que registrar rutinas de despacho con el administrador de E/S (consulte la siguiente sección). Al analizar los controladores, la primera y más importante tarea es identificar estas rutinas de despacho y comprender cómo interactúan con el núcleo.

Puntos de entrada

Todos los controladores tienen un punto de entrada llamado DriverEntry, que se define de la siguiente manera:

```
Entrada del conductor  
ESTADO DEL NT  
Entrada del controlador (  
    PDRIVER_OBJECT Objeto del controlador,  
    PUNICODE_STRING Ruta de registro  
);  
  
OBJETO_CONDUCTOR  
  
tipo de definición de estructura _DRIVER_OBJECT {  
    Tipo CSHORT;
```

```

Tamaño CSHORT;
PDEVICE_OBJECT ObjetoDispositivo;
Banderas ULONG;
Controlador PVOIDStart;
Tamaño del controlador ULONG;
Sección del controlador PVOID;
PDRIVER_EXTENSION Extensión del controlador;
UNICODE_STRING NombreConductor;
PUNICODE_STRING Base de datos de hardware;
PFAST_IO_DISPATCH Despacho rápido de IO;
PDRIVER_INITIALIZE Inicio del controlador;
PDRIVER_STARTIO ControladorStartIo;
PDRIVER_UNLOAD Descarga del controlador;
PDRIVER_DISPATCH Función principal[IRP_MJ_FUNCIÓN_MÁXIMA + 1];
} OBJETO_CONTROLADOR, *OBJETO_CONTROLADOR;

```

NOTA: Técnicamente, el punto de entrada no tiene que llamarse DriverEntry.

Cuando es necesario cargar un controlador, su imagen se asigna a la memoria del espacio del núcleo, se crea un objeto de controlador para él y se registra en el administrador de objetos y, a continuación, el administrador de E/S llama al punto de entrada. DRIVER_OBJECT es una estructura que completa el administrador de E/S durante el proceso de carga del controlador; la documentación oficial indica que es una estructura parcialmente opaca, pero se puede ver su definición completa en los archivos de encabezado. DriverInit se establece en el punto de entrada del controlador y el administrador de E/S llama directamente a este campo. La responsabilidad principal de DriverEntry es inicializar los ajustes específicos del controlador y registrar las rutinas de despacho de IRP según sea necesario. Estas rutinas se almacenan en la matriz MajorFunction . Como se mencionó anteriormente, Windows tiene un conjunto predefinido de funciones principales de IRP para describir de forma genérica cada solicitud de E/S; siempre que llega una solicitud de E/S para el controlador, el administrador de E/S llama al controlador de función principal de IRP adecuado para procesar la solicitud. Por lo tanto, es común ver un código como el siguiente en DriverEntry:

```

DriverObject->MajorFunction[IRP_MJ_CREATE] = CrearManejadorCerrar;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = CrearManejadorCerrar;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ControladorDeControlDeDispositivo;
...

```

Tenga en cuenta que la misma rutina de envío se puede especificar para varias funciones principales de IRP. A veces se inicializarán en un bucle:

```

para (i=0; i<IRP_MJ_MÁXIMO; i++) {
    DriverObject->MajorFunction[i] = ManejadorGenérico;
}
DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateHandler;
DriverObject->FunciónPrincipal[IRP_MJ_PNP] = PnpHandler;
...

```

Si no inicializa la tabla MajorFunction, contendrá el controlador predeterminado lopInvalidDeviceRequest , que simplemente devuelve un error al solicitante.

Si un controlador admite la descarga dinámica, también debe completar el formulario DriverUnload. campo; de lo contrario, el controlador permanecerá en la memoria para siempre (hasta que se reinicie). Una rutina DriverUnload normalmente realiza tareas de limpieza específicas del controlador. Muchos rootkits no registran una rutina de descarga.

RegistryPath es la ruta de registro del controlador. Se crea como parte del Proceso normal de instalación del controlador.

Objetos de controlador y dispositivo

La sección anterior indica que el administrador de E/S crea un DRIVER_OBJECT para cada controlador cargado en el sistema. Un controlador puede elegir crear uno o más objetos de dispositivo. Los objetos de dispositivo se definen mediante el objeto DEVICE_OBJECT parcialmente opaco. estructura:

```
tipo de definición de estructura _DEVICE_OBJECT {
    Tipo CSHORT;
    Tamaño USHORT;
    LARGO ReferenciaCount;
    estructura _DRIVER_OBJECT *ObjetoConductor;
    estructura _DEVICE_OBJECT *NextDevice;
    estructura _DEVICE_OBJECT *DispositivoAdjunto;
    estructura _IRP *CurrentIrp;
    ...
    Extensión de dispositivo PVOID;
    TIPO_DE_DISPOSITIVO TipoDeDispositivo;
    CCHAR Tamaño de pila;
    ...
    ULONG Cuenta de subprocessos activos;
    PSECURITY_DESCRIPTOR Descriptor de seguridad;
    ...
    PVOID Reservado;
} OBJETO_DISPOSITIVO, *POBJETO_DISPOSITIVO;
```

DriverObject es el objeto de controlador asociado con este objeto de dispositivo. Si el controlador creó más de un objeto de dispositivo, NextDevice apuntará al siguiente objeto de dispositivo en la cadena. Un controlador puede crear varios objetos de dispositivo para administrar los diferentes recursos de hardware que maneja. Si no se crean objetos de dispositivo , nadie puede enviar solicitudes al dispositivo. Normalmente, los controladores crearán objetos de dispositivo en DriverEntry a través de la API IoCreateDevice .

DeviceExtension es un puntero a datos específicos del dispositivo almacenados en un grupo no paginado. Su tamaño se especifica como parámetro para IoCreateDevice. Los desarrolladores suelen almacenar aquí información de contexto o datos importantes sobre el controlador y otros dispositivos relacionados. La recuperación de la estructura de extensión del dispositivo es probablemente la segunda tarea más importante en el análisis de controladores.

Un controlador puede “adjuntar” uno de sus propios objetos de dispositivo a otro objeto de dispositivo para que reciba solicitudes de E/S destinadas al objeto de dispositivo de destino. Por ejemplo, si el dispositivo A se adjunta al dispositivo B, todas las solicitudes de IRP enviadas a B se enrutarán primero a A. Este mecanismo de asociación se utiliza para admitir controladores de filtro para que puedan modificar o inspeccionar solicitudes a otros controladores. El campo AttachedDevice apunta al dispositivo al que está adjunto el objeto de dispositivo actual. La asociación de dispositivos se realiza a través de la familia de API IoAttachDevice .

Manejo de IRP

Como se mencionó anteriormente, DriverEntry generalmente registra rutinas de despacho para manejar varias funciones principales de IRP. El prototipo de estas rutinas de despacho es el siguiente:

```
NTSTATUSXXX_Envío  
(     PDEVICE_OBJECT Objeto del dispositivo,  
PIRP (Irp);
```

El primer argumento es el objeto del dispositivo de destino de la solicitud. El segundo argumento es el IRP que describe la solicitud.

Una rutina de despacho normalmente determina primero qué función principal de IRP recibió y luego determina los parámetros de la solicitud. Esto lo hace verificando IO_STACK_LOCATION en el IRP. Si la rutina de despacho completa exitosamente la solicitud, llama a IoCompleteRequest y retorna. Si no puede completar la solicitud, entonces tiene tres opciones: retorna un error, pasa el IRP a otro controlador o deja pendiente el IRP. Por ejemplo, un controlador de filtro puede elegir procesar solo las solicitudes IRP_MJ_READ por sí mismo y pasar todas las demás solicitudes al dispositivo conectado. Un controlador puede pasar IRP a otro controlador a través de la API IoCallDriver .

Porque los parámetros IRP para cada solicitud se almacenan en su propio IO_STACK_LOCATION: un controlador debe asegurarse de acceder a la ubicación correcta. Esto se hace a través de la API IoGetCurrentIrpStackLocation . Si el controlador desea pasar el mismo IRP a otro controlador, debe copiar los parámetros actuales a la siguiente IO_STACK_LOCATION (IoCopyCurrentIrpStackLocationToNext) o pasar el parámetro al siguiente controlador (IoSkipCurrentStackLocation).

Un mecanismo común para la comunicación entre el usuario y el núcleo

Se utilizan muchos mecanismos para facilitar la comunicación entre el usuario y el núcleo. Por ejemplo, un controlador puede comunicarse con el código en modo usuario a través de una región de memoria compartida con doble asignación en el espacio del usuario y del núcleo. Otro método es que el controlador cree un evento que un subproceso en modo usuario pueda esperar; el estado del evento se puede utilizar como un disparador para una acción posterior. Otro método (aunque un poco chapucero) es a través del manejo de interrupciones. Un controlador puede configurar manualmente un manejador de interrupciones personalizado.

en el código de modo usuario e IDT se puede activar con la instrucción INT ; probablemente nunca verá esta técnica utilizada en un controlador comercial.

Si bien el mecanismo de comunicación preciso depende del objetivo final del desarrollador, normalmente se utiliza una interfaz genérica documentada para el intercambio de datos entre el usuario y el núcleo. Este mecanismo es compatible con IRP_MJ_DEVICE_CONTROL

Operación y comúnmente conocida como control de E/S del dispositivo o simplemente IOCTL. Funciona de la siguiente manera:

1. El controlador define uno o más códigos IOCTL para cada operación que admite.
2. Para cada operación admitida, el controlador especifica cómo debe acceder a la entrada del usuario y devolver los datos al usuario. Hay tres métodos de acceso: E/S almacenada en búfer, E/S directa y ninguno de ellos. Estos métodos se tratan en la siguiente sección.
3. Dentro del controlador IRP_MJ_DEVICE_CONTROL , el controlador recupera el código IOCTL de su IO_STACK_LOCATION y procesa los datos según el método de entrada.

El código de modo de usuario puede solicitar estas operaciones IOCTL a través de la API DeviceloControl .

Métodos de amortiguación

Un controlador puede acceder a un búfer de modo de usuario utilizando uno de los siguientes tres métodos:

- E/S con búfer : en el núcleo, se denomina METHOD_BUFFERED . Cuando se utiliza este método, el núcleo valida que el búfer de usuario se encuentre en la memoria de modo de usuario accesible, asigna un bloque de memoria en un grupo no paginado y copia el búfer de usuario en él. El controlador accede a este búfer de modo de núcleo a través del campo AssociatedIrp.SystemBuffer en la estructura IRP. Mientras procesa la solicitud, el controlador puede modificar el búfer del sistema (tal vez necesite devolver algunos datos al usuario); después de completar la solicitud, el núcleo copia el contenido del búfer del sistema nuevamente al búfer de modo de usuario y libera automáticamente el búfer del sistema.
- E/S directa : esto se conoce como METHOD_IN_DIRECT o METHOD_OUT_DIRECT en el núcleo. El primero se utiliza para pasar datos al controlador; el segundo se utiliza para obtener datos del controlador. Este método es similar a la E/S almacenada en búfer, excepto que el controlador obtiene un MDL que describe el búfer del usuario. El administrador de E/S crea el MDL y lo bloquea en la memoria antes de pasarlo al controlador. Los controladores pueden acceder a este MDL a través del campo MdlAddress de la estructura IRP.
- Ninguno: en el núcleo, se lo denomina METHOD_NEITHER . Cuando se utiliza este método, el administrador de E/S no realiza ningún tipo de validación de los datos del usuario; pasa los datos sin procesar al controlador. Los controladores pueden acceder a los datos a través de Parameters.DeviceloControl.Type3InputBuffer

152 Capítulo 3 ■ El núcleo de Windows

campo en su IO_STACK_LOCATION. Si bien este método puede parecer el más rápido de los tres (ya que no hay validación ni asignación de buffers adicionales), es sin duda el más inseguro. Deja toda la validación al desarrollador. Sin una validación adecuada, un controlador que utilice este método puede exponerse a vulnerabilidades de seguridad, como corrupción de la memoria del núcleo o fugas/divulgación.

No existe una regla escrita para determinar qué método utilizar en los controladores, ya que depende de los requisitos específicos del controlador. Sin embargo, en la práctica, la mayoría de los controladores de software utilizan E/S con búfer, ya que proporciona un buen equilibrio entre simplicidad y seguridad. La E/S directa es común en los controladores de hardware porque se puede utilizar para pasar grandes fragmentos de datos sin sobrecarga de búfer.

Código de control de E/S

Un código IOCTL es un entero de 32 bits que codifica el tipo de dispositivo, el código específico de la operación, el método de almacenamiento en búfer y el acceso de seguridad. Los controladores suelen definir los códigos IOCTL a través de la macro CTL_CODE :

```
#define CTL_CODE( TipoDeDispositivo, Función, Método, Acceso ) \
    (((TipoDeDispositivo) << 16) | ((Acceso) << 14) | ((Función) << 2) | (Método) \\\
```

DeviceType suele ser una de las constantes FILE_DEVICE_*, pero para controladores de terceros puede usar cualquier valor superior a 0x8000. (Este es solo el valor recomendado y no hay nada que lo imponga). Access especifica operaciones genéricas de lectura/escritura permitidas por IOCTL; puede ser una combinación de FILE_ANY_ACCESS, FILE_READ_ACCESS y FILE_WRITE_ACCESS. La función es el código IOCTL específico del controlador; puede ser cualquier valor superior a 0x800. El método especifica uno de los métodos de almacenamiento en búfer.

Una forma típica de definir un código IOCTL es la siguiente:

```
#define FILE_DEVICE_GENIOCTL 0xa000 // nuestro tipo de dispositivo
#define PROCESO_GENIOCTL          0x800 // nuestro código IOCTL especial

#define IOCTL_PROCESS CTL_CODE(FILE_DEVICE_GENIOCTL, \
                           PROCESO_GENIOCTL, \
                           MÉTODO_ALMACENADO_EN_BUFER, DATOS_LEIDOS_DE_ARCHIVO)
```

Esto define un IOCTL llamado IOCTL_PROCESS para un controlador personalizado que utiliza MÉTODO_BUFFERED.

Al analizar un controlador, es importante descomponer el IOCTL en su tipo de dispositivo, código, acceso y método de almacenamiento en búfer. Esto se puede lograr con un par de macros simples documentadas:

```
#define TIPO_DE_DISPOSITIVO DESDE_CÓDIGO_CTL(ctrlCode) \
    (((ULONG)(ctrlCode) & 0xffff0000) >> 16)
#define MÉTODO_DE_CÓDIGO_CTL(ctrlCode)           ((ULONG)(ctrlCode) & 3)
```

Mecanismos de sistemas diversos

En esta sección se analizan construcciones que, si bien no son esenciales para comprender los controladores del núcleo, se observan con frecuencia en los controladores de la vida real.

Registros de control del sistema

Para lograr sus objetivos, muchos desarrolladores de rootkits recurren a funciones de enganche en el núcleo. Sin embargo, todo el código del núcleo está asignado como de sólo lectura, por lo que al aplicarle un parche se producirá una comprobación de errores. En x86/x64, este mecanismo de protección se aplica en realidad a nivel de hardware a través de un registro de control especial: CR0. CR0 determina varias configuraciones importantes del procesador, como si está en modo protegido y si la paginación está habilitada; también determina si la CPU puede escribir en páginas de sólo lectura (bit WP). CR0 solo es accesible por el código que se ejecuta en el anillo 0. De forma predeterminada, Windows activa el bit WP , que prohíbe las escrituras en páginas marcadas como de sólo lectura.

NOTA: En x64 y ARM, hay una característica de Windows llamada Protección de parches de kernel, También conocido como PatchGuard, que intenta detectar ganchos y modificaciones a varios estructuras de datos críticas para la seguridad y comprueba errores en la máquina. Por lo tanto, no es común ver ganchos en estas plataformas en los controladores de producción/envío. Sin embargo, los ganchos aún prevalecen porque hay muchas máquinas x86, por lo que los encontrará con frecuencia.

Existen varias formas de evitar esta restricción y la más sencilla es desactivar el bit WP . Por ello, verás con frecuencia este patrón de código en los rootkits.

Por ejemplo, Muestra G:

```
01: .text:0001062F empujar    fácil
02:.text:00010630 mov         EAX, CR0
03:.text:00010633 mov         [esp+8+var_4], fácil
04:.text:00010637 y           eax, 0FFFFEFFFFh
05:.text:0001063C mov         cr0, eax
06: .text:0001063F emergente  fácil
```

Las líneas 2 y 3 copian CR0 a EAX y lo guardan en una variable local. Las líneas 4 y 5 desactivan el bit 16 en EAX y escribirlo nuevamente en CR0. El bit 16 en CR0 es el bit WP.

Existen al menos otras dos soluciones que no modifican directamente CR0. Implican MDL y conocimiento de la plataforma MMU. Deberá realizar esto como uno de los ejercicios.

Tabla de descriptores de servicios KeService

Como se mencionó anteriormente, muchos rootkits recurren a interceptar llamadas del sistema. Sin embargo, como aprendiste, las llamadas del sistema se identifican mediante un número que se usa como índice.

en una tabla de llamadas al sistema. Además, la tabla de llamadas al sistema (KiServiceTable) no se exporta, por lo que no hay una manera sencilla de acceder a ella desde un controlador. ¿Cómo solucionan esto los autores de rootkits?

El núcleo exporta el símbolo KeServiceDescriptorTable , que contiene una estructura KSERVICE_TABLE_DESCRIPTOR con la información de la llamada del sistema. (Recuerde que en x64, este símbolo no se exporta). Así es como la mayoría de los rootkits acceden a la tabla de llamadas del sistema. El siguiente paso es identificar dónde se encuentra la llamada del sistema de destino. Recuerde que las llamadas del sistema se identifican por un número, no por un nombre. Los autores de rootkits tienen varias formas de encontrar la llamada al sistema correcta. Una forma es codificar de forma rígida el índice de llamadas al sistema. Otro método es desensamblar el fragmento de código de la llamada al sistema y obtener el índice desde allí. Ambos métodos tienen una desventaja: son fáciles de implementar, pero dependen de patrones de código o datos que pueden cambiar de un Service Pack a otro; pueden ser confiables en algunas plataformas, pero sin duda provocarán inestabilidad del sistema en otras. A pesar de la falta de confiabilidad, estos dos métodos son utilizados con frecuencia por los rootkits en la red. Por ejemplo, Sample G tiene el siguiente código:

```
01: .text:000117D4 sub_117D4 proc cerca
02: .text:000117D4 push
03: .text:000117D5 mov
04: .text:000117D7 push
05: .text:000117D8 mov
06: .text:000117DE mov
07: .text:000117E0 push
08: .text:000117E1 mov
09: ...
10: .text:00011808 llamada
11: .text:0001180D mov
12: .text:00011813 mov
13: .text:00011816 mov
14: .text:00011818 mov
15: ...
16: .text:00011836 sub_117D4 fin
```

Las líneas 5 a 10 guardan la dirección de KiServiceTable en ECX, guardan la dirección de ZwQuerySystemInformation en ESI y deshabilitan el bit WP . La línea 12 recupera el segundo byte de ZwQuerySystemInformation; lo hace porque supone que la primera instrucción en la función mueve el número de llamada al sistema a un registro y, por lo tanto, el valor de 32 bits después del código de operación contiene el número de llamada al sistema real (consulte la siguiente barra lateral). Las líneas 13 y 14 sobrescriben esa entrada de llamada al sistema en la tabla de servicios con una nueva función: sub_1123e. Todas las llamadas a ZwQuerySystemInformation Ahora será redirigido a sub_1123e.

NOTA Mencionamos anteriormente que la línea 12 recupera el segundo byte de ZwQuerySystemInformation. En Windows 7 de 32 bits, la primera instrucción en

ZwQuerySystemInformation es b805010000 mov eax y 105h. b8 es el código de operación MOV, mientras que 05010000 (0x105) codifica el inmediato, que en este caso es el número de llamada al sistema.

Secciones

Una sección es un objeto que se utiliza para describir la memoria respaldada por alguna forma de almacenamiento. La sección puede estar respaldada por un archivo normal o un archivo de paginación. Una sección respaldada por un archivo es aquella cuyo contenido de memoria es el de un archivo en el disco; si hay modificaciones en la sección, se realizarán directamente en el disco. Una sección respaldada por un archivo de paginación es aquella cuyo contenido está respaldado por el archivo de paginación; las modificaciones a dicha sección se descartarán después de que se cierre. Un controlador puede crear una sección con la API ZwCreateSection y luego asignar una vista de la misma a otro proceso con ZwMapViewOfSection. Cada vista es básicamente un rango de direcciones virtuales que se puede utilizar para acceder a la memoria representada por el objeto de sección asociado. Por lo tanto, puede haber múltiples vistas para una sección.

Recorridos paso a paso

Ahora que ya tiene una sólida comprensión de los conceptos de kernel y controladores de Windows, es hora de aplicar ese conocimiento mediante el análisis de algunos rootkits reales. Esta sección tiene dos propósitos: explicar el proceso de pensamiento de la ingeniería inversa en modo kernel y demostrar la aplicación de las técnicas de desarrollo de controladores para comprender los rootkits.

Los rootkits se presentan en muchas formas diferentes. Algunos interceptan llamadas del sistema, otros ocultan archivos filtrando respuestas de E/S, algunos interceptan comunicaciones de red, algunos registran pulsaciones de teclas, etc. Sin embargo, como todos los controladores, comparten la misma estructura genérica; por ejemplo, todos tienen una función DriverEntry con controladores de envío IRP opcionales que interactúan con el núcleo a través de interfaces documentadas y no documentadas. Con este conocimiento, puede diseccionar los componentes principales de un controlador y analizarlos sistemáticamente. El proceso de análisis general es el siguiente:

1. Identifique DriverEntry y determine los controladores de despacho de IRP, si los hay.
2. Determine si el controlador se conecta a otro dispositivo para filtrar o interceptar sus solicitudes de E/S. Si es así, ¿cuál es el dispositivo de destino?
3. Si el controlador crea un objeto de dispositivo, determine el nombre y la extensión del dispositivo. tamaño de la sión.
4. Recupere la estructura de extensión del dispositivo observando cómo se forman sus miembros de campo. se utilizan

5. Si el controlador admite IOCTL, identifique todos los códigos IOCTL y su funcionalidad correspondiente. Determine qué método de almacenamiento en búfer utilizan.
6. Identifique DPC, elementos de trabajo, APC, temporizadores, rutinas de finalización, devoluciones de llamadas y subprocesos del sistema.
7. Trata de entender cómo encajan todas las piezas.

Un rootkit x86

El recorrido comienza con la muestra A.

Su DriverEntry comienza en 0x105F0 y termina en 0x106AD. Primero inicializa una estructura UNICODE_STRING con las cadenas \Device\fsodhfn2m y \DosDevices\fsodhfn2m. En el modo kernel, la mayoría de las cadenas se describen utilizando UNICODE_STRING estructura:

```
estructura de tipo definido _UNICODE_STRING {
    USHORT Longitud;
    USHORT LongitudMáxima;
    Búfer PWSTR;
} CADENA UNICODE, *CADENA_PUNICODE;
```

Se inicializa a través de la API RtlInitUnicodeString . La cadena “Device” es un nombre de dispositivo en el administrador de objetos; la cadena “DosDevices” se utiliza como un enlace simbólico al nombre real del dispositivo. El administrador de objetos de Windows mantiene y organiza los objetos en una estructura similar a un sistema de archivos con la raíz en “”. Hay directorios bien definidos como \Devices, \BaseNamedObjects, \

KernelObjects, etc. \DosDevices es un alias para el directorio \?? ; está ahí porque cuando las aplicaciones en modo usuario especifican la ruta a un objeto al que quieren acceder, se le antepone \??\ ; \?? contiene enlaces simbólicos que apuntan al objeto real. Por ejemplo, cuando un usuario quiere acceder a “c:\test.txt” a través de la API CreateFile , la ruta real enviada al kernel es “\??\c:\

test.txt”; debido a que “c:” es un enlace simbólico a \Device\HarddiskVolume2 (puede variar en su sistema), la ruta completa eventualmente se resolverá en \Device\HarddiskVolume2\test.txt. El enlace simbólico es necesario porque las API de modo de usuario suelen acceder a los dispositivos a través del directorio \?? ; si no hubiera enlaces simbólicos allí, es posible que el dispositivo no sea accesible para las aplicaciones de modo de usuario.

Después de inicializar las dos cadenas, procede a crear el objeto del dispositivo real. IoCreateDevice se define de la siguiente manera:

```
ESTADO DEL NT
Dispositivo de creación de Io (
    EN PDRIVER_OBJECT ObjetoConductor,
    En ULONG DeviceExtensionSize,
    EN PUNICODE_STRING NombreDeDispositivo OPCIONAL,
    EN TIPO DE DISPOSITIVO Tipo_de_dispositivo,
    EN ULONG Características del dispositivo,
```

```
EN BOOLEAN Exclusivo,
FUERA PDEVICE_OBJECT *ObjetoDispositivo
);
```

DriverObject es el DRIVER_OBJECT del llamador; es el objeto de controlador con el que está asociado el nuevo objeto de dispositivo. DeviceExtensionSize es la cantidad de bytes de memoria de grupo no paginado que se deben asignar para la estructura específica del controlador. Debido a que es una estructura definida por el usuario, es muy importante recuperar sus campos. DeviceName es el nombre del dispositivo nativo. DeviceType es uno de los FILE_DEVICE_* predefinidos tipos; si el dispositivo no entra en una categoría genérica, FILE_DEVICE_UNKNOWN. En su lugar, se utiliza DeviceCharacteristics. DeviceCharacteristics hace referencia a la característica del dispositivo; la mayoría de las veces verá FILE_DEVICE_SECURE_OPEN. Exclusive determina si puede haber más de un identificador para el dispositivo. DeviceObject recibe el objeto de dispositivo real.

A partir del desmontaje, puedes descompilar el primer bloque básico y su condición de salida de la siguiente manera:

```
01: UNICODE_STRING nombre_desarrollo;
02: UNICODE_STRING nombre_simbolo;
03:
04: NTSTATUS DriverEntry(PDRIVER_OBJECT Objeto del controlador, \
                         PUNICODE_STRING ruta de registro)
05: {
06: Estado NTSTATUS;
07: PDEVICE_OBJECT devobj;
08:
09: RtlInitUnicodeString(&nombre_de_dispositivo, L"\Device\fsodhfn2m");
10: RtlInitUnicodeString(&symname, L"\DosDevices\fsodhfn2m");
11: estado = IoCreateDevice(
12:             Objeto del controlador,
13:             0,
14:             &nombre_de_desarrollo,
15:             ARCHIVO_DISPOSITIVO_DESCONOCIDO,
16:             ARCHIVO_DISPOSITIVO_SEGURO_ABIERTO,
17:             FALSO,
18:             &devobj);
19:     si (!NT_SUCCESS(estado)) {
20:         estado de retorno; // loc_106A3
21:     }
22: }
```

NT_SUCCESS() es una macro común que verifica si el estado es mayor o igual a 0. Después de crear correctamente el objeto, procede a lo siguiente:

01:.texto:00010643 mov	ecx, [ebp+Objeto del controlador]
02:.texto:00010646 mov	dword ptr [ecx+38h], desplazamiento sub_10300
03:.texto:0001064D mov	edx, [ebp+Objeto del controlador]
04:.texto:00010650 mov	dword ptr [edx+40h], desplazamiento sub_10300
05:.texto:00010657mov	eax, [ebp+ObjetoControlador]
06:.texto:0001065A movimiento	dword ptr [eax+70h], desplazamiento sub_10300

158 Capítulo 3 ■ El núcleo de Windows

```

07: .text:00010661 mov         ecx, [ebp+Objeto del controlador]
08: .text:00010664 mov         dword ptr [ecx+34h], desplazamiento sub_10580
09: .text:0001066B push        desplazamiento NombreDeEnlaceSimbólico ; NombreDeEnlaceSimbólico
10: .text:00010670 llamar     ds:IoDeleteSymbolicLink
11: .text:00010676 push        desplazamiento DestinationString; NombreDeDispositivo
12: .text:0001067B push 13: .text:00010680
    llamada      desplazamiento NombreDeEnlaceSimbólico ; NombreDeEnlaceSimbólico
13: .text:0001068D jge        ds:IoCreateSymbolicLink
14: .text:0001068E mov         [ebp+var_4], eax
15: .text:00010689 cmp         [ebp+var_4], 0
16: .text:0001068D jge        loc_106A1 corto

```

Las líneas 1 a 8 establecen algunos campos DRIVER_OBJECT en dos punteros de función. ¿Qué es lo que está en el desplazamiento 0x38, 0x40, 0x70 y 0x34?

```

0: kd> dt _OBJETO_DRIVER
_nt!_OBJETO_CONTROLADOR
  +0x000 Tipo +0x002 :Int2B
  Tamaño :Int2B
  +0x004 Objeto de dispositivo : Ptr32 _OBJETO_DISPOSITIVO
  ...
  +0x034 Descarga del controlador :Ptr32          vacío
  +0x038 Función principal : [28] Ptr32           largo

```

El desplazamiento 0x34 es la rutina DriverUnload ; ahora, usted sabe que el controlador soporta la descarga dinámica y sub_10580 es la rutina de descarga. El desplazamiento 0x38 es el comienzo de la matriz MajorFunction ; recuerde que esta es una matriz de controladores de despacho de IRP. Debido a que hay un máximo de 28 funciones principales genéricas de IRP , la matriz MajorFunction tiene 28 miembros. El primer índice es 0, que corresponde a IRP_MJ_CREATE; por lo tanto, usted sabe que sub_10300 es el controlador para ese IRP. El desplazamiento 0x40 es el tercer elemento en la matriz MajorFunction (índice 2); esto corresponde a IRP_MJ_CLOSE, y sub_10300 se reutiliza como el controlador.

El desplazamiento 0x70 es el elemento número 16 de la matriz (índice 0xe), que corresponde a IRP_MJ_DEVICE_CONTROL, y sub_10300 es el controlador. En este punto, ya sabes que sub_10300 es el controlador para el IRP de lectura, cierre y control del dispositivo.

Las líneas 10 a 13 eliminan cualquier enlace simbólico existente y crean uno nuevo para apuntar al objeto de dispositivo creado previamente.

Ahora puede continuar descompilando este bloque en DriverEntry de la siguiente manera:

```

01: DriverObject->FunciónPrincipal[IRP_MJ_READ] = sub_10300;
02: DriverObject->FunciónPrincipal[IRP_MJ_CLOSE] = sub_10300;
03: DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = sub_10300;
04: DriverObject->DriverUnload = sub_10580;
05:
06: IoDeleteSymbolicLink(&nombre de sym);
07: estado = IoCreateSymbolicLink(&symname, &devname);
08: si (INT_SUCCESS(estado)) {
09:     ... // bloque .text:0001068F
10: }

```

```
11: }
12: estado de retorno;
```

Para facilitar la vida, puede cambiar el nombre de sub_10300 a IRP_ReadCloseDeviceo y sub_10580 como DRV_Unload.

El siguiente bloque en 0x1068F elimina el objeto de dispositivo creado anteriormente si falla la creación del vínculo simbólico. Tenga en cuenta que obtiene el objeto de dispositivo del objeto del controlador en lugar de usar el puntero pasado a IoCreateDevice. Puede descompilar este bloque de la siguiente manera:

```
01: IoDeleteDevice(ObjetoControlador->ObjetoDispositivo);
```

Con esto se completa la descompilación de DriverEntry de este rootkit . Para resumir lo que se ha aprendido hasta ahora:

- El controlador crea un objeto de dispositivo llamado \Device\fsodhfn2m.
- Admite descarga dinámica y la rutina de descarga es sub_10580 (renombrado a DRV_Unload).
- Admite operaciones IRP_MJ_READ, IRP_MJ_WRITE e IRP_MJ_DEVICE_CONTROL , y sub_10300 es el controlador (renombrado a IRP_ReadCloseDeviceo).
- Crea un enlace simbólico al objeto del dispositivo. Si esto falla, el controlador devuelve un error

El siguiente paso es comprender qué hace la rutina DriverUnload . El WDK define el prototipo para la rutina de descarga del controlador de la siguiente manera:

```
VACIO
Descargar(
    PDRIVER_OBJECT Objeto del controlador
);
```

Después de un pequeño masaje, nuestra rutina de descarga se ve así:

```
01: .text:00010580 ; void __stdcall DRV_Unload(PDRIVER_OBJECT drvobj)
02: .text:00010580 DRV_Unload proc cerca
03: .text:00010580
04: .text:00010580 drvobj=dword ptr 8
05: .text:00010580
06: .text:00010580 push
07: .text:00010581 mov             ebp, esp
08: .text:00010583 push 09: .text:00010588
09: .text:00010584 mov             desplazamiento NombreDeEnlaceSimbólico ; NombreDeEnlaceSimbólico
10: .text:0001058E mov             ds:IoDeleteSymbolicLink
11: .text:00010591 mov             eax, [ebp+drvobj]
12: .text:00010595 mov             ecx, [eax+DRIVER_OBJECT ObjetoDispositivo]
13: .text:0001059A push 13: .text:00010595
14: .text:0001059B pop             ds: IoDeleteDevice
15: .text:0001059C retn
16: .text:0001059C DRV_Fin de descarga
```

Lo anterior se puede descompilar de la siguiente manera:

```
01: VACÍO DRV_Unload(PDRIVER_OBJECT drvobj)
02: {
03: IoDeleteSymbolicLink(&nombre de sym);
04: IoDeleteDevice(drvobj->ObjetoDispositivo);
05: }
```

Como se dijo anteriormente, una clave importante para comprender la funcionalidad de un controlador es a través de sus controladores de despacho IRP. Al analizar _IRP_ReadCloseDeviceO, comenzamos por el principio:

```
01: .text:00010300 ; NTSTATUS __stdcall IRP_ReadCloseDeviceO(
    PDEVICE_OBJECT devobj, PIRP Irp)
02: .text:00010300 Proceso IRP_ReadCloseDeviceO cerca
03: .text:00010300 var_14= dword ptr -14h
04: .text:00010300 var_10= dword ptr -10h
05: .text:00010300 var_C= dword ptr -0Ch
06: .text:00010300 var_8= dword ptr -8
07: .text:00010300 var_4= dword ptr -4
08: .text:00010300 devobj=dword ptr 8
09: .text:00010300 Irp= dword ptr 0Ch
10: .text:00010300
11: .text:00010300 push    ebp
12: .text:00010301 mov     ebp, esp
13: .text:00010303 subtítulo
14: .text:00010306 mov     [ebp+var_4], 0
15: .text:0001030D mov     eax, [ebp+Irp]
16: .text:00010310 mov     ecx, [ebp+var_4]
17: .text:00010313 mov     [eax+18h], ecx
18: .text:00010316 mov     edx, [ebp+Irp]
19: .text:00010319 mov     palabra clave d [edx+1Ch], 0
20: .text:00010320 mov     eax, [ebp+Irp]
21: .text:00010323 mov     ecx, [eax+60h]
22: .text:00010326 mov     [ebp+var_10], ecx
23: .text:00010329 mov     edx, [ebp+var_10]
24: .text:0001032C movzx eax, byte ptr [edx]
25: .text:0001032F cmp     Ea, 0Eh
26: .text:00010332 jnz    loc_1037D corto
```

Ya conocemos su prototipo porque es el mismo para todos los manejadores IRP. Al analizar los manejadores de IRP, es necesario tener en cuenta algunos hechos:

- Un IRP es una estructura dinámica con una matriz de IO_STACK_LOCATION después su encabezado.
- La mayoría de los parámetros IRP están en IO_STACK_LOCATION (incluido su IRP número mayor/menor).
- Un controlador accede a su IO_STACK_LOCATION utilizando la rutina `IrpStacLocation`. Debido a que esta rutina es en línea forzada, debe

Reconózcalo a través de sus patrones en línea. Es un patrón de codificación común para recuperar IO_STACK_LOCATION al comienzo de un controlador IRP.

Las líneas 15 a 17 leen la estructura IRP y escriben un 0 en un campo en el desplazamiento 0x18.

Mirando la estructura del IRP se ve lo siguiente:

```
0: kd> dt nt!_IRP
+0x000 Tipo :Int2B
+0x002 Tamaño :Uint2B
...
+0x00c Irp asociado : <etiqueta sin nombre>
...
+0x018 Estado de lo : _BLOQUE_DE_ESTADO_DE_IO
    +0x000 Estado :Int4B
    Puntero +0x000 : Ptr32 Vacío
    +0x004 Información :Uint4B
...
+0x020 Modo Solicitante :Carácter
...
+0x040 Cola : <etiqueta sin nombre>
```

Una estructura IO_STATUS_BLOCK almacena información de estado sobre un IRP:

```
tipo de estructura _IO_STATUS_BLOCK {
    unión {
        NTSTATUS Estado;
        Puntero PVOID;
    };
    ULONG_PTR Información;
} BLOQUE_DE_ESTADO_DE_IO, *BLOQUE_DE_ESTADO_DE_PIO;
```

Un controlador de IRP normalmente configura el campo Estado para indicar si el IRP se realizó correctamente o requiere un procesamiento adicional. La información almacena información específica de la solicitud para el IRP; un controlador puede usarla para almacenar un puntero a un búfer o establecer el estado de finalización. El puntero está reservado.

Por lo tanto, sabes que la línea 17 establece el campo IRP->IoStatus.Status en 0 y que la variable local var_4 es del tipo NTSTATUS. Las líneas 18 y 19 acceden a la estructura IRP y escriben un 0 en el desplazamiento 0x1c, que es el campo Información en IoStatus.

Esto consiste simplemente en configurar IRP->IoStatus.Information en 0. Las líneas 20 a 22 acceden al desplazamiento 0x60 en la estructura IRP y guardan su dirección en una variable local. La estructura IRP se llena con uniones en el campo Tail (comenzando en el desplazamiento 0x40), por lo que puede resultar algo confuso determinar a qué miembro del campo de unión se accede. Vamos a deshacernos de algunos sindicatos:

```
0: kd> dt nt!_IRP Superposición de cola.
+0x040 Cola : ...
Superposición +0x000 : ...
+0x000 Entrada de cola de dispositivo: _KDEVICE_QUEUE_ENTRY
+0x000 DriverContext: [4] Pir32 Vacío
```

```
+0x010 Hilo : Ptr32 _HILO
+0x014 Buffer auxiliar: carácter Ptr32
+0x018 Lista de entradas :_ENTRADA_DE_LISTA
+0x020 Ubicación actual de la pila: Ptr32 _IO_STACK_LOCATION
+0x020 Tipo de paquete :Uint4B
+0x024 ObjetoArchivoOriginal: Ptr32 _FILE_OBJECT
```

Esto indica que el desplazamiento 0x60 podría ser un puntero a un IO_STACK_UBICACIÓN o un entero sin signo que indica el tipo de paquete. Podemos hacer una suposición fundamentada de que se trata del campo CurrentStackLocation debido al contexto del código (que aparece al principio de un controlador IRP). Además, sabemos que la rutina en línea IoGetCurrentIrpStackLocation se define de la siguiente manera:

```
FUERZA EN LÍNEA
UBICACIÓN DE LA PILA PIO
IoGetCurrentIrpStackLocation(Irp PIRP) (Ubicación de pila de Irp actual)
{
    devuelve Irp->Tail.Overlay.CurrentStackLocation;
}
```

Por lo tanto, las líneas 20 a 22 guardan la IO_STACK_LOCATION actual en una ubicación local variable. La variable local _var_10 es del tipo PIO_STACK_LOCATION.

NOTA Muchas de estas funciones se declaran como FORCEINLINE y, por lo tanto, nunca aparecerán como destinos de llamadas; es decir, nunca verá el símbolo IoGetCurrentIrpStackLocation en el código ensamblador. Le recomendamos que escriba un controlador simple utilizando estas rutinas de inserción en línea forzada para que pueda acostumbrarse al patrón del código.

Las líneas 23 a 25 acceden al primer byte en el desplazamiento 0 en IO_STACK_LOCATION mediante la instrucción MOVZX . Esto indica que el campo es de tipo unsigned char. De la sección IRP, sabemos que este es el campo MajorFunction . La línea 5 verifica si el número de MajorFunction es 0xe, es decir, IRP_MJ_DEVICE_CONTROL.

Ahora puede descompilar el primer bloque de IRP_ReadCloseIo de la siguiente manera:

```
NTSTATUS IRP_ReadCloseIo(PDEVICE_OBJECT objeto de descodificación, PIRP Irp)
{
    NTSTATUS estado = ESTADO_EXITO;
    UBICACIÓN DE LA PILA PIO isl;
    Irp->IoStatus.Status = estado;
    Irp->IoStatus.Información = 0;

    isl = IoGetCurrentIrpStackLocation(Irp);
    si (isl->FunciónPrincipal != IRP_MJ_DEVICE_CONTROL) {
        ... // loc_1037D
    }
    ... // .texto:00010334
}
```

A continuación, analizamos el bloque 0x10334, que se ejecuta si el código principal es IRP_MJ CONTROL DEL DISPOSITIVO:

```

01:.texto:00010334mov          ecx, [ebp+var_10]
02:.texto:00010337mov          edx, [ecx+0Ch]
03:.texto:0001033A movimiento    [ebp+var_C], edición
04:.texto:0001033D mov          eax, [ebp+IrP]
05:.texto:00010340 mov          ecx, [eax+0Ch]
06:.texto:00010343 mov          [ebp+var_8], ecx
07:.texto:00010346 mov          edx, [ebp+IrP]
08:.texto:00010349 mov          palabra clave d [edx+1Ch], 644h
09:.texto:00010350 mov          eax, [ebp+var_C]
10:.texto:00010353 mov          [ebp+var_14], eax
11:.texto:00010356 cmp          [ebp+var_14], 22C004h
12:.texto:0001035D jz           loc_10361 corto

```

En el párrafo anterior, deducimos que var_10 es de tipo PIO_STACK_UBICACIÓN. Las líneas 1 y 2 acceden al desplazamiento 0xC de IO_STACK_LOCATION. Nuevamente, recuerde que IO_STACK_LOCATION contiene los parámetros de solicitud de E/S, que se almacenan en uniones. ¿Cómo determina qué unión utilizar? Sabemos que utilizará el campo DeviceloControl porque estamos procesando un IRP_MJ_Solicitud DEVICE_CONTROL. Además, el IoControlField está en el desplazamiento 0xC desde la base de IO_STACK_LOCATION:

```

1: kd> dt nt!_IO_STACK_LOCATION Parámetros.
+0x004 Parámetros:
    +0x000 Crear : <etiqueta sin nombre>
    +0x000 CreatePipe : <etiqueta sin nombre>
    +0x000 CreateMailslot : <etiqueta sin nombre>
    +0x000 Leer : <etiqueta sin nombre>
    +0x000 Escribir : <etiqueta sin nombre>
    +0x000 QueryDirectory : <etiqueta sin nombre>
...
    +0x000 DeviceloControl : <etiqueta sin nombre>
...
1: kd> dt nt!_IO_STACK_LOCATION Parámetros.DeviceloControl.
+0x004 Parámetros:
    +0x000 Control de dispositivo Io : 
        +0x000 Longitud del búfer de salida : Uint4B
        +0x004 Longitud del búfer de entrada : Uint4B
        +0x008 Código de control de Io : Uint4B
        +0x00c Buffer de entrada tipo 3 : Ptr32 Vacío

```

Por lo tanto, las líneas 1 a 3 recuperan el campo IoControlCode y lo guardan en var_C, que ahora sabemos que es del tipo ULONG.

Las líneas 4 a 6 acceden al desplazamiento 0xC en un IRP y guardan el puntero en una variable local var_8. De la sección anterior, sabemos que en el desplazamiento 0xC se encuentra el AssociatedIrP unión:

```

1: kd> dt nt!_IRP Irp asociado.
+0x00c Irp asociado:

```

```
+0x000 MasterIrp +0x000 :Ptr32_IRP
IrpCount +0x000 Búfer :Int4B
del sistema: Ptr32 Vacío
```

¿Cuál de los tres campos debería utilizar? Dada la información actual, no puede saberlo. El contexto necesario para determinar el campo adecuado se encuentra en las líneas 9 a 12, que recuperan el código IOCTL guardado (var_C) y lo comparan con 0x22c004. Ya sabes que un código IOCTL codifica el tipo de dispositivo, el código de función, el acceso y el método de almacenamiento en búfer. Por lo tanto, después de decodificar 0x22c004, sabes lo siguiente:

- El tipo de dispositivo es FILE_DEVICE_UNKNOWN (0x22).
- El código IOCTL es 0x1.
- El acceso es (FILE_READ_DATA | FILE_WRITE_DATA).
- El método de almacenamiento en búfer es METHOD_BUFFERED.

Recordemos que estamos en un controlador IOCTL y que los controladores deben especificar un método de almacenamiento en búfer al definir el código IOCTL. Para la E/S almacenada en búfer, el SystemBuffer El campo apunta a un buffer de grupo no paginado que almacena la entrada del usuario. Ahora podemos decir que las líneas 4 a 6 acceden al campo SystemBuffer .

Las líneas 7 y 8 escriben 0x644 en el desplazamiento 0x1c dentro de un IRP, que es IRP->IoStatus. Campo de información . No está claro por qué el autor eligió este valor.

Dada esta información, sabes que el código de control debe haberse construido de esta manera:

```
#define IOCTL_1 CÓDIGO_CTL(ARCHIVO_DISPOSITIVO_DESCONOCIDO, 1, MÉTODO_ALMACENADO_EN_BÜFFER, \
ARCHIVO_LECTURA_DATOS | ARCHIVO_ESCRITURA_DATOS)
```

Como no hemos analizado ni comprendido completamente la operación IOCTL, le hemos dado el nombre genérico IOCTL_1 . Ahora, este bloque se puede descompilar de la siguiente manera:

```
PVOID entrada de usuario = Irp->AssociatedIrp.SystemBuffer;
Irp->IoStatus.Información = (ULONG_PTR) 0x644;
si (isl->Parámetros.DeviceControl.IoControlCode == IOCTL_1)
{
    ... // loc_10361
}
... // 0001035F
```

Para entender lo que hace IOCTL, necesitamos analizar loc_10361 y la función sub_103B0. Sin embargo, antes de hacerlo, terminemos primero los bloques cercanos (ya que son más simples):

```
// recuerda que var_4 es una variable local de estado (tipo NTSTATUS)
01: .texto:0001035F jmp corto loc_1036C
02: .texto:00010361 loc_10361: ecx, [ebp+var_8];
03: .texto:00010361 mov
04: .texto:00010364 push
```

```

05: .text:00010365 llamada          Controlador IOCTL_1
06: .texto:0001036A jmp corto loc_1037D
07: .text:0001036C loc_1036C; 08: .text:0001036C mov
                                         [ebp+var_4], 0C00000010h edx, [ebp+Irp]
09: .text:00010373 mov
10:.text:00010376 mov          palabra clave d [edx+1Ch], 0
11: .text:0001037D loc_1037D; 12: .text:0001037D cmp
13: .text:00010384 jz 14: .text:00010386 xor
                                         [ebp+var_4], 103h
loc_1039A corto
dl, dl                                ; Impulso de prioridad
15: .text:00010388 mov          ecx, [ebp+Irp]           ;Irp
16: .text:0001038B llamada          ds:IoCompleteRequest
17: .text:00010391 mov          eax, [ebp+Irp]
18: .text:00010394 mov          ecx, [ebp+var_4]
19: .text:00010397 mov          [eax+18h], ecx
20: .text:0001039A loc_1039A; 21: .text:0001039A mov
eax, [ebp+var_4]
22: .text:0001039D mov          esp, ebp
23: .text:0001039F pop 24: .text:000103A0 retn
                                         .retn
                                         8
25: .text:000103A0 IRP_ReadCloseDeviceIO finaliza

```

Si el código IOCTL no coincide, se introduce 0x1035F . Salta inmediatamente a la línea 7, que establece la variable de estado local en 0xC0000010, que es STATUS_INVALID_OPERATION; e Irp->IoStatus.Information en 0. A continuación, en la línea 11, comprueba si el estado local es 0x103 (STATUS_PENDING); este bloque es realmente redundante porque la variable de estado en esta función solo puede tener dos valores (STATUS_SUCCESS o STATUS_INVALID_OPERATION). Cuando un IRP está marcado con STATUS_PENDING, significa que la operación está incompleta y está esperando que otro controlador la complete. Esto ocurre a menudo en los controladores, por lo que es conveniente recordar la constante mágica 0x103. Si el estado es STATUS_PENDING, el controlador regresa inmediatamente con ese estado (líneas 13 y 20). De lo contrario, llama a IoCompleteRequest para marcar el IRP como completado y guarda el estado en IRP->IoStatus.Status (línea 19) y lo devuelve. En realidad, esto es un error porque un controlador debe configurar el campo IoStatusBlock antes de completar la solicitud; una vez que se completa un IRP, no se debe volver a tocar. Estos bloques se pueden descompilar de la siguiente manera:

```

estado = ESTADO_OPERACIÓN_INVÁLIDA;
Irp->IoStatus.Información = 0;
si (estado == ESTADO_PENDIENTE) {
    estado de retorno;
}
IoCompleteRequest(Irp, IO_NO_INCREMENTO);
Irp->IoStatus.Status = estado;
estado de retorno;

```

Volviendo a la rutina IOCTL_1_handler , note que solo llama a otras dos funciones: sub_10460 y sub_10550. sub_10550 es una rutina de hoja pequeña, por lo que la analizaremos primero:

```

01: .text:00010550 ; void __stdcall sub_10550(PMDL Mdl, PVOID Dirección base)
02: .text:00010550 sub_10550 proc cerca 03: .text:00010550 push
04: .text:00010551 mov
05: .text:00010553 mov
06: .text:00010556 push
07: .text:00010557 mov
08: .text:0001055A empuje
09: .text:0001055B llamada
10: .text:00010561 mov
11: .text:00010564 push 12: .text:00010565
llamada
13: .text:00010568 mov
14: .text:0001056E empujar
15: .text:0001056F llamada
16: .text:00010575 emergente
17: .text:00010576 retn
18: .text:00010576 sub_10550 final

```

Esta función anula la asignación, desbloquea y libera un MDL. No está claro qué describen los MDL porque no hemos analizado las otras rutinas. Esta función se puede descompilar de la siguiente manera:

```

void UnmapMdl(PMDL mdl, PVOID dirección base)
{
    MmUnmapLockedPages(dirección base, mdl);
    MmUnlockPages(mdl);
    IoFreeMdl(mdl);
}

```

sub_10460 es otra rutina leaf que involucra MDL; su principal función es crear, bloquear y mapear un MDL para un buffer y una longitud determinados. Su prototipo es el siguiente:

```
PVOID MapMdl(PMDL *mdl, PVOID VirtualAddress, Longitud ULONG);
```

De forma predeterminada, el desensamblador no pudo inferir el tipo del primer parámetro. Se puede saber que es un PMDL* por la instrucción en 0x1049D. El listado del ensamblado se muestra aquí, pero sin comentarios línea por línea, ya que es muy simple:

```

01: .text:00010460 ; PVOID __stdcall MapMdl(PMDL *mdl, PVOID Dirección virtual,
ULONG Longitud)
02: .text:00010460 Procedimiento MapMdl cerca
03: .text:00010460 push
04: .text:00010461 mov
05: .text:00010463 push
06: .text:00010465 push
07: .text:0001046A empuje

```

08: .texto:0001046F movimiento	eax, fs grande:0
09: .texto:00010475 push 10: .texto:00010476 mov	fácil
	fs grande: 0, esp
11: .texto:0001047D agregar	esp, 0FFFFFFF0h
12: .texto:00010480 push 13: .texto:00010481 push	ebx
14: .texto:00010482 push 15: .texto:00010483 mov	esi
	editar
	[ebp+var_18], esp
16: .texto:00010486 push 17: .texto:00010488 push	0
18: .texto:0001048A push 19: .texto:0001048C mov	0 ; Irp
	0 ; Cuota de carga
	0 ; Buffer secundario
	eax, [ebp+Longitud]
20: .texto:0001048E empujar 21: texto:00010490 mover	fácil ; Longitud
22: .text:00010493 push 23: .text:00010494 llamada	ecx, [ebp+Dirección Virtual] ; Dirección virtual
	ds:IoAllocateMdl
24: .texto:0001049A movimiento	edx, [ebp+mdl]
25: .texto:0001049D mov	[edx], fácil
26: .texto:0001049F mov	eax, [ebp+mdl]
27: .texto:000104A2 cmp 28: .texto:000104A5 jnz	palabra clave d [eax], 0
29: .texto:000104A7 xor	loc_104AE corto
	Ea, Ea
30: .texto:000104A9 jmp loc_10534	
31: .texto:000104AE loc_104AE: 32: .texto:000104AE mov	[ebp+var_4], 0
	1 ; Operación
33: .texto:000104B5 push 34: .texto:000104B7 push	0 ; Modo de acceso
35: .text:000104B9 mov	ecx, [ebp+mdl]
36: .text:000104BC mov	edx, [ecx]
37: .text:000104BE push 38: .text:000104BF llamada	ds:IoAllocateVirtualMemory ; Lista de descriptores de memoria
39: .text:000104C5 mov	[ebp+var_4], 0FFFFFFFh
40: .text:000104CC jmp corto loc_104F6	
41: .text:000104CE loc_104CE: 42: .text:000104CE mov	Ej., 1
43: .text:000104D3 retn	
44: .text:000104D4 loc_104D4: 45: .text:000104D4 mov esp, [ebp+var_18]	
46: .text:000104D7 mov	eax, [ebp+mdl]
47: .text:000104DA movimiento	ecx, [eax]
48: .text:000104DC push 49: .text:000104DD llamada	ds:IoFreeMdl ; Mdl
50: .text:000104E3 mov	[ebp+var_20], 0
51: .text:000104EA mov	[ebp+var_4], 0FFFFFFFh
52: .text:000104F1 mov	eax, [ebp+var_20]
53: .text:000104F4 jmp corto loc_10534	
54: .text:000104F6 loc_104F6: 55: .text:000104F6 push push	10 horas ; Prioridad
	0 ; Comprobación de errores en caso de error
	0 ; Dirección base

168 Capítulo 3 ■ El núcleo de Windows

```

58: .texto:000104FC push 59: .texto:000104FE push 0 ; Tipo de caché
60: .texto:00010500 mov 0 ;Modo de acceso
                           edx, [ebp+mdl]
61: .texto:00010503 mov Eax, [edx] fácil ; Lista de descriptores de memoria
62: .texto:00010505 push 63: .text:00010506 llamada ds:MmMapLockedPagesSpecifyCache
64: .text:0001050C movimiento [ebp+var_1C], eax
65: .text:0001050F cmp 66: .text:00010513 jnz [ebp+var_1C], 0
67: .text:00010515 mov loc_10531 corto
                           ecx, [ebp+mdl]
68: .text:00010518 mov edx, [ecx] educación ; Lista de descriptores de memoria
69: .text:0001051A push 70: .text:0001051B llamada ds:MmUnlockPages
70: .text:00010521 mov eax, [ebp+mdl]
72: .text:00010524 mov ecx, [eax]
73: .text:00010526 push 74: .text:00010527 llamada ds:IoFreeMdl ; Mdl
75: .text:0001052D xor Ea, Ea
76: .text:0001052F jmp corto loc_10534
77: .text:00010531 loc_10531: 78: .text:00010531 mov eax, [ebp+var_1C]
79: .text:00010534 loc_10534: 80: .text:00010534 mov ecx, [ebp+var_10]
81: .text:00010537 mov gran fs:0, ecx
82: .text:0001053E pop 83: .text:0001053F pop editar
84: .text:00010540 pop 85: .text:00010541 mov esi
                           ebx
                           esp, ebp
86: .text:00010543 pop 87: .text:00010544 retn 0Ch
88: .text:00010544 Fin del MapMdl

```

Aunque esta función parece larga y complicada, no es difícil de entender si se observa cómo se utilizan las API en conjunto. IoAllocateMdl, MmProbeAndLockPages y MmMapLockedPagesSpecifyCache son rutinas que se utilizan para crear, bloquear y mapear MDL; MmProbeAndLockPages debe realizarse dentro de un bloque try/except, por lo que se genera código adicional al principio para configurar el controlador de excepciones (es decir, las líneas que involucran fs:0). Esta rutina mapea efectivamente un búfer en el espacio del núcleo como escribible y devuelve la dirección de un nuevo mapeo para este búfer. La rutina completa se puede descompilar aproximadamente de la siguiente manera:

```

PVOID MapMdl(PMDL *mdl, dirección virtual PVOID, longitud ULONG)
{
    PVOID addr; // dirección virtual del MDL mapeado

    *mdl = IoAllocateMdl(DirecciónVirtual, Longitud, FALSO, FALSO, NULO);
    si (*mdl == NULL) devuelve NULL;
    __intental {
        MmProbeAndLockPages(*mdl, ModoNúcleo, IoWriteAccess);
        dirección = MmMapLockedPagesSpecifyCache(

```

```

        *mdl,
        Modo kernel,
        MmSin caché,
        NULO,
        FALSO,
        PrioridadPáginaNormal);

    si (dirección == NULL) {
        MmUnlockPages(*mdl);
        IoFreeMdl(*mdl);
    }
} __except (MANEJADOR_DE_EJECUCIÓN_DE_EXCEPCIONES) {
    IoFreeMdl(*mdl);
}
dirección de retorno;
}

```

Una vez que entendemos estas dos rutinas, podemos abordar el controlador. Observe que acepta un parámetro, `IrP->AssociatedIrp.SystemBuffer`. Recuerde que el contenido de este buffer se puede volver a copiar al modo de usuario una vez que se complete el IRP:

```

01: .text:000103B0 ; void __stdcall IOCTL_1_handler(búfer PVOID)
02: .text:000103B0 IOCTL_1_handler proc cerca
03: .text:000103B0 push 04: .text:000103B1 mov
    ;-----[ebp]-----
    ;-----[esp]-----
05: .text:000103B3 subtítulo
06: .text:000103B6 push 07: .text:000103B7
    llamada
07: .text:000103BD mov
08: .text:000103C0 mov
09: .text:000103C5 mov
10: .text:000103C8 shl
11: .text:000103CB empujar 13: .text:000103CC
    mover
12: .text:000103D2 mov
13: .text:000103D4 empujar 16: .text:000103D5
    leer
14: .text:000103D8 push 18: .text:000103D9
    llamada
15: .text:000103DE mov
16: .text:000103E1 cmp 21: .text:000103E5 jz
17: .text:000103E7 mov
18: .text:000103EE jmp corto loc_103F9
19: .text:000103F0 loc_103F0:
20: .text:000103F0 movimiento edx, [ebp+var_8]
21: .text:000103F3 agregar
22: .text:000103F6 mov
23: .text:000103F9 loc_103F9:
24: .text:000103F9 mov eax, [ebp+buffer]
    ;-----[ebp]-----
    ;-----[esp]-----
25: .text:000103F9 loc_103F9:
26: .text:000103F9 loc_103F9:
27: .text:000103F9 loc_103F9:
28: .text:000103F9 loc_103F9:
29: .text:000103F9 loc_103F9:
    ;-----[ebp]-----
    ;-----[esp]-----

```

170 Capítulo 3 ■ El núcleo de Windows

```

30: .text:000103FC mov          ecx, [ebp+var_8]
31: .text:000103FF cmp 32: .text:00010401 jnb      ecx, [eax]
33: .text:00010403 mov          loc_1043C corto
34: .text:00010406 mov          edx, [ebp+var_8]
35: .text:00010409 cmp 36: .text:0001040E jz       eax, [ebp+búfer]
37: .text:00010410 mov          palabra clave d [eax+edx*4+4], 0
loc_1043A corto
38: .text:00010413 mov          ecx, [ebp+var_8]
39: .text:00010416 mov          edx, [ebp+Dirección base]
40: .text:00010419 mov          eax, [ebp+var_8]
41: .text:0001041C mov          esi, [ebp+búfer]
42: .text:0001041F cmp 43: .text:00010423 jz       ecx, [edx+ecx*4]
44: .text:00010425 mov          loc_1043A corto
45: .text:00010428 mov          edx, [ebp+var_8]
46: .text:0001042B mov          eax, [ebp+búfer]
47: .text:0001042F movimiento    ecx, [eax+edx*4+4]
48: .text:00010432 mov          edx, [ebp+var_8]
49: .text:00010435 lea          eax, [ebp+Dirección base]
50: .text:00010438 xchg 51: .text:0001043A
loc_1043A:
52: .text:0001043A jmp corto loc_103F0
53: .text:0001043C loc_1043C:
54: .text:0001043C mov eax, [ebp+Dirección base]
55: .text:0001043F empujar 56: .text:00010440
mover           facil
                ; Dirección base
57: .text:00010443 push 58: .text:00010444
llamada          ecx, [ebp+Mdl]
                ; Mdl
                ; DesasignarMdl
59: .text:00010449 loc_10449:
60: .text:00010449 mov cl, [ebp+NewIrql] ds:KfLowerIrql
                ; Nuevo Irql
61: .text:0001044C llamada
62: .text:00010452 pop 63: .text:00010453 mov
                esi
                esp, ebp
64: .text:00010455 pop 65: .text:00010456 retn
                ; Retorno
                4
66: .text:00010456 Fin del controlador IOCTL_1

```

Esta función primero eleva el IRQL a DISPATCH_LEVEL (línea 7), lo que efectivamente suspende el despachador de subprocessos en el procesador actual. Cualquiera que sea lo que haga esta función, no puede esperar ni aceptar un error de página; de lo contrario, la máquina comprobará si hay errores. Se puede lograr el mismo efecto con KeRaiseIrql. La línea 8 guarda el IRQL anterior para que se pueda restaurar más tarde (consulte la línea 61). Las líneas 9 a 11 recuperan el campo de entrada KeServiceDescriptorTable no documentado y lo multiplican por 4. Las líneas 12 a 18 pasan KiServiceTable, una longitud (cuatro veces el tamaño de la tabla de llamadas al sistema) y un puntero MDL a MapMdl. Debido a que ya analizamos MapMdl, sabemos que esto simplemente asigna un búfer que comienza desde KiServiceTable a KiService. Tabla+(NumberOfSyscalls*4). La línea 12 guarda la dirección virtual del búfer recién mapeado. Las líneas 20 a 22 verifican el estado del mapeo; si no fue exitoso,

Se reduce el IRQL y el código retorna (líneas 60 a 65); de lo contrario, se ingresa a un bucle cuyo contador se determina según la entrada del usuario (líneas 29 a 31). El cuerpo del bucle es de las líneas 33 a 50 y se puede entender de la siguiente manera:

```
DWORD *userbuffer = Irp->AssociatedIrp.SystemBuffer;
DWORD *mappedKiServiceTable = MapMdl(mdl, KiServiceTable, nsysecls*4);
para (i=0; i < userbuffer[0] ; i++)
{
    si (buffer de usuario[i+1] != 0) {
        si (usuariobuffer[i+1] != mappedKiServiceTable[i]) {
            intercambiar(mappedKiServiceTable[i], userbuffer[i+1]);
        }
    }
}
...
DesasignarMdl(mdl);
KeLowerIrql(antiguoirql);
```

Después de muchas páginas de explicaciones y de descompilar todo el controlador, ahora puede comprender el objetivo del ejemplo. Por alguna razón, el desarrollador de este controlador quería utilizar una IOCTL para sobrescribir la tabla de llamadas del sistema nativo de NT con direcciones personalizadas. El búfer de modo de usuario es una estructura en este formato:

```
[# de llamadas al sistema]
[Dirección de reemplazo de syscall 1]
[Dirección de reemplazo de syscall 2]
...
[syscall n dirección de reemplazo]
```

Si bien el desarrollador puede haber logrado sus objetivos, el controlador tiene varios problemas críticos que pueden provocar inestabilidad del sistema y vulnerabilidades de seguridad. Algunos de ellos se mencionaron durante el tutorial, pero deberías poder identificar muchos otros. Aquí tienes algunas preguntas para comenzar tu búsqueda:

- ¿Funcionará este controlador en un sistema multinúcleo? Explique su razonamiento.
- ¿Por qué cree el autor que es necesario elevar el IRQL a DISPATCH_LEVEL?
 - ¿Es realmente necesario?
- ¿Cómo puede un usuario normal usar este controlador para ejecutar código arbitrario en el anillo? 0 contexto?
- Supongamos que el autor quisiera reemplazar algunas llamadas del sistema con una implementación personalizada en el espacio de usuario. ¿Qué problemas podrían surgir?

Este controlador es muy pequeño y simple, pero tiene la mayoría de las construcciones importantes que se encuentran típicamente en los controladores de software: rutinas de despacho, control de E/S de dispositivos desde el modo de usuario, métodos de almacenamiento en búfer, enlaces simbólicos, aumento y disminución de niveles de IRQL, administración de MDL, IO_STACK_LOCATION, etc. Puede aplicar las mismas técnicas analíticas que se muestran aquí a otros controladores. Simplemente no imite sus técnicas de desarrollo en la vida real.

Un rootkit x64

En esta sección se analiza el ejemplo B, un controlador x64. Debido a que es bastante grande y complejo , nos centraremos únicamente en las áreas relacionadas con las devoluciones de llamadas. No pegaremos cada línea de esta función, por lo que deberá seguirla en un desensamblador.

Tenga en cuenta que este controlador especifica la creación de procesos y las notificaciones de carga de imágenes mediante las API documentadas. 0x4045F8 es el inicio de la rutina de devolución de llamada de creación de procesos. Primero, borra una estructura LARGE_INTEGER a cero. Una estructura LARGE_INTEGER La estructura se utiliza normalmente para representar el tamaño o el tiempo del archivo (tenga en cuenta que se utiliza más adelante en 0x4046FF como argumento para KeDelayExecutionThread). A continuación, obtiene el id del proceso actual con PsGetCurrentProcessId. ¿Esto obtiene el id del proceso recién creado? No necesariamente. El prototipo de devolución de llamada de creación de proceso es el siguiente:

```
VACÍO
(*PCREATE_RUTINA_DE_NOTIFICACIÓN_DE_PROCESO) (
    EN MANEJO Parentid,
    EN MANEJO ProcessId, // processId del proceso creado/terminado
    EN BOOLEAN Crear // VERDADERO=creación FALSO=terminación
);
```

El parámetro Creation se guarda y se prueba en 0x404604 y 0x404631, respectivamente ; si es TRUE, entonces la devolución de llamada simplemente retorna. Por lo tanto, sabemos que esta devolución de llamada solo rastrea la terminación del proceso. En el caso de la terminación del proceso, la devolución de llamada se ejecuta en el contexto del proceso que está muriendo. Después de recopilar el id del proceso que termina (que no se usa en absoluto), recupera el objeto EPROCESS para el proceso actual a través de IoGetCurrentProcess (0x40461C y 0x404622). No está claro por qué se llama a IoGetCurrentProcess dos veces (podría ser un error tipográfico en el código fuente original). A continuación, recupera y guarda la cadena de nombre del archivo de imagen del proceso mediante PsGetProcessImageFileName (0x404633). Si bien esta rutina no está documentada, es simple, se exporta y el kernel la utiliza con frecuencia. Luego intenta adquirir un bloqueo de recursos previamente inicializado en DriverEntry (0x4025EB); ingresa a una región crítica antes de adquirir un bloqueo de recursos porque KeAcquireResourceExclusiveLite requiere que los APC del kernel normales estén deshabilitados (que es lo que hace KeEnterCriticalSection). A continuación, obtiene un puntero a una lista enlazada y verifica el nombre de la imagen del proceso de terminación con cada entrada en la lista (desplazamiento 0x20). Sabes que se trata de una lista enlazada porque el bucle itera por punteros (0x404679) y termina cuando dos punteros son iguales (0x40465F).

Si no hay ninguna coincidencia, libera el bloqueo de recursos y pausa el subprocesso actual (0x4046FF) un segundo después de la hora actual. Si el nombre de archivo del proceso que finaliza coincide con uno de los de la lista, desasigna, desbloquea y libera un MDL almacenado en la entrada de la lista (desplazamiento 0x1070). Si el búfer en el desplazamiento 0x10b0 en la entrada de la lista es NULL, se libera; de lo contrario, la entrada se libera de la lista mediante la macro RemoveEntryList :

```

01: .texto:000000000004046CA loc_4046CA:
02: .texto:000000000004046CA movimiento Rax, [rbx+8]
03: .texto:000000000004046CE movimiento r8, [rbx]
04: .texto:000000000004046D1 mov edx, edición ; Etiqueta
05: .texto:000000000004046D3 mov [rax], r8
06: .texto:000000000004046D6 mov Rx, Rx ; PAG
07: .texto:000000000004046D9 mov [r8+8], rax
08: .text:00000000004046DD llamada cs:ExFreePoolConEtiqueta

```

Nuevamente, podemos reconocer la operación de lista debido al patrón de manipulación Flink (desplazamiento 0x0) y Blink (desplazamiento 0x8). De hecho, ahora podemos decir que qword_40A590 es de tipo LIST_ENTRY.

Aunque esta devolución de llamada es solo una pieza del rompecabezas, puedes aplicar los hechos anteriores para comprender indirectamente otros componentes del rootkit. Por ejemplo, puedes saber que el rootkit asigna o inyecta código en los procesos y los rastrea en una gran lista enlazada (usando el nombre del proceso como clave). Cuando el proceso muere, tienen que desasignar esos MDL porque el sistema comprobará si un proceso muerto todavía tiene páginas bloqueadas. Las asignaciones de MDL originalmente probablemente se realizaron a través de la rutina de devolución de llamada de carga de imagen (0x406494).

Otra rutina interesante en este archivo es 0x4038F0. Realizaremos un análisis línea por línea de esta rutina porque utiliza estructuras que verás con frecuencia en otros controladores. Además, enseña algunas lecciones valiosas sobre el análisis de código x64 optimizado:

```

01: ; NTSTATUS __cdecl sub_4038F0(PFILE_OBJECT ObjetoArchivo, l
                                         HANDLE (Manejador, indicador BOOLEAN)

02: sub_4038F0 proc cerca
03:    empajar 04:      Rbx
04:    empajar 05:      PBI
05:    empajar 06:      rsi
06:    empajar 07:      RDI-dl
07:    empajar 08:      sub   r12
08:                  rsp, 60 horas
09:    movimiento      bpl, r8b
10:    movimiento      R12, rdx
11:    movimiento      RDI, RCX
12:    llamada        cs:IoGetRelatedDeviceObject
13:    movimiento      [rsp+88h+arg_18], 1
14:    xor             educación, educación ; Cuota de carga
15:    movimiento      cl, [rax+4Ch] ; Tamaño de pila
16:    movimiento      RSI, RAX
17:    llamada        cs:IoAllocateIrp
18:    prueba         Rax, Rax
19:    movimiento      Rbx, rax
20:    jnz            loc_403932 corto
21:    movimiento      Ej., 0C0000017h
22:    jmp loc_403A0C
23: loc_403932: 24: lea rax,
24:               [rsp+88h+arg_18]
25:    xor             r8d, r8d          ; Estado

```

174 Capítulo 3 ■ El núcleo de Windows

26: lea	rcx, [rsp+88h+Evento] ; Evento
27: movimiento	[rbx+18h], rax ; IRP Búfer de sistema asociado a Irp
28: lea	rax, [rsp+88h+Evento]
29: lea	edx, [r8+1] ; Tipo
30: movimiento	[rbx+50h], rax ; IRP_EventoUsuario
31: lea	rax, [rsp+88h+var_58]
32: movimiento	[rbx+48h], rax ; IRP_UsuarioIosb
33: movimiento	rax, gs<188h ; KPCR_Prcb_Hilo actual
34: movimiento	[rbx+0C0h], rdi ; IRP_Tail_Overlay_ObjetoArchivoOriginal
35: movimiento	[rbx+98h], rax ; IRP_Cola_Superposición_Hilo
36: movimiento	byte ptr [rbx+40h], 0; IRP_ModoSolicitante
37: llamada	cs:KeInitializeEvent
38: prueba	bpl, bpl
39: movimiento	rcx, [rbx+0B8h]
40: movimiento	byte ptr [rcx-48h], 6; IRP_MJ_SET_INFORMACION
41: movimiento	[rcx-20h], rsi; UBICACIÓN DE LA PILA DE E/S.Objeto del dispositivo
42: movimiento	[rcx-18h], rdi ; UBICACIÓN DE LA PILA DE E/S.FileObject
43: yo	loc_4039A6 corto
44: movimiento	rax, [rdi+28h] ; OBJETO_ARCHIVO PointerObjeto_Sección
45: prueba	Rax, Rax
46:jz	loc_4039A6 corto
47: movimiento	[rax+10h], 0 ; PUNTERO_DE_OBJETO_DE_SECCIÓN.Objeto_de_sección_de_imagen
48: loc_4039A6:	
49: movimiento	[rcx-28h], r12 ; IO_STACK_LOCATION Parámetros SetFile.DeleteHandle
50: movimiento	[rcx-30h], rdi ; IO_STACK_LOCATION Parámetros SetFile.FileObject
51: movimiento	dword ptr [rcx-38h], 0Dh ; Información de disposición del archivo ; IO_STACK_LOCATION Parámetros SetFile.FileInformationClass
52: movimiento	palabra clave d [rcx-40h], 1 ; IO_STACK_LOCATION Parámetros EstablecerArchivo Longitud
53: movimiento	rax, [rbx+0B8h] ; Ubicación actual de la pila de Irp
54: lea	rcx, sub_4038B4; rutina de finalización
55: movimiento	[rax-10h], rcx; UBICACIÓN_DE_PILA_DE_IO.Rutina_de_completado
56: movimiento	Rcx, Rsi ;Objeto del dispositivo
57: movimiento	rdx, rbx ;irp
58: movimiento	qword ptr [rax-8], 0 byte ptr [rax-46h],
59: movimiento	0E0h ; bandera
60: llamada	cs:IoCallDriver
61: cmp	eax, 103h ; ESTADO_PENDIENTE
62: jnz	loc_403A09 corto
63: lea	rcx, [rsp+88h+Evento] ; Objeto
64: movimiento	r8b, 1 ; Alertable
65: xor	r8d, r8d ;Modo de espera
66: xor	edx, edx ; Motivo de espera
67: movimiento	[rsp+88h+var_58], 0
68: llamada	cs:KeWaitForSingleObject
69: loc_403A09: 70: movimiento	
	eax, [rbx+30h] ; IRP_IoSStatus Estado
71: loc_403A0C: 72: agregar	
rsp, 60h	
73: estallido	r12
74: estallido	RDI-di

```

75: estallido      rsi
76: estallido      PBI
77: estallido      Rbx
78: retornado
79: sub_4038F0 fin

```

Primero, recuperamos el prototipo de la función notando que el llamador de la función usa tres registros: RCX, RDX, R8 (vea 0x404AC8 a 0x404ADB). Aunque el desensamblador marca CDECL como la convención de llamada de la función, no es realmente correcto. Recuerde que Windows en la plataforma x64 solo usa una convención de llamada que especifica que los primeros cuatro argumentos se pasan a través de registros (RCX, RDX, R8 y R9) y el resto se coloca en la pila. La línea 12 llama a IoGetRelatedDeviceObject usando FileObject como parámetro; esta API devuelve el objeto de dispositivo asociado con el objeto de archivo. El objeto de dispositivo asociado se guarda en RSI. Las líneas 14 a 17 asignan un IRP desde cero con IoAllocateIrp; el campo StackSize del objeto de dispositivo se usa como el tamaño IO_STACK_LOCATION. Si la asignación de IRP falla de alguna manera, la rutina devuelve STATUS_NO_MEMORY (líneas 20 a 22). De lo contrario, el nuevo IRP se guarda en RBX (línea 19) y continuamos con la línea 24. Las líneas 24 a 37 inicializan los campos básicos de un IRP y llaman a KeInitializeEvent.

La línea 33 puede parecer extraña debido al parámetro GS:188h . Recuerde que en Windows x64 , el núcleo almacena un puntero al PCR en GS, que contiene el PRCB que almacena la información de programación. De hecho, esta rutina es simplemente la forma en línea de KeGetCurrentThread. La línea 39 accede a un campo en el desplazamiento 0xb8 en la estructura IRP. ¿Qué es este campo?

```

0: kd> dt ntl!_IRP Superposición de cola.
+0x078 Cola
Superposición +0x000
+0x000 Entrada de cola de dispositivo: _KDEVICE_QUEUE_ENTRY
+0x000 DriverContext: [4] Ptr64 Vacío
+0x020 Hilo : Ptr64 _HILO
+0x028 Buffer auxiliar: Ptr64 Char
+0x030 Lista de entradas : _ENTRADA_DE_LISTA
+0x040 Ubicación actual de la pila: Ptr64 _IO_STACK_LOCATION
+0x040 Tipo de paquete : Uint4B
+0x048 ObjetoArchivoOriginal: Ptr64 _FILE_OBJECT

```

Está accediendo al puntero CurrentStackLocation en la unión Overlay . ¿Le suena familiar? La línea 39 en realidad es simplemente IoGetCurrentIrpStackLocation. Las líneas 40 a 42 establecen algunos campos utilizando desplazamientos negativos desde la ubicación actual de la pila. Recuerde que la parte dinámica de un IRP es una matriz de IO_STACK_LOCATION estructuras y la ubicación de la pila “próxima” es en realidad el elemento que se encuentra sobre el actual . Revise esta estructura y su tamaño:

```

0: kd> tamaño de (_UBICACIÓN_DE_PILA_IO)
entero sin signo 64 0x48
0: kd> dt _UBICACIÓN_DE_PILA_DE_IO
nt!_UBICACIÓN_DE_PILA_IO

```

+0x000 Función principal +0x001	:UCar
Función secundaria	:UCar
+0x002 Banderas	:UCar
+0x003 Control	:UCar
+0x008 Parámetros	: <etiqueta sin nombre>
+0x028 Objeto de dispositivo	: Ptr64 _OBJETO_DISPOSITIVO
+0x030 Objeto de archivo	: Ptr64 _OBJETO_ARCHIVO
+0x038 Rutina de finalización: Ptr64 largo	
+0x040 Context	:Ptr64 Vacío

El tamaño de un IRP en Windows x64 es 0x48. Por lo tanto, la línea 40 debe estar accediendo a la “próxima” IO_STACK_LOCATION porque está restando 0x48 bytes de la ubicación actual; está configurando el campo MajorFunction en 0x6 (IRP_MJ_SET_INFORMATION). Esto le indica que los parámetros para esta solicitud se describirán utilizando el miembro de unión SetFile . La línea 41 accede al IRP “próximo” con desplazamientos negativos 0x20 y 0x18, que corresponde al DeviceObject y FileObject , respectivamente. Lo que sucede aquí es que el desarrollador utilizó IoGetNextIrpStackLocation y luego completó el campo, y el agresivo compilador x64 de Microsoft optimizó el código de esa manera. El optimizador decidió que debido a que estamos operando en una matriz de estructuras, es más barato (en términos de espacio) acceder directamente al elemento anterior utilizando desplazamientos negativos; la alternativa habría sido calcular un nuevo puntero base para el elemento anterior y acceder a sus campos utilizando desplazamientos positivos. Te encontrarás con esta optimización con bastante frecuencia en binarios x64.

La línea 43 prueba un indicador para determinar si se deben realizar comprobaciones adicionales para los objetos de sección. Las líneas 44 a 47 establecen el campo ImageSectionObject en consecuencia. Las líneas 48 a 52 inicializan varios campos en la “próxima” ubicación de la pila IRP utilizando nuevamente desplazamientos negativos. Estos desplazamientos están dentro de la unión de parámetros ; como ya conocemos la función principal de IRP (IRP_MJ_SET_INFORMATION), sabemos que Utilice el miembro de unión SetFile :

1: kd> dt ntl!IO_STACK_LOCATION Parámetros.SetFile.	
+0x008 Parámetros	:
+0x000 Establecer archivo	:
+0x000 Longitud +0x008	:Uint4B
Clase de información de archivo: _FILE_INFORMATION_CLASS	
+0x010 FileObject +0x018	: Ptr64 _OBJETO_ARCHIVO
Reemplazar si existe +0x019 Solo avance	:UCar
+0x018 Conteo de clústeres	:UCar
+0x01C Longitud	:Uint4B
+0x018 Eliminar identificador	:Ptr64 Vacío

Después de calcular los desplazamientos, sabemos que la línea 49 establece el campo DeleteHandle con el segundo parámetro, la línea 50 establece el campo FileObject , la línea 51 establece el campo FileInformationClass (0xD es FileDispositionInformation) y la línea 52 establece el campo Length . La documentación de FileDispositionInformation La clase dice que tomará una estructura con un campo de un byte; si es 1, entonces el archivo

El identificador está marcado para su eliminación. Por lo tanto, ahora sabemos por qué las líneas 13 y 27 establecen el IRP.AssociatedIrp.SystemBuffer en 1. Las líneas 53 a 55 establecen sub_4038B4 como la rutina de finalización de este IRP. La línea 60 pasa el IRP recién completado a otro controlador (tomado de la línea 16) para su procesamiento (muy probablemente el controlador del sistema de archivos). La línea 61 verifica el estado con STATUS_PENDING para ver si la operación se realizó; si es así, el estado del IRP se devuelve en EAX; si no, se llama a KeWaitForSingleObject para esperar el evento inicializado en la línea 37. La rutina de finalización establecerá el evento y liberará el IRP cuando haya terminado:

```

01: sub_4038B4 proc cerca
02: empujar      Rbx
03: sub          rsp, 20h
04: movdqu xmm0, xmmword ptr [rdx+30h]
05: movimiento   Rax, [rdx+48h]
06: movimiento   Rbx, rdx
07: xor          r8d, r8d           ; Esperar
08: xor          educación, educación ; Incremento
09: movdqu xmmword ptr [rcx], xmm0
10: movimiento   rcx, [rbx+50h] ; Event
11: llamada      cs:KeSetEvent
12: movimiento   Rx, Rx           ;:lrp
13: llamada      cs:IoFreeIrp
14: movimiento   Ej., 0C0000016h
15: añadir       rsp, 20h
16: estallido    Rbx
17: retornado
18: sub_4038B4 fin

```

La rutina completa se puede descompilar de la siguiente manera:

```

NTSTATUS sub_4038F0(PFILE_OBJECT FileObj, HANDLE hdelete, indicador BOOLEAN)
{
    Estado NTSTATUS;
    UBICACIÓN DE LA PILA PIO ios;
    PIRP Irp;
    PDEVICE_OBJECT devobj;
    Evento KEVENT;
    BLOQUE DE ESTADO DE E/S iosb;
    CHARbuf = 1;

    devobj = IoGetRelatedDeviceObject(ObjetoArchivo);
    Irp = IoAllocateIrp(devobj->StackSize, FALSO);
    si (Irp == NULL) { devolver ESTADO_SIN_MEMORIA; }
    Irp->AssociatedIrp.SystemBuffer = &buf;
    Irp->UserEvent = &event;
    Irp->Useriosb = &iosb;
    Irp->Tail.Overlay.Thread = KeGetCurrentThread();
    Irp->Tail.Overlay.OriginalFileObject = FileObj;
    Irp->ModoSolicitante = ModoNúcleo;
    KeInitializeEvent(&evento, Evento de sincronización, FALSO);
}

```

```
iosl = IoGetNextIrpStackLocation(Irp);
iosl->DeviceObject = devobj;
iosl->ObjetoArchivo = ObjetoArchivo;
si (!bandera && FileObj->SectionObjectPointer != NULL) {
    ArchivoObj->SectionObjectPointer.ImageSectionObject = NULL;
}
iosl->Parámetros.SetFile.FileObject = FileObj;
iosl->Parámetros.SetFile.DeleteHandle = hdelete;
iosl->Parámetros.SetFile.FileInformationClass = \
    Información de disposición del archivo;
iosl->Parámetros.SetFile.Length = 1;
Ubicación de pila de Irp actual de Irp
IoSetCompletionRoutine(Irp, sub_4038B4, NULL, VERDADERO, VERDADERO, VERDADERO);
si (IoCallDriver(devobj, Irp) == ESTADO_PENDIENTE) {
    KeWaitForSingleObject(&evento, Ejecutivo, KernelMode, VERDADERO, NULL);
}
devuelve Irp->IoStatus.Status;
}
```

Ahora puede ver que el controlador utiliza esta función para eliminar un archivo del sistema sin utilizar la API de eliminación de archivos (ZwDeleteFile). Lo logra creando su propio IRP para describir la operación de eliminación de archivos y lo pasa a un controlador inferior (presumiblemente el sistema de archivos). Además, utiliza una rutina de finalización para recibir una notificación cuando se completa el IRP (ya sea exitoso, fallido o cancelado de alguna manera). Si bien es un poco esotérico, este método es muy útil porque puede eludir el software de seguridad que intenta detectar la eliminación de archivos a través del enganche de llamadas del sistema.

Este tutorial demostró dos puntos principales. En primer lugar, si conoce y comprende los objetos y mecanismos que utilizan los controladores para interactuar con el núcleo, su tarea analítica se vuelve más sencilla. En segundo lugar, debe estar preparado para lidiar con código que parece extraño debido a un optimizador agresivo. Esto es especialmente cierto para el código x64. La única forma de mejorar es practicando.

Próximos pasos

Hemos cubierto la mayoría de los conceptos importantes específicos del dominio que son relevantes para el código en modo kernel en Windows. Este conocimiento se puede aplicar inmediatamente a las tareas de ingeniería inversa de controladores. Sin embargo, para ser más efectivos, es instructivo entender cómo se ven los controladores normales en formato fuente. La mejor manera de aprenderlo es estudiar los ejemplos de controladores incluidos en el WDK o desarrollar sus propios controladores. Si bien no son rootkits, demuestran la estructura y las construcciones adecuadas que utilizan los controladores.

¿Qué hacer a partir de ahora? Nuestro consejo es el siguiente (en orden):

- Lea el manual de WDK detenidamente. Puede comenzar con la sección “Arquitectura del controlador en modo kernel”. Al principio puede resultar confuso, pero si lee este capítulo le resultará mucho más fácil, ya que hemos pasado por alto todos los temas no esenciales.
- Lea Desarrollo de controladores de dispositivos de Windows NT de Peter G. Viscarola y W. Anthony Mason de principio a fin (puede omitir el capítulo sobre DMA y E/S programada).
- Escriba algunos controladores pequeños y simples. Luego analícelos en un desensamblador sin mirar el código fuente. Asegúrese de hacer esto tanto para x86 como para x64.
- Revise la presentación de Recon 2011 Descompilación de controladores del kernel e IDA complementos, de Bruce Dang y Rolf Rolles.
- Lea la documentación del depurador de Microsoft para obtener extensiones de kernel útiles (por ejemplo, !proceso, !hilo, !pcr, !devobj, !drvobj, etc.)
- Lea todos los artículos publicados en The NT Insider y los artículos relacionados con el kernel en Uninformed. El primero es probablemente el recurso más útil para el desarrollo de controladores del kernel de Windows en general. El segundo está más orientado a los entusiastas de la seguridad.
- Realice todos los ejercicios que se encuentran al final de este capítulo. Todos ellos. Algunos pueden requerir una cantidad considerable de tiempo porque deberá leer sobre áreas no documentadas que no se cubren en el libro. Leer y explorar son pasos en el proceso de aprendizaje.
- Abra el binario del kernel de Windows en un desensamblador e intente comprender Cómo funcionan algunas de las API más comunes.
- Lea el <http://kernel-mode.info> foros.
- Analice tantos rootkits como pueda. Mientras analiza, piense en por qué y cómo el autor del rootkit eligió utilizar determinados objetos o mecanismos y evalúe si son apropiados.
- Busque y lea controladores de Windows de código abierto.
- Una vez que crea que comprende bien los conceptos básicos, puede explorar otras áreas del núcleo, como la red y las pilas de almacenamiento. Se trata de dos áreas sumamente complejas, por lo que necesitará mucho tiempo y paciencia.
- Suscríbase a las listas de correo NTDEV y NTFSD para leer sobre otros Problemas de los desarrolladores y cómo los resolvieron.

¡Sigue leyendo, practicando y aprendiendo! Hay una curva de aprendizaje empinada, pero una vez que la superas, todo es pan comido. Recuerda: sin fracasos, es difícil apreciar el éxito. ¡Que disfrutes de la comprobación de errores!

Ceremonias

Creemos que la mejor manera de aprender es mediante una combinación de debates conceptuales , tutoriales prácticos y ejercicios independientes. Los dos primeros puntos se han tratado en las secciones anteriores. Los siguientes ejercicios independientes se han diseñado para ayudarle a ganar confianza, consolidar su comprensión de los conceptos del núcleo de Windows, explorar y ampliar el conocimiento en áreas no cubiertas en el libro y continuar analizando controladores del mundo real. Al igual que en otros capítulos, todos los ejercicios se han tomado de escenarios del mundo real. Hacemos referencia a los archivos como (Muestra A, B, C, etc.). El hash SHA1 de cada muestra se incluye en el Apéndice.

Desarrollar la confianza y consolidar sus conocimientos

Cada uno de estos ejercicios puede resolverse en 30 minutos. Algunos pueden requerir lectura o reflexión adicional, por lo que pueden llevar más tiempo.

1. Explique por qué el código que se ejecuta en DISPATCH_LEVEL no puede detectar un error de página.
Puede haber múltiples explicaciones para esto. Deberías poder pensar en al menos dos.
2. Supongamos que lees un artículo en Internet sobre el núcleo de Windows y que afirma que los subprocessos en modo núcleo siempre tienen mayor prioridad que los subprocessos en modo usuario; por lo tanto, si escribes todo en modo núcleo, será más rápido. Evalúe la validez de esta afirmación utilizando su conocimiento de IRQL, distribución de subprocessos y prioridad de subprocessos.
3. Escriba un controlador para Windows 7/8 que imprima la dirección base de cada imagen recién cargada. Repita lo mismo para los procesos y subprocessos. Este controlador no necesita configurar ningún controlador IRP porque no necesita procesar solicitudes de usuarios u otros controladores.
4. Explique las implicaciones de seguridad del uso de METHOD_NEITHER y qué hacen los desarrolladores de controladores para mitigarlas.
5. Dada una dirección virtual en modo kernel, conviértala manualmente en una dirección física. Verifique su respuesta utilizando la extensión !vtop en el depurador del kernel.
6. Desarrolle un controlador que utilice todas las operaciones de lista e identifique todas las rutinas de lista en línea en formato de ensamblaje. ¿Existe un patrón genérico para cada rutina? Si es así, explícalos. Si no, explica por qué.
7. Aprendió sobre las listas enlazadas, pero el núcleo también admite tablas hash, árboles de búsqueda y mapas de bits. Investigue su uso y desarrolle un controlador que los utilice todos.
8. Explique cómo funciona la macro FIELD_OFFSET .

9. La función exportada ExGetCurrentProcessorCpuUsage no está documentada, pero una API de NDIS documentada, NdisGetCurrentProcessorCpuUsage, la utiliza internamente. Explique cómo funciona ExGetCurrentProcessorCpuUsage en Windows x64 y x86.

10. Explique cómo funciona KeGetCurrentIql en x86 y x64.

11. Explique cómo funcionan las siguientes API en Windows 7/8 en x86/x64/ARM:



12. Las estructuras PCR, PRCB, EPROCESS, KPROCESS, ETHREAD y KTHREAD almacenan mucha información útil. Desafortunadamente, todas ellas son estructuras opacas y pueden cambiar de una versión de Windows a la siguiente. Por lo tanto, muchos rootkits codifican offsets en estas estructuras. Investigue estas estructuras en Windows XP, 2003, Vista y 7 y observe las diferencias.

¿Puedes idear formas de obtener de forma genérica los desplazamientos de algunos campos útiles sin codificarlos? Si es así, ¿puedes hacerlo de forma que funcione en todas las plataformas mencionadas? (Sugerencia: puedes usar un desensamblador, coincidencia de patrones y distancia relativa).

13. La API MmGetPhysicalAddress toma una dirección virtual y devuelve su dirección física. A veces, la dirección física devuelta contiene datos basura. Explique por qué puede suceder esto y cómo mitigarlo.

14. Configure la firma de prueba en sus máquinas de 32 y 64 bits y firme su controlador. Verifique que funcione.

15. Explique cómo funciona AuxKlibGetImageExportDirectory . A continuación, explique cómo funcionan RtlImageNtHeader y RtlImageDirectoryEntryToData .

16. Supongamos que desea realizar un seguimiento de la vida y la muerte de los procesos. ¿Qué estructura de datos utilizaría y qué propiedades puede utilizar para identificar de forma única un proceso?

17. ¿Dónde se almacena la tabla de directorio de páginas (CR3 en x86 y TTBR en ARM)?
¿un proceso?

Investigando y ampliando sus conocimientos

Estos ejercicios requieren que investigues más sobre el tema. Es posible que tengas que desarrollar controladores utilizando API no documentadas o acceder a estructuras no documentadas. Deberías usar el conocimiento adquirido en los experimentos solo para el bien.

1. Muchos sistemas operativos modernos admiten una función llamada Prevención de ejecución de datos (DEP). A veces se la llama Nunca ejecutar (NX) o Nunca ejecutar (XN). Esta función simplemente bloquea la ejecución de código en páginas de memoria que no están marcadas como ejecutables. Investigue cómo se implementa esta función en hardware (x86, x64 y ARM) y cómo la admite el sistema operativo . Despúes de eso, investigue cómo se implementaría esta función sin ningún soporte de hardware.

2. Aunque cubrimos la idea básica detrás de los APC, no explicamos cómo usarlos. Investigue las API (no documentadas) relacionadas con los APC en modo kernel y cómo se usan. Escriba un controlador que las use.

3. Diseñe e implemente al menos dos métodos para ejecutar un proceso en modo usuario desde un controlador en modo kernel. Evalúe las ventajas y desventajas de cada método.

4. Suponga que está en un sistema SMP con cuatro procesadores y desea modificar un recurso global compartido. El recurso global está en un grupo no paginado y cualquier procesador puede modificarlo en cualquier momento. Diseñe un mecanismo de sincronización para modificar este recurso de forma segura. (Sugerencia: piense en IRQL y el despachador de subprocesos).

5. Escriba un controlador que bloquee todos los controladores futuros con el nombre "bda.sys". cargando.

6. Investigue cómo funciona la pila de entrada de Windows e implemente un registrador de teclado. El registrador de teclas se puede implementar de varias maneras diferentes (con y sin conexión). Evalúe las ventajas y desventajas de cada método de registro de teclas. ¿Es posible lograr que la aplicación reciba las pulsaciones de teclas?

7. Implemente una función que tome una dirección virtual y cambie su protección de página a legible, escribible y ejecutable. Repita la misma tarea para una dirección virtual que esté en el espacio de sesión (por ejemplo, win32k.sys).

8. Explicamos que DriverEntry es la primera función que se llama en un controlador. Explica qué función es la que realmente llama a esta rutina. ¿Cómo lo averiguaste?

9. El depurador del núcleo de Microsoft proporciona un mecanismo que interrumpe el depurador cuando se carga un controlador. Esto se hace mediante el comando "sxe !d:drivername" . Cree un controlador simple y experimente con este comando. Explique cómo funciona. Enumere todas las diferentes formas en que puede fallar.
10. Los depuradores en modo usuario pueden "congelar" fácilmente los subprocessos de un proceso; sin embargo, el depurador del núcleo no tiene la capacidad para hacerlo. Piense en una forma de congelar y descongelar un subprocesso en modo usuario del núcleo.
11. Los controladores utilizan temporizadores periódicos para ejecutar algo de forma regular . Desarrolle un controlador que imprima un mensaje de "hola" cada 10 minutos. Luego, diseñe una forma de modificar la expiración del temporizador después de que se haya puesto en cola. Puede utilizar un depurador para hacer esto.
12. Implemente un controlador que instale su propio controlador de interrupciones y valide que se pueda activar desde el modo de usuario. En Windows x64, se encontrará con PatchGuard, así que asegúrese de probarlo solo en modo de depuración.
13. Los privilegios de los procesos se definen mediante tokens. El privilegio más alto es LocalSystem (el proceso SYSTEM se ejecuta en este contexto). Desarrolle un controlador que cambie el privilegio de un proceso en ejecución de modo que se ejecute con el privilegio LocalSystem.
14. Windows Vista y versiones posteriores admiten operaciones criptográficas en modo kernel a través del controlador KSECDD. Si bien no está documentado en el archivo oficial WDK, se encuentra en MSDN bajo la biblioteca bcrypt de modo usuario. Desarrolle un controlador que utilice AES, RSA, MD5, SHA1 y un generador de números aleatorios.
15. Desarrolle un controlador que enumere la dirección y el nombre de todos los símbolos exportados en NTDLL, KERNEL32 y KERNELBASE. Repita lo mismo para USER32 y GDI32. ¿Encontró alguna dificultad? Si es así, ¿cómo la solucionó?
16. Desarrolle un controlador que conecte una función exportada en NTDLL en el proceso "explorer.exe" . Evalúe el mérito de su método. Investigue y evalúe otros métodos.
17. Desarrolle un controlador que se conecte al proceso SMSS.EXE y aplique un parche a una llamada del sistema win32k mientras se encuentre en ese contexto de proceso. Explique los problemas que encontró y cómo los resolvió.
18. Supongamos que alguien le dice que las excepciones en modo usuario nunca entran en el núcleo. Investigue cómo funciona el manejo de excepciones en modo usuario en Windows x86 y x64 y evalúe la afirmación antes mencionada.
19. Suponga que tiene un controlador malicioso en el sistema que intercepta INT 1 e INT 3 para dificultar la depuración y el seguimiento. Piense en una forma de obtener un

Rastreo de ejecución (o código de depuración) incluso con estos ganchos instalados.
No tiene restricciones. ¿Cuáles son algunos de los casos especiales que debe manejar?

20. La instrucción INT 3 se puede representar de dos formas. La versión de un byte , 0xCC, es la más común. La forma de dos bytes, menos común, es 0xCD03. Explique qué sucede cuando utiliza el formato de dos bytes en Windows.

Análisis de factores de la vida real

Estos ejercicios están pensados para que practiques tus habilidades analíticas en un controlador real. Te proporcionamos los hashes de los archivos y te hacemos preguntas sobre ellos. La mayoría de las preguntas (si no todas) se pueden responder mediante un análisis estático, pero puedes ejecutar la muestra si es necesario.

1. (Muestra D) Analice y explique qué hace la función 0x10001277 .
¿De dónde viene el segundo argumento? ¿Puede ser inválido en algún caso?
¿Qué hacen las funciones en el desplazamiento 0x100012B0 y 0x100012BC ?
2. (Ejemplo E) Este archivo es bastante grande y complejo; algunas de sus estructuras son enormes (casi 4000 bytes de tamaño). Sin embargo, contiene funciones que realizan tareas interesantes que se trataron en el capítulo, por lo que varios de los ejercicios se han tomado de él. Para este ejercicio, recupere el prototipo de las funciones 0x40400D, 0x403ECC , 0x403FAD , 0x403F48, 0x404088, 0x4057B8, 0x404102 y 0x405C7C, y explique las diferencias y relaciones entre ellas (si las hay); explique cómo llegó a la solución. A continuación, explique la importancia de la asignación de un grupo no paginado de 0x30 bytes en las funciones 0x403F48, 0x403ECC y 0x403FA; mientras lo hace, recupere también su tipo. Además, explique por qué en algunas de las rutinas anteriores hay una operación de liberación de grupo al principio. Estas rutinas utilizan funciones no documentadas, por lo que es posible que deba buscar el prototipo en Internet.
3. (Ejemplo E) En DriverEntry, identifique todos los subprocessos de trabajo del sistema. En el desplazamiento 0x402C12, se crea un subprocesso del sistema para hacer algo rutinario utilizando una técnica interesante. Analice y explique el objetivo de la función 0x405775 y todas las funciones que llama. En particular, explique el mecanismo utilizado en la función 0x403D65. Cuando comprenda el mecanismo, escriba un controlador para realizar el mismo truco (pero aplicado a una solicitud de E/S diferente). Complete el ejercicio descompilando las cuatro rutinas. Este ejercicio es muy instructivo y le resultará muy útil.
4. (Ejemplo E) La función 0x402CEC toma el objeto de dispositivo asociado con \Device\Disk\DR0 como uno de sus parámetros y le envía una solicitud mediante IoBuildDeviceControlRequest. Este objeto de dispositivo describe la primera partición de su unidad de arranque. Descodifique el IOCTL que utiliza y encuentre el nombre significativo para él. (Sugerencia: busque todos los archivos incluidos en el WDK, incluidos los archivos de modo de usuario). Identifique la estructura asociada con esta solicitud.

A continuación, embellezca la salida de IDA de modo que cada variable local tenga un tipo y un nombre significativo. Por último, descompila la rutina de nuevo en C y explica lo que hace (quizás incluso escriba otro controlador que utilice este método).

5. (Ejemplo E) Descompila la función 0x401031 y dale un nombre significativo. A menos que estés familiarizado con el funcionamiento de SCSI, se recomienda que leas el Manual de referencia de comandos SCSI.
6. (Ejemplo F) Explique qué hace la función 0x100051D2 y por qué. ¿Qué tiene de especial el desplazamiento 0x38 en la estructura de extensión del dispositivo? Recupere tantos tipos como sea posible y descompila esta rutina. Por último, identifique todos los temporizadores, DPC y elementos de trabajo utilizados por el controlador.

CAPÍTULO

4

Depuración y automatización

Los depuradores son programas que aprovechan el apoyo del procesador y del sistema operativo para permitir el seguimiento de otros programas, de modo que se puedan descubrir errores o simplemente comprender la lógica del programa depurado. Los depuradores son una herramienta esencial para los ingenieros inversos porque, a diferencia de los desensambladores, permiten la inspección en tiempo de ejecución del estado del programa.

El objetivo de este capítulo es familiarizarse con las herramientas de depuración gratuitas de Microsoft. No pretende enseñarle técnicas de depuración ni cómo solucionar problemas de fugas de memoria, bloqueos, etc. En cambio, se centra en los comandos y las funciones de automatización y creación de scripts más importantes, y en cómo escribir extensiones de depuración con el único fin de ayudarlo en tareas de ingeniería inversa.

El capítulo cubre los siguientes temas:

- Herramientas de depuración y comandos básicos: esta sección cubre los conceptos básicos de depuración, varios comandos, evaluaciones de expresiones y operadores, comandos relacionados con procesos y subprocesos, y manipulación de memoria.

- Scripting: el lenguaje de scripting del motor del depurador no es muy fácil de usar. Esta sección explica el lenguaje de una manera estructurada y fácil de seguir, con varios ejemplos y un conjunto de scripts para ilustrar cada tema. Después de leer esta sección, comenzará a aprovechar el poder del scripting en el depurador.
- Uso del SDK: cuando los scripts no son suficientes, siempre puede escribir extensiones en C o C++. Esta sección describe los conceptos básicos de la escritura de extensiones en C/C++.

Las herramientas de depuración y los comandos básicos

El paquete de herramientas de depuración para Windows es un conjunto de utilidades de depuración que se pueden descargar de forma gratuita desde el sitio web de Microsoft. El conjunto de herramientas incluye cuatro depuradores que se basan en el mismo motor de depuración (DbgEng).

DbgEng es un objeto COM que permite que otros programas utilicen API de depuración avanzadas en lugar de las API de depuración de Windows. De hecho, el paquete de herramientas de depuración incluye un SDK que muestra cómo escribir extensiones para DbgEng o alojarlo en sus propios programas.

El paquete Herramientas de depuración para Windows incluye los siguientes depuradores:

- NTSD/CDB: Microsoft NT Symbolic Debugger (NTSD) y Microsoft Console Debugger (CDB) son idénticos excepto que el primero crea una nueva ventana de consola cuando se inicia, mientras que el segundo hereda la ventana de consola que se utilizó para iniciar lo.
- WinDbg: esta es una interfaz gráfica para DbgEng. Admite código fuente, depuración de niveles y guardado de espacios de trabajo.
- KD—Kernel Debugger (KD) se utiliza para depurar el kernel.

Los depuradores tienen un amplio conjunto de parámetros de línea de comandos. Un parámetro particularmente útil es -z, que se utiliza para analizar archivos de volcado de memoria (*.dmp) y archivos cab (*.cab) que contienen un archivo de volcado de memoria. Otro uso del parámetro -z es analizar archivos PE (ejecutables o DLL) haciendo que DbgEng los asigne como si estuvieran en un volcado de memoria.

El siguiente ejemplo ejecuta el depurador cdb con el modificador -z para mapear calc.exe en el depurador:

```
C:\>cdb -z c:\windows\syswow64\calc.exe
```

```
Depurador de Microsoft (R) Windows versión 6.13.0009.1140 X86  
Copyright (c) Microsoft Corporation. Reservados todos los derechos.
```

```
Cargando archivo de volcado [c:\windows\syswow64\calc.exe]  
La ruta de búsqueda de símbolos es: SRV*C\cache*http://msdl.microsoft.com/download/  
símbolos
```

```
La ruta de búsqueda ejecutable es:  
Carga del módulo: 00400000 004c7000 c:\windows\syswow64\calc.exe  
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000  
eip=0041a592 esp=00000000 ebp=00000000 iopl=0 cs=0000 ss=0000  
ds=0000 es=0000 fs=0000 gs=0000 calc!WinMainCRTStartup: nv arriba de pl nz na po nc  
efl=00000000  
0041a592 e84bf0ffff llamar calc!_security_init_cookie (004195e2)  
0:000>
```

Tenga en cuenta dos cosas:

- **Calc.exe se asignó al depurador y EIP apunta a su punto de entrada** (a diferencia de los objetivos activos, que apuntan dentro de ntdll.dll).
- Muchos comandos del depurador no estarán presentes, especialmente los comandos de control de proceso (porque el programa está asignado para análisis/inspección, no para seguimiento/depuración dinámicos).

Usando el modificador -z , puedes escribir scripts potentes para analizar programas y Extraer información.

NOTA: Puede configurar WinDbg para que actúe como depurador just-in-time (JIT) (para fines de depuración post mortem) ejecutando Windbg.exe -l una vez como usuario privilegiado.

Las siguientes secciones explican varios comandos del depurador, proporcionando ejemplos a lo largo del camino.

Configuración de la ruta del símbolo

Antes de iniciar cualquiera de los depuradores (WinDbg, CDB, NTSD o KD), configuremos la variable de entorno _NT_SYMBOL_PATH :

```
_NT_SYMBOL_PATH=SRV*c:\cache*http://msdl.microsoft.com/download/symbols
```

También puedes configurarlo desde dentro del depurador usando el comando .sympath :

NOTA: Es importante configurar la ruta de símbolos para poder inspeccionar algunas estructuras básicas del sistema operativo mientras depura los programas en cuestión. Por ejemplo, el comando de extensión !peb no funcionará sin símbolos cargados para NTDLL.

Ventanas del depurador

Las siguientes ventanas, incluidas sus teclas de acceso rápido cuando corresponda, están expuestas en WinDbg:

- **Ventana de comandos/salida (Alt+1):** esta ventana le permite escribir comandos y ver el resultado de las operaciones. Si bien es posible depurar

Al utilizar otras ventanas y elementos de menú, la ventana de comandos le permite aprovechar todo el poder de los comandos integrados de DbgEng y las extensiones disponibles.

- Ventana de registros (Alt+4): muestra los registros configurados. Es posible personalizar esta vista para controlar qué registros se muestran u ocultan.
- Memoria (Alt+5): ventana de volcado de memoria. Esta ventana le permite ver el contenido de la memoria, así como desplazarse por ella, copiarla e incluso editarla .
- Llamadas (Alt+6): muestra la información de la pila de llamadas.
- Desensamblado (Alt+7): mientras que la ventana de comandos mostrará la lista de desensamblado de instrucciones actual, la ventana de desensamblado mostrará una página de código desensamblado. En esta ventana también es posible realizar acciones con teclas de acceso rápido:
 - Agregar o eliminar puntos de interrupción en la línea seleccionada (F9)
 - Control de procesos (paso a paso/F11, reanudación/F5, etc.)
 - Navegación (Página arriba/Página abajo para explorar el código desensamblado)

NOTA: WinDbg admite espacios de trabajo que permiten guardar o restaurar la configuración de la ventana.

Evaluando expresiones

El depurador entiende dos sintaxis para la evaluación de expresiones: Microsoft Macro Assembler (MASM) y C++.

Para determinar el evaluador de expresión predeterminado, utilice .expr sin ningún argumento:

```
0:000>.expr  
Evaluador de expresiones actuales: MASM - Expresiones de Microsoft Assembler
```

Para cambiar la sintaxis de evaluación de expresión actual, utilice

```
0:000>.expr/s c++  
Evaluador de expresiones actual: C++ - Expresiones fuente de C++
```

O

```
0:000> .expr /s masm  
Evaluador de expresiones actuales: MASM - Expresiones de Microsoft Assembler
```

Utilice el comando ? para evaluar expresiones (utilizando la sintaxis predeterminada).

El comando ?? se utiliza para evaluar una expresión de C++ (sin tener en cuenta el sintaxis seleccionada por defecto).

NOTA: La sintaxis C++ es preferible cuando hay información de tipo/símbolo y necesita acceder a miembros de la estructura o simplemente aprovechar los operadores de C++.

Los números, si no tienen como prefijo un especificador de base, se interpretan utilizando la configuración de base predeterminada. Utilice el comando `n` para mostrar la base numérica actual o `n` `base_value` para configurar la nueva base predeterminada.

Al utilizar la sintaxis MASM, puede expresar un número en una base de su elección, utilice los siguientes prefijos:

- `0n123` para decimal
- `0x123` para hexadecimal
- `0t123` para octal
- `0y10101` para binario

A diferencia de la evaluación con la sintaxis MASM, al usar `??` para evaluar comandos , no es posible anular el radix:

```
?0y101 -> funciona  
?? 0y101 -> no funciona.
```

NOTA: Cuando el valor predeterminado es 16 y se intenta evaluar una expresión como `abc`, se puede confundir entre un símbolo llamado `abc` o el número hexadecimal `abc` (2748 decimal). Para resolver el símbolo en su lugar, anteponga `!` antes de la variable. nombre: `? !abc`.

Al igual que en el lenguaje C++, la sintaxis del evaluador de C++ solo permite el prefijo `0x` para números hexadecimales y el prefijo `0` para números octales. Si no se especifica ningún prefijo, se utiliza la base 10.

Para mezclar y combinar varios tipos de expresiones, utilice `@@@c++(expresión)` o `@@@masm(expresión)`:

```
0:000>.expr  
Evaluador de expresiones actuales: MASM - Expresiones de Microsoft Assembler  
0:000> ? @@@c++(@$peb->VersiónPrincipal del Subsistema de Imagen) + @@@masm(0y1)  
Evaluar expresión: 7 = 00000007
```

El prefijo `@@@` es un prefijo abreviado que se puede utilizar para indicar la sintaxis de evaluación de expresión alternativa (no la sintaxis establecida actualmente):

```
0:000>.expr  
Evaluador de expresiones actuales: MASM - Expresiones de Microsoft Assembler  
0:000> ? @@@(@$peb->VersiónPrincipal del Subsistema de Imagen) + @@@masm(0y1)  
Evaluar expresión: 7 = 00000007
```

No es necesario especificar `@@@c++(...)` porque cuando MASM es el **predeterminado**, `@@@(...)` utilizará la sintaxis C++ y viceversa.

Operadores útiles

Esta sección ilustra varios operadores útiles que se pueden utilizar en expresiones. Para fines de demostración, utilizamos los pseudo-registros predefinidos \$ip y \$peb, que denotan el puntero de instrucción actual y el _PEB * del proceso actual, respectivamente. Otros pseudo-registros se mencionan más adelante en el capítulo.

La notación utilizada es “operador (sintaxis de expresión)”, donde la sintaxis de expresión será C++ o MASM. Tenga en cuenta que en los siguientes ejemplos, el evaluador de expresiones MASM está configurado de forma predeterminada.

- Puntero->Campo (C++): como en el ejemplo anterior, se utiliza el operador de flecha para acceder al valor del campo señalado por \$peb y al desplazamiento del campo ImageSubsystemMajorVersion .
- sizeof(type) (C++): este operador devuelve el tamaño de la estructura. Esto puede resultar útil cuando intenta analizar estructuras de datos o escribir puntos de interrupción condicionales potentes:

```
0:000> ? @@c++(tamaño de(_PEB))  
Evaluar expresión: 592 = 00000250
```

- #FIELD_OFFSET(Tipo, Campo) (C++)—Esta macro devuelve el desplazamiento de bytes del campo en el tipo:

```
0:000> ? #FIELD_OFFSET(_PEB, Versión principal del subsistema de imagen)  
Evaluar expresión: 184 = 000000b8
```

- El operador ternario (C++): este operador se comporta como lo hace en el Lenguaje C++:

```
0:000> ? @@c++(@$peb->ImageSubsystemMajorVersion >= 6? 1: 0)  
Evaluar expresión: 1 = 00000001
```

- (tipo) Valor (C++): la conversión de tipos le permite convertir de un tipo a otro:

```
0:000> ? #FIELD_OFFSET(_PEB, en proceso de depuración)  
Evaluar expresión: 2 = 00000002  
0:000> ? @$peb  
Evaluar expresión: 2118967296 = 7e4ce000  
0:000> ? #FIELD_OFFSET(_PEB, siendo depurado) + (char *)@$peb  
Evaluar expresión: 2118967298 = 7e4ce002
```

Tenga en cuenta que debe convertir @\$peb a (char*) antes de agregarle el desplazamiento de BeingDebugged.

- *(puntero) (C++)—Operador de desreferenciación:

```
0:000> dd @$ip L 4  
012a9615 2ec048a3 8b5e5f01 90c35de5 90909090  
0:000> ? *( (largo sin signo *)0x12a9615 )  
Evaluar expresión: 784353443 = 2ec048a3
```

Tenga en cuenta que antes de desreferenciar el puntero, debe darle un tipo adecuado (mediante su

conversión). ■ poi(dirección) (MASM)—Desreferenciación de puntero:

```
0:000> ? @@masm(poi(0x12a9615))
Evaluuar expresión: 784353443 = 2ec048a3
```

■ hallow(número) (MASM): Devuelve el valor alto o bajo de 16 bits de un número:

```
0:000> ? hola(0x11223344)
Evaluuar expresión: 4386 = 00001122
0:000> ? bajol(0x11223344)
Evaluuar expresión: 13124 = 00003344
```

■ by/wo/dwo(dirección) (MASM): devuelve el valor de byte/palabra/dword cuando se desreferencia la dirección:

```
0:000> db @$ip L 4
012a9615a3480000
0:000> ? por (@$ip)
Evaluuar expresión: 163 = 000000a3 0:000> ? wo(@$ip)

Evaluuar expresión: 18595 = 000048a3 0:000> ? dwo(@$ip)

Evaluuar expresión: 18595 = 000048a3
```

■ pointer[index] (C++): el operador de subíndice de matriz le permite desreferenciar la memoria mediante índices:

```
0:000> db @ $ip L 10
012a9615 a3 48 c0 2e 01 5f 5e 8b e5 5d
0:000> ? @@c++(((carácter sin signo *)@$ip)(3))
Evaluuar expresión: 46 = 0000002e
```

Se puede lograr lo mismo utilizando la sintaxis MASM y poi() o by():

```
0:000> ? poi(@$ip+3) y 0xff
Evaluuar expresión: 46 = 0000002e 0:000> ? por(@$ip+3)

Evaluuar expresión: 46 = 0000002e
```

NOTA: Cuando se utiliza el puntero[indice] , se tendrá en cuenta el tamaño del tipo base (a diferencia de poi(), para el cual se debe tener en cuenta el tamaño del tipo).

■ \$scmp("string1", "string2")/\$sicmp("String1", "String2") (MASM)— Comparación de cadenas (distingue entre mayúsculas y minúsculas/sin distinción entre mayúsculas y minúsculas). Devuelve -1, 0 o 1, como en strcmp() / strcmp() de C :

```
0:000> ? $scmp("práctica", "práctica")
Evaluuar expresión: 1 = 00000001
```

```
0:000> ? $scmp("práctico", "práctico")
Evaluar expresión: 0 = 00000000
0:000> ? $scmp("práctica", "práctico")
Evaluar expresión: -1 = ffffffff
0:000> ? $scmp("Práctico", "práctico")
Evaluar expresión: -1 = ffffffff
0:000> ? $sicmp("Práctico", "práctico")
Evaluar expresión: 0 = 00000000
```

- **\$iment(address) (MASM)**: devuelve el punto de entrada de la imagen existente en esa dirección. El encabezado PE se analiza y se utiliza:

```
0:000> !mvm ole32
comenzar      fin          nombre del módulo
74b70000 74c79000 ole32
...
0:000> ? $imento(74b70000)
Evaluar expresión: 1958154432 = 74b710c0
0:000> u $imento(74b70000)
ole32!_DIIIMainCRTInicio:
74b710c08bff      movimiento    edito,edito
74b710c2 55        empujar       .....
74b710c38bec      movimiento    ebp,esp
```

- **\$vvalid(dirección, longitud) (MASM)** — Verifica si la memoria a la que apunta la dirección hasta dirección + longitud es accesible (devuelve 1) o inaccesible (devuelve 0):

```
0:000> ? @@masm($válido(@$ip, 100))
Evaluar expresión: 1 = 00000001
0:000> ? @@masm($vvalid(0x0, 100))
Evaluar expresión: 0 = 00000000
```

- **\$spat("string", "pattern") (MASM)**—Utiliza la coincidencia de patrones para determinar si el patrón existe en la cadena y devuelve verdadero o falso.

Control de procesos y eventos de debut

Esta sección presenta los comandos básicos de control de procesos (como paso único, paso a paso, etc.) y los comandos que se pueden usar para cambiar la forma en que el depurador reacciona a ciertos eventos de depuración.

Control de procesos y subprocessos

Estos son algunos comandos que le permiten controlar el flujo del depurador:

- **t (F11)**—Entrar.
- **gu (Shift+F11)**—Subir. Sale de la función actual y regresa a la llamador.

- p (F10)—Pasar por encima.
- g (F5)—Ir. Reanuda la ejecución del programa.
- Ctrl+Interrumpir: cuando el proceso depurado se esté ejecutando, utilice esta tecla de acceso rápido para suspenderlo.

Tenga en cuenta que los comandos anteriores solo funcionan con objetivos vivos.

Existen variaciones útiles de las expresiones “reanudar”, “entrar” y “pasar por encima”. instrucciones, incluyendo lo siguiente:

- [tp]a Dirección: ingresar . Pasa hasta llegar a la dirección especificada.
- gc: se utiliza para reanudar la ejecución cuando un punto de interrupción condicional suspende la ejecución.
- g[h|n]—Esto se utiliza para reanudar la ejecución como manejada o no manejada cuando ocurre una excepción.

Otro conjunto de comandos de seguimiento/pasos es útil para descubrir bloques básicos:

- [pt]c—Pasar por encima/dentro hasta que se encuentre una instrucción CALL .
- [pt]h—Pasar por encima/entra hasta que se encuentre una instrucción de bifurcación (todas tipos de instrucciones de salto, retorno o llamada).
- [pt]t—Pasar por encima/dentro hasta que se encuentre una instrucción RET .
- [pt]ct—Pasar por encima/dentro hasta que se encuentre una instrucción CALL o RET .

La mayoría de los comandos anteriores (trazar y pasar por encima) están implícitos. operando dentro del contexto del hilo actual.

Para listar todos los hilos, utilice el comando ~ :

```
0:004> ~
0 Id: 1224.13d8 Suspender: 1 Teb: ff4ab000 Descongelado
1 Id: 1224.1758 Suspender: 1 Teb: ff4a5000 Descongelado
2 Id: 1224.2920 Suspender: 1 Teb: ff37f000 Descongelado
3 Id: 1224.1514 Suspender: 1 Teb: ff37c000 Descongelado
· 4 Id: 1224.b0 Suspendido: 1 Teb: ff2f7000 Descongelado
```

La primera columna es el número de hilo (decidido por DbgEng), seguido de un par de SystemProcessId.SystemThreadId en formato hexadecimal.

Los comandos DbgEng funcionan con ID de DbgEng, en lugar de con ID de procesos o subprocesos del sistema operativo.

Para cambiar a otro hilo, utilice el comando ~Ns , donde N es el hilo. número al que desea cambiar:

```
0:004> ~1 s
eax=00000000 ebx=00bb1ab0 ecx=00000000 edx=00000000 esi=02faf9ec edi=00b2ec00
eip=7712c46c esp=02faf8a4 ebp=02fafaa4 iopl=0
es=002b fs=0053 gs=002b
nv arriba ei pl nz na po nc
efl=000000202
```

196 Capítulo 4 ■ Depuración y automatización

```
ntdll!NIWaitForWorkViaWorkerFactory+0xc:
7712c46c c21400          retirado      14 horas
0:001>
```

El indicador del depurador también muestra el ID del hilo seleccionado en el indicador. ProcesOID:ThreadId>.

No es necesario cambiar a los subprocessos antes de emitir un comando; por ejemplo, para mostrar los registros del ID de subprocesso 3, utilice el prefijo ~3 seguido del comando de depuración deseado (en este caso, el comando r):

```
0:001> ~3r
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000001 edi=00000001
eip=7712af2c esp=031afb38 ebp=031afcbb iopl=0          nv arriba ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b nt!NIWaitForMultipleObjects+0xc:    efl=00000202
                                         . . .
7712af2c c21400          retirado      14 horas
0:001> ~3t
eax=00000000 ebx=00000000 ecx=77072772 edx=00000000 esi=00000001 edi=00000001
eip=758c11b5 esp=031afb50 ebp=031afcbb iopl=0          nv arriba ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b KERNELBASE!    efl=00000202
EsperaAMúltiplesObjetosEx+0xdc:
758c11b58bf8          . . .          edi,eax
```

Para mostrar los valores de registro de todos los hilos, simplemente pase * como el número de hilo.

NOTA: A todos los comandos del depurador pueden tener el prefijo ~N cmd para que proporcionen información sobre el hilo N. En su lugar, utilice el comando específico del hilo ~eN cmd.

Si está depurando varios procesos en modo usuario (es decir, cuando el depurador se inicia con el modificador -o), es posible cambiar de un proceso a otro utilizando el comando | . El siguiente ejemplo utiliza Internet Explorer porque normalmente genera varios procesos secundarios (con diferentes niveles de integridad y para diversos propósitos):

```
C:\dbg64>windbg -o "c:\Archivos de programa (x86)\Internet Explorer\iexplore.exe"
```

Déjelo ejecutar, abra algunas pestañas y luego deje que el depurador se reanude con g y luego suspéndalo y escriba |:

```
0:030> |
- 0 id: 1818          Nombre del niño: iexplore.exe
1 id: 1384          Nombre del niño: iexplore.exe
```

Para cambiar de un proceso a otro, escriba |Ns, donde N es el número de proceso:

```
0:030> |1s
1:083> |
# 0 id: 1818          Nombre del niño: iexplore.exe
.   1 id: 1384          Nombre del niño: iexplore.exe
```

Una vez que cambie a un nuevo proceso, los comandos futuros se aplicarán a este proceso. Los puntos de interrupción que establezca para un proceso no estarán presentes en el otro proceso.

NOTA Los alias y pseudoregistros serán comunes a todos los procesos que se estén depurando.

Supervisión de eventos de depuración y excepciones

Es posible capturar ciertos eventos de depuración y excepciones a medida que ocurren y dejar que el depurador suspenda, muestre, maneje, deje sin manejar o simplemente ignore el evento por completo.

DbgEng puede suspender el objetivo y darle al usuario la oportunidad de decidir qué acción tomar en las siguientes dos circunstancias:

- **Excepciones:** estos eventos ocurren cuando se activa una excepción en el contexto de la aplicación (violación de acceso, división por cero, excepción de un solo paso, etc.).
- **Eventos:** estos eventos no son errores, son activados por el sistema operativo para notificar al depurador sobre ciertas actividades que tienen lugar (se ha creado o finalizado un nuevo hilo, se ha cargado o descargado un módulo, se ha creado o finalizado un nuevo proceso, etc.).

Para listar todos los eventos, utilice el comando sx . De la misma manera, si está utilizando WinDbg, puede navegar al menú Depurar/Filtros de eventos para configurar gráficamente los eventos, como se muestra en la Figura 4-1.

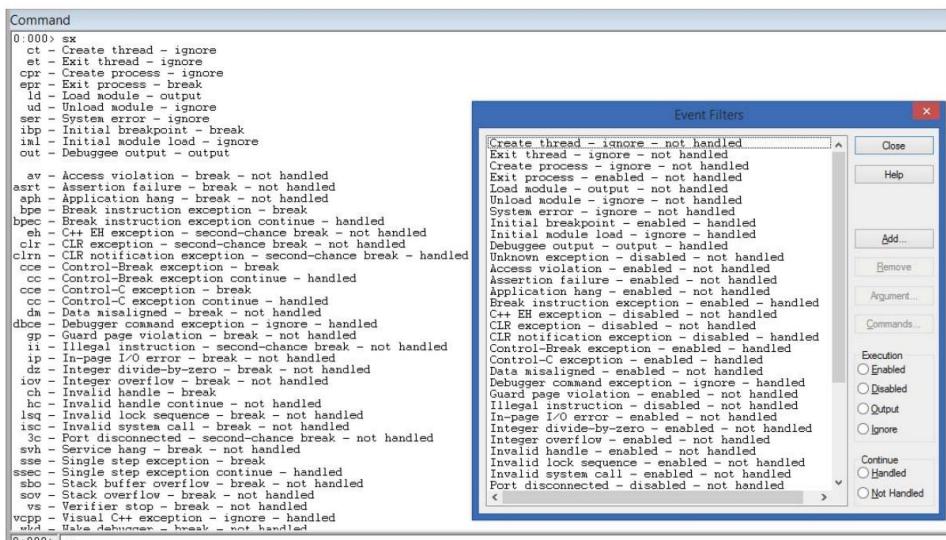


Figura 4-1

La captura de pantalla muestra dos conjuntos de configuración para controlar eventos:

- Ejecución: dicta qué hacer cuando ocurre ese evento.
- Continuar: decide cómo reanudar el evento o la excepción.
 - Manejado: marca la excepción como manejada (el manejador de excepciones de la aplicación no se activará). Esto es útil cuando el depurador falla y usted corrige la situación manualmente y luego reanuda la aplicación con el comando gh .
 - No manejado: permite que el controlador de excepciones de la aplicación se encargue de La excepción. Utilice el comando gn para reanudar.

Utilice los siguientes comandos para controlar cómo se manejan los eventos/excepciones:

- evento sxe : permite interrumpir un evento
- Evento sxd : deshabilita la interrupción de un evento
- evento sxr : habilita la salida solo para un evento
- Evento sxi : ignora el evento (no genera ningún resultado)

El parámetro de evento puede ser un número de código de excepción o un código corto de evento. nombre, o * para cualquier evento.

Una aplicación bastante útil de los comandos sxe o sxd es detectar la carga o descarga de módulos. Por ejemplo, al depurar el núcleo, para detener el depurador cuando se carga un determinado controlador, utilice el siguiente comando:

`sxe Id.nombre_del_controlador.sys`

Para asociar un comando con un evento, utilice el comando sx- -c event . Por ejemplo, para mostrar la pila de llamadas cada vez que se carga un módulo, utilice el siguiente comando:

`sx- -c "k" Id`

Registros, memoria y símbolos

Esta sección cubre algunos de los comandos útiles que tratan con la administración de registros , inspección y modificación del contenido de la memoria, símbolos, estructuras y otros comandos útiles.

Registros

El comando r se utiliza para mostrar valores de registros o para cambiarlos.

NOTA: El comando r también se puede utilizar para modificar alias de nombres fijos y valores de pseudoregistros. Este uso se explica en las secciones siguientes.

La sintaxis general del comando r es la siguiente:

```
r[Máscara M|F|X] [Nombre_registro_o_nombre_bandera|[Num]Tipo] [=Expresión_o_valor]]
```

Aquí está la sintaxis más simple del comando r :

```
r RegisterName|FlagName [= Expresión_o_Value ]
```

Si se omite la expresión o el valor, r mostrará el valor actual del registro:

```
0:001> r eax
eax=ffffda000
0:001> r eax = 2
0:001> r eax
eax=00000002
```

Para mostrar los registros involucrados en la instrucción actual, utilice el comando r .:

```
0:000> te rompo L1
00007ff6`f54d6470 488995c2420           movamento      qword ptr [rsp+20h],rbx
0:000> r.
rsp=0000000c9`e256fb8 rbx=00000000`00000000 0:000> en eip L1

usuario32!MessageBoxA+0x3:
773922c58bec           movimento      ebp,esp
0:000> r.
esp= 0018ff78
```

Registrar máscarillas

El comando r puede tener como sufijo el carácter M seguido de un valor de máscara de 32 bits . La máscara designa qué registros se mostrarán cuando se escriba r sin parámetros. La Tabla 4-1 muestra una lista breve de los valores de máscara:

Tabla 4-1: Valores de la máscara de registro

DESCRIPCIÓN DEL VALOR DE LA MÁSCARA DE REGISTRO

2	Registros generales
4	Registros de punto flotante
8	Registros de segmentos
0x10	MMX
0x20	Registros de depuración
0x40	SSE XMM
0x80	Modo kernel: registros de control
0x100	Modo kernel: TSS

NOTA Utilice el operador OR (|) para combinar varias máscaras.

Para ver la máscara actual, escriba rm:

```
0:000> rm
La máscara de salida del registro es:
 2 - Estado entero (64 bits)
 8 - Registros de segmento
```

Ahora, si ejecuta r, debería ver solo los registros de propósito general y el registros de segmento:

```
eax=025ad9d4 ebx=00000000 ecx=7c91056d edx=00ba0000 esi=7c810976 edi=10000080
eip=7c810978 esp=025ad780 ebp=025adbec iopl=0
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
nv arriba ei pl nz na po nc
efl=00000202
```

Para mostrar todos los registros posibles, configure todos los bits en uno en el parámetro de máscara (máscara 0x1ff):

```
kd> rM1ff
eax=025ad9d4 ebx=00000000 ecx=7c91056d edx=00ba0000 esi=7c810976 edi=10000080
eip=7c810978 esp=025ad780 ebp=025adbec iopl=0
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 fpcw=007F: rn 53 puede fpsw=0000: top=0 cc=0000
----- fptw=FFFF fopcode=0000 ftiip=0000.00000000 fpdp=0000.00000000 st0= 0.0000000000000000e+0000 st1=
0.303405511757512497160e-4933

st2=-3.685298464319287816590e-4320 st3= 0.00000015933281407050e-4357
st4=-0.008610620845784322250e-4310 st5= 0.000000125598791309870e-4184
st6=-0.008011795206688037930e+0474 st7=-1.#QNAN0000000000000000e+0000
mm0=0000000000000000 mm1=0127b52000584c8e
mm2=2390ccb400318a24 mm3=000000057c910732
mm4=003187ec00000000 mm5=000000117c910732
mm6=003187ec00000000 mm7=7c9107387c90ee18
xmm0=1.79366e-043 0 6.02419e+036 6.02657e+036
xmm1= 3.08237e-038 3.08148e-038 0
xmm2=3.30832e-029 5.69433e-039 0 3.08147e-038
xmm3=5.6938e-039 0 9.62692e-043 5.69433e-039
xmm4=3.04894e-038 2.12997e-042 3.07319e-038 5.69433e-039
xmm5=5.69528e-039 6.02651e+036 4.54966e-039 1.16728e-042
xmm6=5.69567e-039 0 5.69509e-039 6.02419e+036
xmm7=4.54901e-039 5.69575e-039 0 5.69559e-039
cr0=8001003b cr2=7c99a3d8 cr3=07f40280
dr0=00000000 dr1=00000000 dr2=00000000
dr3=00000000 dr6=ffff4ff0 dr7=00000400 cr4=00000619
gdtr=8003f000 gdtr=03ff idtr=8003f400 idtr=07ff tr=0028 ldr=0000
```

NOTA Algunos registros del procesador (GDT, IDT, registros de control, etc.) solo se pueden mostrar en modo de depuración kernel.

Para establecer la máscara predeterminada, utilice el comando rm seguido del valor de máscara deseado:

```
0:000> rm2|4|8  
0:000> rm  
  
La máscara de salida del registro es f:  
 2 - Estado entero (64 bits)  
 4 - Estado de punto flotante  
 8 - Registros de segmento
```

DbgEng proporciona indicadores abreviados para ciertas máscaras, a saber, los registros de punto flotante y MMX.

Para mostrar registros de punto flotante, utilice rF; y para mostrar registros XMM, utilice rX:

```
0:000> rF  
fpcw=027F: 53 segundos pueden ser fpsw=4020: top=0 cc=1000 --p---- fptw=FFFF  
código fop=0000 fpip=0023:74b785bc fpdp=002b:00020a84  
st0= 0.0000000000000000000000000000e+0000 st1= 0.0000000000000000000000000000e+0000  
...  
0:000> rX  
xmm0=0 0 0 0  
xmm1=0 0 0 0  
xmm2=0 0 0 0  
...
```

Formato de visualización de

registros Es posible especificar cómo se deben visualizar los registros. Esto resulta muy útil en muchos casos, como se ilustra en los siguientes ejemplos.

Visualización de registros en formatos de punto flotante

Supongamos que está depurando y observa que el registro eax contiene un valor de punto flotante:

```
0:000> r eax  
eax=3f8ccccd
```

Para visualizarlo correctamente, utilice lo siguiente:

```
0:000> r eax:f  
eje=1.1
```

Para mostrar el contenido de rax en valor de punto flotante de doble precisión, use lo siguiente:

```
0:000> r rax  
rax=4014666666666666  
0:000> r rax:d  
rax=5.1
```

202 Capítulo 4 ■ Depuración y automatización

Visualización de registros en formatos Bytes/Word/Dword/Qword

Cuando los registros intervienen en la transferencia de datos, es útil ver los bytes individuales del registro:

```
msvcrt!memcpy+0x220:  
00007ff9`5f671a5d f30f7f40f0          movdqu xmmword ptr [rax-10h],xmm0  
0:000> rxmm0  
xmm0=          0 1.05612e-038 1.01939e-038 1.00102e-038  
0:000> rxmm0:ub  
xmm0=00 00 00 00 00 73 00 6c 00 6f 00 62 00 6d 00 79  
0:000> rX xmm0:uw  
xmm0=0000 0000 0073 006c 006f 0062 006d 0079  
0:000> rX xmm0:ud  
xmm0=00000000 0073006c 006f0062 006d0079  
0:000> rX xmm0:uq  
xmm0=000000000073006c 006f0062006d0079
```

En el ejemplo anterior, `memcpy()` utiliza los registros XMM para transferir 16 bytes a la vez. Se utiliza el formato ub para mostrar el contenido de `xmm0` en formato de bytes sin signo, uw para formato de palabra, ud para formato de doble palabra y uq para formato de cuatro palabras. Para mostrar en formato con signo, utilice el prefijo i en lugar de u.

Comando de selección de pantalla

El comando selector de pantalla tiene la siguiente sintaxis:

```
dg Primer Selector [Último Selector]
```

Muestra información sobre un selector determinado (o un rango de selectores). En este caso, le interesan los valores de selector que están configurados actualmente en uno de los registros x86/x64, es decir, los registros cs, ds, ss, gs y fs .

Los selectores se utilizan en la parte de segmento de una dirección en modo protegido.

El siguiente ejemplo ejecuta el comando dg para cs, ds, ss, gs y fs, respectivamente:

```
0:001> .foreach /s (sel "cs ds ss gs fs") { dg sel; }  
(Selector cs)  
Sal      Base      Límite      Tipo      P Si Gr Pr Lo  
-----  
0023 00000000 ffffff Código RE Ac 3 Bg Pg P NI 00000cfb  
(Selector DS)  
Sal      Base      Límite      Tipo      P Si Gr Pr Lo  
-----  
002B 00000000 ffffff Datos RW Ac 3 Bg Pg P NI 00000cf3  
(Selector de ss)
```

Sal	Base	Límite	Tipo	P Si Gr Pr Lo I ze an es ng Banderas
<hr/>				
	002B 00000000 ffffff Datos RW Ac 3 Bg Pg P NI 00000cf3			
(Selector gs)				
Sal	Base	Límite	Tipo	P Si Gr Pr Lo I ze an es ng Banderas
<hr/>				
	002B 00000000 ffffff Datos RW Ac 3 Bg Pg P NI 00000cf3			
(Selector de sistema)				
Sal	Base	Límite	Tipo	P Si Gr Pr Lo I ze an es ng Banderas
<hr/>				
	0053 7ffda000 00000fff Datos RW Ac 3 Bg Por P NI 000004f3			

En las aplicaciones de modo usuario/MS Windows, los selectores cs, ds, es, ss y gs tienen un valor base de cero, por lo tanto, la dirección lineal es la misma que la dirección virtual.

Por el contrario, el registro fs es variable y cambia su valor de un subproceso a otro. El segmento fs en los procesos en modo usuario apunta a la estructura TEB (bloque de entorno de subprocesos):

0 : 003> dgfs	Sal	Base	Límite	Tipo	I ze an es ng Banderas
<hr/>					
	0053 ff306000 00000fff Datos RW Ac 3 Bg Por P NI 000004f3				
(Cambiar a otro hilo)					
0:003> ~2s	Sal	Base	Límite	Tipo	I ze an es ng Banderas
<hr/>					
0 : 002> dgfs	Sal	Base	Límite	Tipo	I ze an es ng Banderas
<hr/>					
	0053 ff4a5000 00000fff Datos RW Ac 3 Bg Por P NI 000004f3				

Memoria

Antes de describir los comandos relacionados con la memoria, es importante explicar las notaciones de dirección y rango porque se pasan como argumentos a la mayoría de los comandos que requieren una dirección de memoria y un recuento.

El parámetro Dirección puede ser cualquier valor, expresión o símbolo que se resuelva en un valor numérico que pueda interpretarse como una dirección. El número 0x401000 Puede tratarse como una dirección si la dirección está asignada en la memoria. El nombre kernel32 se resolverá en la base de la imagen del módulo:

```
0:000> Imm kernel32
comenzar     fin           nombre del módulo
75830000 75970000 NÚCLEO32
0:000> ? kernel32
Evaluuar expresión: 1971519488 = 75830000
```

204 Capítulo 4 ■ Depuración y automatización

Un símbolo como module_name!SymbolName se puede utilizar como una dirección como mientras se resuelva:

```
0:000> ? kernel32!ObtenerDirecciónProc
```

No se pudo resolver el error en 'kernel32!GetProcAddress'

```
0:000> ? kernelbase!ObtenerDirecciónProc
```

Evaluar expresión: 1979722334 = 76002a5e

Es posible utilizar cualquier expresión como dirección (independientemente de si el valor se resuelve en una dirección válida o no):

```
0:000> ? (kernelbase!GetProcAddress - kernel32) / 0n4096
```

Evaluar expresión: 2002 = 000007d2

El parámetro Rango se puede especificar de dos maneras. El primer método es con un par de direcciones de inicio y fin:

```
0:000> db02c00000 02c0005
```

```
002c0000 23 01 00 00 00 00
```

#....

El segundo método es utilizar una dirección seguida del carácter L y una expresión (dirección L Expresión_O_Value) que diseña un recuento.

Si el recuento es un valor positivo, entonces la dirección de inicio será la especificada. dirección, y la dirección final está implícita y es igual a dirección + conteo:

```
0:000> db02c0000L5
```

```
002c0000 23 01 00 00 00
```

#....

Si el recuento es un valor negativo, la dirección final se convierte en la especificada. dirección, y la dirección de inicio se convierte en dirección - contar:

```
0:000> db02c0000L-5
```

```
002c0000 23 01 00 00 00
```

#....

De forma predeterminada, la expresión o el valor pasado después de L no puede superar los 256 MB. Esto es para evitar pasar valores muy grandes por accidente. Para sobrescribir esta limitación, use L? en lugar de solo L. Por ejemplo, observe cómo DbgEng se quejará de este gran tamaño:

```
0:000> db @$ip L0xffffffff
```

^ Error de rango en 'db @\$ip l0xffffffff'

Cuando se utiliza L?, DbgEng estará encantado de cumplir:

```
0:000> db @$ip L?0xffffffff
```

```
760039c2 83 e4 f8 83 ec 18 8b 4d-1c 8b c1 25 b7 7f 00 00 .....M...%....
```

...

Volcado de contenidos de la memoria

El comando d se utiliza para volcar el contenido de la memoria. La sintaxis general es la siguiente:

```
d[a|b|c|d|D|f|p|q|u|w|W] [Opciones] [Rango]
```

Se pueden utilizar varios formatos para visualizar el contenido de la memoria. El más común es Los formatos son los siguientes:

- b, w, d, q: para formatos de byte, palabra, palabra doble y palabra cuádruple, respectivamente
- f, D—Para valores de punto flotante de precisión simple y doble, respectivamente
- a, u—Para mostrar el contenido de la memoria ASCII o Unicode, respectivamente
- p—Para valores de puntero (el tamaño varía según el tamaño actual del puntero del destino)

Cuando dp, dd o dq tienen el sufijo s, se mostrarán los símbolos correspondientes a las direcciones. Esto puede resultar útil para descubrir punteros de función que están definidos en una matriz o una tabla virtual:

```
(1)
0:011> combinación de pb!CoCreateInstance
(2)
0:024>g
Punto de interrupción 0 alcanzado
;Combase! CoCreateInstance:
7526aeb08bff          esp=00000000      edi=00000000      edi=00000000
0:011> ? poi(esp+4*5)
Evaluar expresión: 112323728 = 06b1ec90
0:011> ? poi(poi(esp+4*5))
Evaluar expresión: 0 = 00000000
(3)
0:011> g poi(esp)
combase!CustomUnmarshalInterface+0x15d:
752743e7 fe8ef0000000      dic=00000000      byte ptr [esi+0F0h]
ds:002b:08664160=01
0:011> ? poi(06b1ec90)
Evaluar expresión: 141774136 = 08734d38
(4)
0:011> dps 08734d38 L1
08734d38 752c9688 combase!CErrorObject::`vftable'
0:011> dps 752c9688 L3
752c9688 752f6bd1 combase![thunk]:CErrorObject:QueryInterface`ajustador{8}'
752c968c 752f6bd0 !combase![thunk]:CErrorObject: AddRef ajustador{8}'
752c9690 752a9b91 combase![thunk]:CErrorObject: Release`ajustador{8}'
```

El marcador 1 agrega un punto de interrupción en la siguiente función:

```
HRESULT CoCreateInstancia(
REFCLSID rclsid,
```

```
LPUNKNOWN punk exterior,
DWORD dwClContexto,
REFIIDO riido,
LPVOID *ppv)
```

Nos interesa determinar el valor del puntero (parámetro 5) de la interfaz recién creada después de que la función retorna. En el marcador 2, reanudamos la ejecución . El programa luego se interrumpe en el punto de interrupción y se suspende. Luego inspeccionamos la ubicación del quinto puntero y lo desreferenciamos. Su valor desreferenciado debe ser NULL e inicializarse correctamente solo si la función retorna exitosamente. En el marcador 3, dejamos que el depurador ejecute la función CoCreateInstance y regrese al llamador. Luego desreferenciamos nuevamente el puntero de salida. Finalmente, en el marcador 4, usamos el comando dps para mostrar la dirección de la vftable y luego usamos dps una vez más para mostrar tres punteros en la vftable.

NOTA: Ips es equivalente a dds en objetivos de 32 bits y a dqs en objetivos de 64 bits.

Edición de contenidos de la memoria

Para editar el contenido de la memoria, utilice el comando e . La sintaxis general es la siguiente:

```
e[b|d|D][p|q|w] Dirección [Valores]
```

NOTA: Si no se especifica ningún sufijo después del comando e , se utilizará el último sufijo que se haya utilizado previamente con e . Por ejemplo, si se utilizó ed la primera vez, la próxima vez que se utilice e sola, actuará como si fuera ed.

Utilice los especificadores de formato b, w, d o q para establecer valores de byte, word, dword o qword, respectivamente, en la dirección de memoria especificada:

```
0:000> eb 0x1b0000 11 22 33 44; db 0x1b0000 L 4
001b0000 11 22 33 44
0:000> ed 0x1b0000 0xdeadbeef 0xdeadc0de; dd 0x1b0000 L 2
001b0000 carne muerta código muerto
```

Es posible utilizar comillas simples para introducir valores de caracteres cuando se utilizan los formatos w/d o q. DbgEng respetará el orden de bits del destino:

```
0:000> ed 1b0000 'ETIQUETA1'
0:000>db 1b0000 'ETIQUETA1' L 4
001b0000 31 47 41 54
1GAT
```

Además de editar la memoria con valores enteros, el comando e tiene otros especificadores de formato que le permiten ingresar otros tipos:

- e[f|D] (valores de dirección): establece un puntero flotante de precisión simple o doble. número:

```
0:000> eD @$10 1999.99
0:000> dD @$10 L 1
000000c9'e2450000          1999.99
```

- ep (valores de dirección): establece valores del tamaño de un puntero. Este comando sabe qué tan grande es un puntero en función del objetivo depurado actualmente.
- e[a|u] (cadena de dirección): ingresa una cadena ASCII o Unicode en la dirección dada. Dirección. La cadena ingresada no terminará en cero:

```
0:000> f 0x1b0000 L0x40 0x21 0x22 0x23; base de datos 0x1b0000 L0x20;
Bytes 0x40 llenos
001b0000 21 22 23 21 22 23 21 22-23 21 22 23 21 22 23 21
!"#!"#!"#!"#!
001b0010 22 23 21 22 23 21 22 23-21 22 23 21 22 23 21 22
"ii ...
0:000> ea 0x1b0000 "Hola mundo"; db 0x1b0000 L0x20
001b0000 48 65 6c 6c 6f 20 77 6f-72 6c 64 23 21 22 23 21 Hola
mundo#!"#
001b0010 22 23 21 22 23 21 22 23-21 22 23 21 22 23 21 22
"ii ...
```

- e[za|zu] (cadena de dirección): a diferencia de e[a|u], este comando ingresará la terminación de carácter cero al final de la cadena.

Para llenar un área de memoria con un patrón dado, utilice el comando f :

```
| Dirección L Contar valores
```

Por ejemplo:

```
0:000> f @eax L0x40 0x21 0x22 0x23; base de datos @eax L0x20
Bytes 0x40 llenos
001b0000 21 22 23 21 22 23 21 22-23 21 22 23 21 22 23 21 !"#!"#!"#!"#
001b0010 22 23 21 22 23 21 22 23-21 22 23 21 22 23 21 22 "ii ...
```

Comandos de memoria varios

A continuación se muestra otro conjunto de comandos relacionados con la memoria que resultan útiles:

- s [-[flags]type] Patrón de rango: busca en la memoria un valor determinado patrón
- c Range _For_Address1 Address2: compara dos regiones de memoria
- .dvalloc [Opciones] Tamaño: asigna memoria en el espacio de proceso de El depurador:


```
0:000> dvalloc 0x2000
Se asignaron 2000 bytes a partir de 001c0000
```
- .dvfree [Opciones] Tamaño de dirección base : libera la memoria asignada previamente por .dvalloc

208 Capítulo 4 ■ Depuración y automatización

- **.readmem Rango de nombres de archivo:** lee un archivo desde el disco al directorio del depurador.

memoria:

```
kd> .readmem archivo.bin @eax L3
Leyendo 3 bytes.
```

- **.writemem Rango de nombres de archivo:** escribe la memoria del depurado en un archivo en el disco

Símbolos

Los siguientes comandos le permiten inspeccionar símbolos y datos estructurados:

- **dt [tipo] [dirección]**—Un comando muy útil para mostrar el tipo de un elemento en la dirección dada:

```
## Muestra el tipo de la estructura UNICODE_STRING
0:000> CADENA UNICODE
ole32!CADENA UNICODE
+0x000 Longitud :Uint2B
+0x002 Longitud máxima :Uint2B
+0x004 Búfer : Ptr32 carácter de escritura

## Mostrar información de tipo y valores en un tipo en una dirección determinada
0:000> 0x18fef4
ntdll!_CADENA_UNICODE
"KERNEL32.DLL"
+0x000 Longitud :0x18
+0x002 Longitud máxima :0x1a
+0x004 Búfer : 0x00590168 "KERNEL32.DLL"
```

- **dv [banderas] [patrón]**—Muestra información sobre las variables locales
- **x [opciones] [patrón_de_módulo][[patrón_de_símbolo]]**—Muestra los símbolos en un módulo o módulos determinados
- **!dh [opciones] Dirección:** vuelca los encabezados de imágenes PE
- **!drvobj DriverObjectPtr [Flags]**—Muestra información sobre un objeto DRIVER_OBJECT .

- **!heap**—Muestra información del montón
- **!pool**: muestra información del grupo de kernel

Puntos de interrupción

En la arquitectura x86/x64, DbgEng admite dos tipos de puntos de interrupción:

- Puntos de interrupción de software: estos puntos de interrupción se crean guardando el byte en la dirección del punto de interrupción y luego reemplazándolo con un byte 0xCC (en

x64/x64). El depurador implementa la lógica subyacente para manejar la magia del punto de interrupción.

- Puntos de interrupción de hardware: también conocidos como puntos de interrupción de procesador o de datos, estos puntos de interrupción pueden estar presentes o no según el hardware que ejecute el destino. Tienen una cantidad limitada y se pueden configurar para que se activen al leer, escribir o ejecutar.

La sintaxis simple para crear un punto de interrupción de software es la siguiente:

```
bp Dirección ["Cadena de comandos"] bu  
Dirección "Cadena de comandos"  
bm SymbolPattern ["Cadena de comandos"]
```

NOTA Consulte la documentación del depurador para conocer la sintaxis completa de b* Comandos.

Para enumerar los puntos de interrupción, simplemente use el comando bl :

```
0:001>bl  
0 y 771175c9      0001 (0001) 0:**** ntdll!RtlInitString+0x9  
1 y 77117668      0001 (0001) 0:**** ntdll!RtlInitUnicodeString+0x38  
2 y 771176be      0001 (0001) 0:**** ntdll!_sin_default+0x26  
3 y 7711777e      0001 (0001) 0:**** ntdll!sqrt+0x2a  
4 y 771177c0      0001 (0001) 0:**** ntdll!sqrt+0x6a
```

Para desactivar los puntos de interrupción, utilice el comando bd . De manera similar, utilice el comando be para habilitar puntos de interrupción y el comando bc para borrar (eliminar) puntos de interrupción.

Puede especificar una serie de identificadores de puntos de interrupción para habilitarlos, deshabilitarlos o borrarlos:

```
ser 0 2 4
```

O un rango:

```
ser 1-3
```

O simplemente todos los puntos de interrupción:

```
ser *
```

Puntos de interrupción no resueltos

El comando bu crea un punto de interrupción cuya dirección aún es desconocida/no está resuelta o cuya dirección puede cambiar si pertenece a un módulo (que es compatible con ASLR) que se carga y descarga muchas veces en diferentes direcciones base.

El depurador intentará reevaluar la dirección del punto de interrupción cuando se cargue un nuevo módulo y, si el símbolo coincide, el punto de interrupción se activa.

Cuando se descarga el módulo, el punto de interrupción queda inactivo hasta que se pueda resolver el símbolo nuevamente.

En resumen, la dirección del punto de interrupción no es fija y se cambiará automáticamente. ajustado por el depurador.

Puntos de interrupción del software

Los puntos de interrupción de software se pueden crear mediante el comando `bp`. Si la dirección se puede resolver cuando se crea el punto de interrupción, este se activa.

Si no se puede resolver el punto de interrupción, este actuará como un punto de interrupción no resuelto y se activará una vez que se pueda resolver la dirección. Si el módulo en la dirección del punto de interrupción se descarga y luego se carga nuevamente, la dirección del punto de interrupción resuelto anteriormente permanecerá fija (a diferencia de los puntos de interrupción no resueltos).

Puntos de interrupción del hardware

Los puntos de interrupción de hardware se pueden crear utilizando el comando `ba`. Estos puntos de interrupción son asistidos por el hardware. Para crear un punto de interrupción de hardware, debe especificar la dirección, el tipo de acceso y el tamaño. El tipo de acceso designa si se debe interrumpir en lectura (lectura/escritura), escritura (solo escritura) o ejecución. El tamaño designa el tamaño del elemento al que se está interrumpiendo en el acceso. Por ejemplo, para interrumpir en "acceso de palabra", especifique el tamaño 2.

NOTA: Existe un límite arquitectónico en la cantidad de puntos de interrupción de hardware que puede tener.

Puntos de interrupción condicionales

Los puntos de interrupción condicionales pueden ser cualquier tipo de punto de interrupción descrito anteriormente. De hecho, cada punto de interrupción puede estar asociado a un comando. Cuando un comando condicional está asociado a un punto de interrupción, el punto de interrupción puede considerarse un punto de interrupción condicional.

El siguiente ejemplo crea un punto de interrupción condicional tal que cuando eax tiene el valor de 5, el punto de interrupción suspenderá la ejecución; de lo contrario, el punto de interrupción continuará reanudando la ejecución:

```
0:000> uf kernelbase!Obtener ultimo error
KERNELBASE!Obtener ultimo error:
7661d0d6 64a118000000           comentado      eax,dword ptr fs:[00000018h]
7661d0dc8b4034                 comentado      eax,dword ptr [eax+34h]
7661d0dfc3                     retirado

0:000> pb 7661d0df ".si @eax!=5 { gc; }"
0:000>bl
0 y 7661d0df 0001 (0001) 0:*** KERNELBASE!GetLastError+0x9 ".if @eax!=5 (gc;"
```

Es posible asociar una condición más elaborada a un punto de interrupción. Esto se explica en la sección “Creación de scripts con las herramientas de depuración”, más adelante en este capítulo.

Inspección de procesos y módulos

DbgEng le permite inspeccionar procesos en ejecución, módulos cargados/descargados o controladores de modo kernel cargados.

Para obtener la lista de módulos cargados y descargados, use lm:

```
0:001> lm n
comenzar      fin          nombre del módulo
00400000 00405000 imagen00400000
5ca40000 5cb44000 MFC42
733a0000 733b9000 mapa de dwmapi
73890000 73928000 ayuda de aplicación
...
```

De manera similar, en la depuración en modo kernel, el comando lm mostrará la lista de controladores de dispositivos cargados:

```
kd> lmn
comenzar      fin          nombre del módulo
804d7000 806cd280 nuevo                  ntkrnlpa.exe
806ce000 806ee380 hal                   halaacpi.dll
b205e000 b2081000 Archivo rápido Archivo rápido.SYS
b2121000 b2161380 HTTP                  HTTP.sys
b2d2b000 b2d4cd00 afd                  sistema afd
b2d4d000 b2d74c00 neto                 netbt.sys
b2d75000 b2dccca80 tcpip                tcpip.sys
bf800000 bf9c0380 win32k win32k.sys
f83e6000 f8472480 NTFS                 Ntfs.sys
f86ca000 f86d6c80 VolSnap VolSnap.sys
f8aaa000 f8aad000 BOOTVID BOOTVID.dll
...
```

NOTA: La opción n se pasó para minimizar la salida predeterminada del comando lm .

Para ver la información del módulo (versión, tamaño, base, etc.), utilice el interruptor v para modo verbose y m para especificar un nombre de módulo para que coincida:

```
kd> lm vm *volsnap*
comenzar      fin          nombre del módulo
f86ca000 f86d6c80 VolSnap
    Archivo de imagen de símbolo cargado: VolSnap.sys
    Ruta de la imagen: VolSnap.sys
    Nombre de la imagen: VolSnap.sys
    Marca de tiempo:           martes 03 agosto 2004 23:00:14 (41107B6E)
```

212 Capítulo 4 ■ Depuración y automatización

Suma de comprobación:	00017B61
Tamaño de la imagen:	0000CC80
Traducciones:	0000.04b0 0000.04e4 0409.04b0 0409.04e4

Cuando se está en modo kernel, se tiene una vista completa de todos los procesos en ejecución. Utilice el comando de extensión !process con los indicadores 0 0 para enumerar todos los procesos en ejecución:

```
kd> !proceso 0 0
**** VOLCADO DE PROCESO ACTIVO NT ****
PROCESO 823c8830 SessionId: ninguno Cid: 0004 Peb: 00000000 ParentCid: 0000
DirBase: 00334000 ObjectTable: e1000c90 HandleCount: 246.
Imagen: Sistema
PROCESO 820ed020 SessionId: ninguno Cid: 017c Peb: 7ffdd000 Cid. parental: 0004
DirBase: 07f40020 ObjectTable: e14f9c60 HandleCount: 21.
Imagen: smss.exe
PROCESO 81e98740 IdSesión: 0 Cid: 0278 Peb: 7ffde000 Cid. parental: 017c
DirBase: 07f40060 ObjectTable: e1010ac8 HandleCount: 517.
Imagen: winlogon.exe
PROCESO 81e865c0 Id. de sesión: 0 Cid: 02a4 Peb: 7ffde000 Cid. parental: 0278
DirBase: 07f40080 ObjectTable: e1a7a450 HandleCount: 265.
Imagen: services.exe
PROCESO 821139f0 IdSesión: 0 Cid: 0354 Peb: 7ffd9000 Cid. parental: 02a4
DirBase: 07f400e0 ObjectTable: e1a78ce0 HandleCount: 201.
Imagen: svchost.exe
PROCESO 81e68558 Id. de sesión: 0 Cid: 0678 Peb: 7ffdd000 Cid. parental: 0658
DirBase: 07f401e0 ObjectTable: e177aa70 HandleCount: 336.
Imagen: explorer.exe
```

NOTA: Esto es equivalente a utilizar el comando de extensión !for_each_process sin ningún parámetro.

Es posible establecer puntos de interrupción en procesos en modo usuario mediante el depurador de kernel. Primero, debe cambiar al contexto de proceso correcto y, para ello, necesita el valor EPROCESS :

```
kd> !proceso 0 0 explorer.exe
PROCESO 81e68558 Id. de sesión: 0 Cid: 0678 Peb: 7ffdd000 Cid. parental: 0658
DirBase: 07f401e0 ObjectTable: e177aa70 HandleCount: 336.
Imagen: explorer.exe
```

Luego utilice el comando .process /r /p EPROCESS para cambiar al contexto del proceso deseado:

```
kd> .process /r /p 81e68558
El proceso implícito ahora es 81e68558
.cache forcedecodeuser hecho
Cargando símbolos de usuario.....
```

En este punto, después del cambio de contexto, use !m no solo para enumerar los controladores del kernel cargados sino también los módulos del modo usuario.

El siguiente ejemplo establece un punto de interrupción en kernel32!CreateFileW para ese EPROCESS:

```
(1) kd> bp /p 81e68558 kernel32!CrearArchivoW  
(2)  
kd>bl  
0 y 7c810976 0001 (0001) kernel32!CrearArchivoW  
      Datos del proceso de coincidencia 81e68558  
(3)  
kd>g  
Punto de interrupción 0 alcanzado  
kernel32!CrearArchivoW:  
001b:7c8109768bff          movimiento           edito,edito  
(4)  
kd> .printf "%mu\n", poi(@esp+4);  
C:\Temp\desktop.ini
```

En el marcador 1, configuramos un filtro EPROCESS con el comando bp /p EPROCESS para que solo el proceso explore.exe active el punto de interrupción. El marcador 2 enumera los puntos de interrupción. Tenga en cuenta que solo coincidirá con un determinado EPROCESS. En el marcador 3, reanudamos la ejecución y esperamos hasta que se active el punto de interrupción. En el marcador 4, mostramos el nombre del archivo al que se accedió. El marcador 4 se verá mucho más claro después de leer la sección "Idioma" más adelante en este capítulo.

Ahora supongamos que desea mostrar todos los procesos que llamaron a CreateFileW API y visualización de a qué nombre de archivo se hizo referencia:

```
kd> bp kernel32>CreateFileW "lprocess @$proc 0;.printf "%mu\n",poi(@esp+4);gc;"
```

Esto se interrumpirá cada vez que cualquier proceso en modo usuario alcance el punto de interrupción, y luego el comando de punto de interrupción invocará !process con el EPROCESS actual. (establecido en el pseudo-registro predefinido \$proc) para mostrar la información de contexto del proceso actual, mostrar el nombre del archivo y finalmente reanudar la ejecución con gc.

NOTA !process @\$proc 0 es equivalente a !process -1 0.

Cuando se reanude la ejecución, verá esta salida redactada:

```
kd>g  
  
PROCESO 82067020 SessionId: 0 Cid: 0138          Peb: 7ffd000 Cid parental: 02a4  
DirBase: 07f40260 ObjectTable: e1b66ef8 HandleCount: 251.  
Imagen: vmtoolsd.exe  
  
C:\WINDOWS\SoftwareDistribution\DataStore\DataStore.edb  
PROCESO 81dc0da0 IdSesión: 0 Cid: 0204          Peb: 7ffd5000 Cid parental: 03fc
```

```
DirBase: 07f40280 ObjectTable: e1ba8ea8 HandleCount: 177.  
Imagen: wuauctl.exe  
  
PROCESO 81e68558 Id. de sesión: 0 Cid: 0678  
DirBase: 07f401e0 ObjectTable: e177aa70 HandleCount: 362.  
Imagen: explorer.exe  
  
C:\WINDOWS\media\Windows XP Start.wav  
PROCESO 81e68558 Id. de sesión: 0 Cid: 0678  
DirBase: 07f401e0 ObjectTable: e177aa70 HandleCount: 351.  
Imagen: explorer.exe  
  
C:\WINDOWS\WinSxS\Policy\Policy_6.0.Microsoft.Windows.Controles-comunes_6595b64144ccf1df_x-ww_5ddad775\6.0.2800.2180.Politica  
  
PROCESO 820f0020 IdSesión: 0 Cid: 0260  
DirBase: 07f40040 ObjectTable: e1503128 Contador de identificadores: 343.  
Imagen: csrss.exe
```

Comandos varios

Esta sección presenta varios comandos de depuración diversos, el .printf comando, junto con los especificadores de formato que admite, y describe cómo utilizar el lenguaje de marcado del depurador (DML) con .printf u otros comandos que admiten DML.

El comando .printf

El comando .printf es uno de los comandos más útiles para mostrar información de scripts o comandos. Al igual que en el lenguaje C, este comando acepta especificadores de formato. A continuación se presentan algunos de los más importantes:

- %p (valor de puntero): muestra un valor de puntero.
- %d, %x, %u (valor numérico): muestra valores enteros. La sintaxis es muy similar a los especificadores de formato de C.
- %ma / %mu (valor del puntero): muestra la cadena ASCII/Unicode en el puntero especificado.
- %msa / %msu (valor del puntero): muestra ANSI_STRING / UNICODE_STRING valor en el puntero especificado.
- %y (valor del puntero): muestra el nombre del símbolo (y el desplazamiento, si lo hay) en el puntero especificado.

He aquí un ejemplo sencillo:

```
0:000> .printf "%d %d eax=%x ebx=%d\n", @$t0, @$t1, @eax, @ebx
t0=0 t1=0 eax=5 ebx=8323228
```

No hay ningún especificador %s para expandir los argumentos de cadena. El siguiente ejemplo expande el valor del alias definido por el usuario incorporándolo en el parámetro de formato:

```
0:000> aS STR "ElValor"
0:000> al
      Alias          Valor
      ---          ---
      STR           El valor
0:000> .printf "Este valor de cadena es ${STR}\n"
```

El comando .printf puede utilizar el lenguaje de marcado del depurador (DML). Para utilizar DML con .printf, especifique el modificador /D.

NOTA: DML solo funciona en WinDbg.

Para mostrar cadenas con colores, utilice el marcado col :

```
0:000> .printf /D "<col fg=\"emphfg\">Hola</col> mundo\n"
Hola Mundo
```

También es posible utilizar las etiquetas u, i y b para subrayado, cursiva y negrita, respectivamente:

```
0:000> .printf /D "<u>subrayado</u> <b>negrita</b> <i>cursiva</i>\n";
subrayar negrita cursiva
```

Un marcado muy útil es el enlace porque hace que la salida sea clicable y esté asociada con un comando:

```
0:000> .printf /D "Haga clic <link cmd=\"u 0x401000\">aquí</link>\n"
haga clic aquí
```

Algunos comandos del depurador también utilizan el modificador /D. Por ejemplo, lm /D enumerará los módulos y se podrá hacer clic en cada uno de ellos. Cuando se hace clic en un módulo, se emitirá el comando !mvm modulename .

NOTA: Utilice el comando .prefer_dml 1 para alternar una configuración global que indica a los comandos que admiten DML que prefieran DML cuando corresponda.

216 Capítulo 4 ■ Depuración y automatización

Para obtener más información, consulte dml.doc en la distribución de herramientas de depuración.

Otros comandos

Antes de finalizar nuestra discusión sobre los comandos del depurador, enumeraremos algunos comandos más útiles:

- **#**: Busca un patrón de desmontaje.
- **!gle**: Devuelve el último código de error.
- **.logopen/.logfile/.logappend/.logclose**: comandos para administrar registros conversión de la salida de la ventana de comandos a archivos de texto.
- **.load**: carga una extensión del depurador.
- **.cls**: borra la ventana de salida del depurador. (Este comando no funciona en scripts porque no es parte del lenguaje de scripts DbgEng).
- **.effmach**: cambia o muestra el modo de procesador que utiliza el depurador . Resulta útil para depurar procesos WOW64. Este comando también es similar al comando de extensión **Iwow64exts.sw**.

Creación de scripts con herramientas de depuración

Esta sección ilustra características de scripting importantes en DbgEng que son útiles para automatizar tareas de ingeniería inversa y depuración.

Pseudo-registros

DbgEng admite pseudoregistros para almacenar determinados valores. Todos los pseudoregistros comienzan con el signo \$. Si se antepone el signo @ a un pseudoregistro o a un registro, se indica al intérprete que el identificador no es un símbolo, por lo que no se realizará una búsqueda exhaustiva de símbolos, que a veces es lenta.

Pseudo-registros predefinidos

En esta sección presentamos algunos pseudoregistros predefinidos útiles. Se pueden utilizar en expresiones o como parámetros para depurar comandos o scripts.

Tenga en cuenta que algunos pseudoregistros pueden estar definidos o no, dependiendo del objetivo depurado.

- **\$csp**: el puntero de la pila de llamadas actual. Esto es útil porque no necesita Tienes que adivinar si debes usar esp o rsp.
- **\$ip**: el puntero de instrucción actual. De manera similar, se puede utilizar un punto (.) denota el puntero de instrucción actual.

■ **\$retreg/\$retreg64:** los registros de retorno (normalmente eax, edx:eax o rax).

■ **\$p:** el primer valor que mostró el último comando d?:

```
0:000> dd @$ip L 1
012aa5e5 012ec188

0:000> ? @$p
Evaluando expresión: 19841416 = 012ec188

0:000> dw @$ip+2 L 1
012aa5e5c188

0:000> ? @$p
Evaluando expresión: 49544 = 0000c188

0:000> db @$ip+2 L 1
012aa5e588

0:000> ? @$p
Evaluando expresión: 136 = 00000088
```

■ **\$ra:** la dirección de retorno actual. Esto es equivalente a poi(@\$csp).

■ **\$sexentry:** la dirección del punto de entrada del primer ejecutable del proceso actual. Esto es muy útil cuando se depura un programa desde el principio porque DbgEng no se interrumpe en el punto de entrada sino en el núcleo.

■ **\$speb:** Bloque de entorno de proceso. Este pseudoregistro tiene el siguiente tipo: ntdll!_PEB *.

■ **\$proc:** la dirección EPROCESS* del proceso actual en modo kernel. En modo usuario equivale a \$peb.

■ **\$teb:** bloque de entorno de subprocesso del subprocesso actual. Tiene las siguientes características: tipo de ing: ntdll!_TEB*.

■ **\$thread—ETHREAD*** en modo kernel. En modo usuario es lo mismo que \$teb.

■ **\$pid:** el identificador del proceso actual.

■ **\$tid:** el identificador del hilo actual.

■ **\$ptrsize:** el tamaño del puntero desde el punto de vista del depurador. Si el sistema operativo del host es de 64 bits y está depurando un proceso de 32 bits, entonces \$ptr-size=4. En modo kernel, devuelve el tamaño del puntero de la máquina de destino.

■ **\$pagesize:** la cantidad de bytes por página de memoria (generalmente 4096).

■ **\$dbgtime:** la hora actual (según la computadora que ejecuta el depurador).

■ **\$bpNUM:** la dirección asociada con el número de punto de interrupción:

```
0:000> bl
0 y 012aa597          0001 (0001) 0:**** calc!WinMainCRTStartup+0xf
1 y 012aa5ab          0001 (0001) 0:**** calc!WinMainCRTStartup+0x23

0:000> ? @$bp0
Evaluando expresión: 19572119 = 012aa597

0:000> ? @$bp1
Evaluando expresión: 19572139 = 012aa5ab
```

■ \$exp: el valor de la última expresión evaluada:

```
0:000> r $t0 = 1 + 4
0:000> ? @$exp
Evaluar expresión: 5 = 00000005
```

0

```
0:000> ?
Evaluar expresión: 1637096 = 0018fae8
0:000> ? @$exp
Evaluar expresión: 1637096 = 0018fae8
```

El primer ejemplo asigna un valor a un pseudoregistro después de haberlo evaluado. Puedes ver cómo \$exp devuelve el último valor. Lo mismo sucede con el segundo ejemplo, que evalúa el valor del registro esp .

Pseudo-registros definidos por el usuario

Además de los pseudoregistros predefinidos, DbgEng permite a los usuarios definir su propio conjunto de pseudoregistros. DbgEng proporciona 20 pseudoregistros definidos por el usuario (UDPR) para su uso y para almacenar valores enteros. Son de \$t0 a \$t19.

El comando r se utiliza para asignar valores a esos registros:

```
0:000> r $t0 = 1234
0:000> ? @$t0
Evaluar expresión: 4660 = 00001234
```

Debido a que los números pueden ser punteros, es posible almacenar punteros tipificados en esos pseudo-registros que utilizan el comando r?:

```
(1)
0:000> ?poi(@$ip)
Evaluar expresión: 409491562 = 1868586a
(2)
0:000> r? $t0 = @@c++((largo sin signo *)@$ip)
(3)
0:000> ? @@c++(*@$t0)
Evaluar expresión: 409491562 = 1868586a
```

En el marcador 1, desreferenciamos y evaluamos el valor al que apunta \$ip. En el marcador 2, usamos r? para asignar una expresión de C++ a \$t0; el operador de conversión se usa para devolver un puntero tipificado (de tipo unsigned long *) a \$t0. Finalmente, en el marcador 3 usamos el operador de desreferenciación de C++ para desreferenciar \$t0. (Esto no hubiera sido posible sin tener un \$t0 tipificado previamente o sin preceder la expresión con una conversión).

He aquí otro ejemplo:

```
0:000> r? $t0 = @@c++(@$peb->ParámetrosDeProceso->NombreDeRutaDelImagen)
0:000> ? $t0
```

```
Evaluar expresión: 0 = 00000000
0:000> ?? @$t0
estructura _UNICODE_STRING
"c:\windows\syswow64\calc.exe"
+0x000 Longitud +0x002 :0x38
Longitud máxima +0x004 Búfer :0x3a
: 0x0098189e "c:\windows\syswow64\calc.exe"
```

Tenga en cuenta que cuando evalúa \$t0 con ?, obtiene cero. Cuando usa C++ sintaxis de evaluación ??, sin embargo, obtiene el valor escrito real.

También se pueden utilizar en las expresiones símbolos, todo tipo de pseudoregistros o alias.

Alias

Un alias es un mecanismo que permite crear una equivalencia entre un valor y un nombre simbólico. Al evaluar el alias, se obtiene el valor que se le asignó.

DbgEng admite tres tipos de alias:

- Alias nombrados por el usuario: como su nombre lo indica, estos alias son elegidos por El usuario.
- Alias de nombre fijo: hay diez de ellos, llamados \$u0 .. \$u9.
- Alias automáticos: son alias predefinidos que se expanden a ciertos valores.

Alias con nombre de usuario

Esta sección describe cómo crear y administrar alias definidos por el usuario y explica cómo se interpretan.

Creación y gestión de alias con nombre de usuario

Los siguientes comandos se utilizan para crear alias con nombre de usuario:

- **como AliasName Alias_Equivalence:** crea una equivalencia de línea para el alias dado:

```
como MiAlias lm;vertarget
```

Esto creará un alias para dos comandos: lm y luego vertarget. Puedes ejecutar ambos comandos invocando MyAlias.

- **aS AliasName Alias_Equivalence:** crea una equivalencia de frase para el alias indicado.

Esto significa que un punto y coma terminará la equivalencia del alias (a menos que la equivalencia estuviera entre comillas) y comenzará un nuevo comando.

```
como MiAlias lm;vertarget
como MyAlias "lm;vertarget"
```

La primera línea ejecutará dos cosas: creará un alias con el valor lm y luego ejecutará el comando vertarget . La segunda línea (porque la equivalencia está entre comillas) define el alias con el valor lm;vertarget.

NOTA: Los nombres de alias definidos por el usuario no pueden contener el carácter de espacio.

Otros comandos de alias incluyen los siguientes:

■ al—Enumera los alias ya definidos.

■ ad [/q] AliasName*—Elimina un alias por nombre o todos los alias. /q

El interruptor no mostrará mensajes de error si no se encuentra el nombre del alias.

El comando aS se puede utilizar para crear alias que equivalen a valores de variables de entorno, expresiones, contenidos de archivos, salida de comandos o incluso contenidos de cadenas de la memoria del depurado:

■ aS /f AliasName FileName: asigna el contenido de un archivo al alias:

```
0:000> aS /f NombreAlias c:\temp\lines.txt
0:000> al
      Alias          Valor
      -----
      AliasNombre    línea1
línea2
línea3
línea 4
línea 5
```

■ aS /x AliasName Expression64: asigna el valor de 64 bits de una expresión al alias. Esto resulta útil de muchas maneras, especialmente cuando se asigna el valor de un alias automático a un alias con nombre de usuario:

```
0:000> r $t0 = 0x123
0:000> como /x NombreAlias @$t0
0:000> al
      Alias          Valor
      -----
      AliasNombre    0x123
0:000> como alias incorrecto @$t0
0:000> al
      Alias          Valor
      -----
      AliasNombre    0x123
      Alias incorrecto @$t0
```

Tenga en cuenta que el primer uso de /x asignó correctamente el valor 0x123 al alias, mientras que la segunda asignación tomó el valor literal de @\$t0. (debido al interruptor /x faltante).

- como /e AliasName EnvVarName: establece el alias AliasName en el valor de la variable de entorno llamada EnvVarName:

```
0:000> como /e CmdPath COMSPEC
0:000> al
      Alias          Valor
      -----
      Ruta de comando    C:\Windows\system32\cmd.exe
```

- como /ma AliasName Dirección: establece el contenido de la dirección terminada en nulo. Cadena ASCII a la que apunta la dirección en el alias:

```
0:000> base de datos 0x40600C
0040600c 54 6f 6f 6c 62 61 72 57-69 6e 64 6f 77 33 32 00 Barra de herramientasWin-
abajo32.
0:000> como /ma Str1 0x40600C
0:000> al
      Alias          Valor
      -----
      Str1           Barra de herramientas Ventana32
```

- como /mu AliasName Dirección: establece el contenido de la dirección terminada en nulo. Cadena Unicode a la que apunta la dirección en el alias
- como /ms[al]u AliasName Dirección: establece el contenido de una cadena ASCII_STRING (estructura definida en el DDK) o UNICODE_STRING en el alias:

```
(1)
0:000> dt _CADENA_UNICODE
ntdll!_CADENA_UNICODE
+0x000 Longitud :UInt2B
Longitud máxima +0x004 Búfer :UInt2B
: Ptr32 UInt2B

(2)
0:000> ?? tamaño de (_UNICODE_STRING)
entero sin signo 8

(3)
0:000> ?? @@c+=(@$peb->Parámetros del proceso->RutaDLL)
estructura _UNICODE_STRING
"C:\Windows\system32\INV"
+0x000 Longitud :UInt2B
Longitud máxima +0x004 Búfer :UInt2B
: Ptr32 UInt2B
```

222 Capítulo 4 ■ Depuración y automatización

```
(4)
0:000> dd @@c++(&(@$peb->ParámetrosDeProceso->DllPath)) L2
001f1408 002e002c 001f1880

(5)
0:000>db001f1880L2e
001f1880 43 00 3a 00 5c 00 57 00-69 00 6e 00 64 00 6f 00
C...».Ventana
001f1890 77 00 73 00 5c 00 73 00-79 00 73 00 74 00 65 00
ws!.sistema
001f18a0 6d 00 33 00 32 00 5c 00-4e 00 56 00 00 00
m.3.2.I.NV..
```

(6)	
0:000> como /msu DllPath @@c++(&(@\$peb->ProcessParameters->DllPath))	
0:000> al	
Alias	Valor
-----	-----
Ruta de dll	C:\Windows\system32\NV

En el marcador 1, mostramos los campos de la estructura _UNICODE_STRING y en el marcador 2 mostramos el tamaño de la estructura utilizando el evaluador de C++. De manera similar, el marcador 3 utiliza la evaluación tipificada de C++ para volcar el valor del campo DllPath . El marcador 4 utiliza el operador & para volcar el contenido del campo _UNICODE_STRING y el marcador 5 volca la dirección del búfer . Por último, el marcador 6 utiliza el comando as para crear un alias con su contenido leído desde un puntero _UNICODE_STRING .

Interpretación de alias con nombre de usuario

Los alias con nombre de usuario se pueden interpretar utilizando la sintaxis básica \${AliasName} o simplemente escribiendo el nombre del alias. La primera opción se debe utilizar cuando el alias está incrustado en una cadena y no está rodeado por caracteres de espacio:

```
0:000> aS AliasName "Valor de alias"
0:000> .printf "El valor es >${AliasName}<\n"
El valor es >Valor de alias<
```

Cuando no se define un alias, la sintaxis de evaluación de alias permanece sin evaluar:

```
0:000> .printf "El valor es >${UnkAliasName}<\n"
El valor es >${UnkAliasName}<
```

Los siguientes parámetros controlan cómo se interpretan los alias:

- \${/d:AliasName}: se evalúa como 1 si el alias está definido y como 0 si no lo está. Este modificador resulta útil cuando se utiliza en un script para determinar si un alias está definido o no:

```
0:000> .printf ">${/d:NombreAlias}<\n"
>1<
```

```
0:000> .printf ">${/d:NombreDeAliasDesconocido}<\n"
>0<
```

- \${/f:AliasName}): cuando se utiliza este modificador, un alias no definido se evaluará como una cadena vacía o como el valor real si el alias fue definido:

```
0:000> .printf ">${/f:NombreDeAliasDefinido}<\n"
>Valor de alias<
0:000> .printf ">${/f:NombreAliasIndefinido}<\n"
><
```

- \${/n:AliasName}): se evalúa como el nombre del alias o permanece sin evaluar Si el alias no está definido:

```
0:000> .printf ">${/n:NombreAlias}<\n"
>AliasNombre<
0:000> .printf ">${/n:AliasName2}<\n"
>${/n:AliasName2}<
0:000> .printf ">${/n:NombreAliasDesconocido}<\n"
>${/n:NombreAliasDesconocido}<
```

- \${/v:AliasName}): este modificador evita cualquier evaluación de alias:

```
0:000> .printf ">${/v:NombreAlias}<\n"
>${/v:NombreAlias}<
0:000> .printf ">${/v:NombreAliasDesconocido}<\n"
>${/v:NombreAliasDesconocido}<
```

Una vez definido un alias, se puede utilizar en cualquier comando posterior (como comando o como parámetro de un comando):

0:000> como mi_printf .printf	
0:000> al	
Alias	Valor
-----	-----
mi_printf	.imprimir

Cuando se utiliza como comando:

```
0:000> ${my_printf} "Hola mundo\n"
Hola Mundo
0:000> my_printf "Hola mundo\n"
Hola Mundo
```

Cuando se utiliza como parámetro de un comando:

```
0:000> .printf "El comando para mostrar cadenas es >${my_printf}<\n"
El comando para mostrar cadenas es >.printf
```

224 Capítulo 4 ■ Depuración y automatización

```
0:000> .printf "El comando para mostrar cadenas es my_printf \n"
El comando para mostrar cadenas es printf
```

Al reasignar valores a alias definidos por el usuario, tenga en cuenta lo siguiente:

- El uso del comando aS de la siguiente manera produce un error:

```
0:000> aS MiVar 0n123;.printf "v=%d", ${MiVar}
v=No se pudo resolver el error en '${MyVar}'
```

El motivo de este error es que los alias se expanden solo en los nuevos bloques. Esto se puede solucionar con lo siguiente:

```
0:000> aS MiVar 0n123;.bloque { .printf "v=%d", ${MiVar}; }
v=123
```

- El modificador /v: se comporta como el modificador /n: cuando se utiliza con aS, as y ad. El motivo por el que lo mencionamos se ilustra en el siguiente ejemplo:

```
0:000> aS MiVar 0n123;.bloque { aS /x MiVar ${MiVar}+1 }
0:000> al
Alias          Valor
-----
0n123          0x7c
MiVariable     0n123
```

El primer comando crea el alias MyVar e incrementa su valor en uno; sin embargo, se crea un nuevo alias llamado 0n123 . Esto se debe a que el alias MyVar ha sido reemplazado por su equivalente en lugar de usarse como un nombre de alias.

Lo que debe hacer es indicarle al comando aS que MyVar es el nombre del alias y que su valor no debe expandirse ni evaluarse. Aquí es donde debe usarse el modificador /v: cuando se utiliza con el comando as o aS :

```
0:000> aS MiVar 123;.bloque { aS /x ${/v:MiVar} ${MiVar}+1 };al
Alias          Valor
-----
MiVariable     0x124
```

Tenga en cuenta que ahora, cuando \${/v:MyVar} se usa junto con aS, se evalúa como el nombre del alias (como lo haría \${/n:AliasName}).

Alias de nombre fijo

Como se mencionó anteriormente, hay 10 alias de nombre fijo denominados \$u0 a \$u9.

Si bien los alias con nombre fijo parecen registros o pseudoregistros, no lo son. Para asignarles valores, utilice el comando r seguido de \$. y el nombre del alias, de la siguiente manera:

```
(1)
0:000> r $.u0 = .printf
(2)
0:000> r $.u1 = 0x123
(3)
0:000> r $.u2 = Hola mundo
(4)
0:000> $u0 "$u2\n"
Hola Mundo
(5)
0:000> $u0 "$u2, u1=%x", $u1
Hola mundo, u1=123
```

El marcador 1 asigna el alias \$u0 al comando .printf . Observe el prefijo \$. y que el comando .printf no está entre comillas en la equivalencia.

El marcador 2 define \$u1 con un valor numérico y el marcador 3 define \$u2 con un valor de cadena. El marcador 4 utiliza \$u0 como equivalente al comando .printf e imprime \$u2, que está entre comillas y se resuelve como “Hola mundo”. Por último, el marcador 5 imprime el valor \$u1 de forma similar al marcador 4.

NOTA Utilice siempre \$. al definir el alias; sin embargo, al utilizar el alias no necesita utilizar \$. o incluso el signo @ como lo hace para los pseudo-registros o alias.

El reemplazo de alias con nombres fijos tiene mayor precedencia que los alias con nombre de usuario.

Alias automáticos

DbgEng define algunos alias cuando se inicia la sesión de depuración. Los alias automáticos son similares a los pseudoregistros predefinidos, excepto que también se pueden usar con la sintaxis \${} (como los alias con nombre de usuario).

Se definen los siguientes registros:

- \$ntnsym
- \$ntwsym
- \$ntsym
- \$CurrentDumpFile

- \$CurrentDumpPath
- \$Archivo de volcado actual
- \$CurrentDumpArchivePath

Para ilustrar esto, lo siguiente invoca el depurador de línea de comandos cdb con el modificador -z para abrir un archivo de volcado de memoria y utiliza -cf script.wds para ejecutar una serie de comandos desde un archivo de texto:

```
c:\Herramientas\dbg>cdb -cf av.wds -zm\xp_kmem.dmp
```

El contenido del archivo de script es el siguiente:

```
.printf "Script iniciado\n" .logopen @"$  
{$CurrentDumpFile}.log" !analyze -v .logclose .printf  
"Script finalizado,  
saliendo\n"  
  
q
```

Cuando se inicia el depurador, interpretará cada línea en av.wds: 1. Imprimirá un mensaje de inicio.

2. Abra un archivo de registro que tenga el nombre del archivo de volcado de memoria actual con la extensión .log adjunta. Observe cómo se expande a alias automático con la sintaxis \${}.
3. Emite el comando !analyze -v .
4. Cierre el archivo de registro, imprima un mensaje de salida y salga del depurador con la tecla q.

NOTA El signo @ se utiliza para definir una cadena literal (o sin formato). Consulte la próxima sección “Caracteres y cadenas”.

Idioma

En esta sección, analizamos el lenguaje de scripting, los tokens y los comandos.

Comentarios

Utilice el comando \$\$ para especificar comentarios. Por ejemplo:

```
$$ Esto es un comentario  
$$ Este es otro comentario
```

Para utilizar más de un comentario en una línea con múltiples declaraciones, utilice el Carácter de punto y coma para terminar el comentario:

```
r eax = 0; $$ borrar EAX; r ebx = ebx + 1; $$ incrementar EBX;
```

El asterisco (*) también se puede utilizar para crear comentarios; sin embargo, la línea completa después del asterisco se ignorará incluso si se utiliza un delimitador de punto y coma:

```
r eax = 0; * borrar EAX; r ebx = ebx + 1;
```

El comando anterior solo borrará EAX; no incrementará EBX en uno.

Hay una ligera diferencia entre el especificador de comentario \$\$ y .echo comando. El comando .echo muestra la línea en lugar de simplemente ignorarla.

Caracteres y cadenas

Los caracteres se especifican cuando se incluyen entre comillas simples:

```
0:000> @dvalloc 1  
0:000> eb @$10 'a' 'b' 'c' 'd' 'f' 'g'  
0:000> db @$10 L 6  
02250000 61 62 63 64 66 67
```

Las cadenas se especifican con comillas dobles:

```
0:000> ea @$10 "ingeniería inversa práctica";  
0:000> db @$10 L20 02250000 50  
72 61 63 74 69 63 61-6c 20 72 65 76 65 72 73 Reversiones prácticas  
02250010 65 20 65 6e 67 69 6e 65-65 72 69 6e 67 00 00 00 e ingeniería...
```

Al igual que en C, la cadena puede contener secuencias de escape; por lo tanto, es necesario escapar la secuencia para obtener el resultado correcto:

```
(1)  
0:000> _printf "c:\\herramientas\\dbg\\windbg.exe\n"  
c\\herramientas\\dbg\\windbg.exe  
(2)  
0:000> _printf "a\tb\tc\n1\t2\t3\n"  
a           b           c  
1           2           3
```

El primer comando escapó de la barra invertida con el carácter de escape. El segundo ejemplo utiliza la secuencia de escape de tabulación horizontal (\t).

DbgEng permite el uso de cadenas sin formato; dichas cadenas se interpretarán de forma literal sin tener en cuenta la secuencia de escape. Para especificar una cadena literal, anteponga la cadena con el signo arroba (@):

228 Capítulo 4 ■ Depuración y automatización

```
(1)
0:000> .printf @"c:\herramientas\dbg\windbg.exe\n";.printf "\n";
c:\herramientas\dbg\windbg.exe\n
(2)
0:000> .printf @"a\tb\tc\n1\t2\t3\n"
a_tb_tc_n1_t2_t3_n
```

Observe cómo las secuencias de escape se mantuvieron como se especificó sin ser interpretadas . De manera similar, si tiene un alias con nombre de usuario que se creó a partir de los contenidos de la memoria y desea evaluarlo literalmente, también anteponga el \${} con @:

```
(1)
0:000> como un S/mu STR 0x3cba030
0:000> al
Alias           Valor
-----
STR             C:\Temp\archivo.txt
(2)
0:000> .printf "${STR}\n";
C:\Archivo temporal.txt
(3)
0:000> .printf @"${STR}";.printf "\n";
C:\Temp\archivo.txt
```

El marcador 1 crea un alias con nombre de usuario a partir de la cadena Unicode terminada en cero en la dirección de memoria especificada y muestra la lista de alias. El marcador 2 imprime el valor del alias. (Observe que la salida no es la deseada). En el marcador 3, después de anteponer @ a la cadena, la salida es correcta.

Bloques

Se puede crear un bloque mediante el comando .block seguido de llaves de apertura y cierre ({}):

```
.bloquear
{
    $$Dentro de un bloque...
.bloquear
{
    $$Bloque anidado...
}
```

Cuando se crea un alias con nombre de usuario en un script, su valor no se evaluará. interpretarse como se pretende a menos que se cree un nuevo bloque:

```
como MiAlias (@eax + @edx)
.bloquear
{
    $$Dentro de un bloque...
```

```
.printf "El valor de mi alias es %X\n", ${MyAlias}  
}
```

Declaraciones condicionales

Los tokens de comando .if, .elsif y .else se utilizan para escribir condicionales declaraciones.

El uso de .if y .elsif es similar al de otros lenguajes en los que se acepta una condición. La condición puede ser cualquier expresión que evalúe cero (se trata como falsa) o un valor distinto de cero (se trata como verdadero):

```
r $t0 = 3;  
.si (@$t0==1) {  
  
.printf "uno\n";  
  
}.elsif @$t0==2 {  
  
.printf "dos\n";  
  
}.elsif (@$t0==3) {  
  
.printf "tres\n";  
}  
.demás  
{  
.printf "desconocido\n";  
}
```

NOTA: El uso de paréntesis alrededor de la condición es opcional.

Todas las estructuras de repetición integradas y las declaraciones condicionales requieren el uso de llaves ({ y }) y, por lo tanto, crean un bloque, lo que da como resultado la evaluación adecuada de los alias:

```
como MiAlias (@eax + @edx).si (1) {  
  
$$ Dentro de un bloque... .printf "El  
valor de mi alias es %X\n", ${MyAlias}  
}
```

También puedes comparar cadenas con .if usando algunos métodos diferentes:

```
$$ Al encerrar las cadenas a comparar entre comillas simples: .if '${my_alias}'=='value' {  
  
.printf "igual\n";  
}
```

230 Capítulo 4 ■ Depuración y automatización

```
.demás
{
    .printf "no es igual\n";
}

$$ Al utilizar el operador MASM scmp (o sicmp):
.if $scmp("${mi_alias}", "valor")
{
    .printf "igual\n";
}

$$ Al utilizar el operador MASM spat:
.if $spat("${mi_alias}", "valor")
{
    .printf "igual\n";
}
.demás
{
    .printf "no es igual\n";
}
```

DbgEng también proporciona el comando `j` , que puede compararse con el operador ternario de C (`cond ? true-expr: false-expr`), excepto que ejecuta comandos en lugar de devolver expresiones:

`j Expresión ["Comando-Verdadero"] ; ["Comando-Falso"]`

El siguiente es un ejemplo muy simple con un comando que se ejecuta en ambos casos (verdadero o falso):

```
0:000> r $t0 = -1
0:000> j (@$t0 < 0) r $t0 = @$t0-1 ; r $t0 = @$t0+1
0:000> ? $t0
Evaluar expresión: -2 = ffffffe
```

Las comillas simples son opcionales en la mayoría de los casos; especifíquelas si hay más de una. El comando a ejecutar es:

```
0:000> r $t0 = 2
0:000> j (@$t0 < 0) 'r $t0 = @$t0-1;echo Valor negativo';
               'r $t0 = @$t0+1;echo Valor positivo'
Valor positivo
0:000> ? $t0
Evaluar expresión: 3 = 00000003
```

Es común utilizar el comando `j` como parte de los comandos de punto de interrupción para formar puntos de interrupción condicionales.

El siguiente ejemplo suspende el depurador (tenga en cuenta las comillas simples vacías que especifican que no se debe ejecutar ningún comando cuando la expresión se evalúa como Verdadero) solo cuando la dirección de retorno coincide con un valor determinado:

```
0:000> bp usuario32!MessageBoxA "j" (@$ra=0x401058) ",'gc';"  
0:000>g  
usuario32!Cuadro de mensajeA:  
756e22c2 8bf  
0:000> ? $ra  
Evaluar expresión: 4198488 = 00401058
```

El siguiente ejemplo suspende el depurador cada vez que se llama a la función GetLastError y devuelve ACCESS_DENIED (valor 5):

```
0:014> bp kernelbase!GetLastError "g @$ra;j @eax==5 ",'gc"  
0:014>g  
Tema uxtheme! TemaPreWndProc+0xd8:  
00007ff8'484915e8 33c9 xor ecx,ecx  
0:000> !gle  
LastErrorValue: (Win32) 0x5 (5) - Acceso denegado.  
LastStatusValue: (NTSTATUS) 0xc0000034 - Nombre del objeto no encontrado.
```

Esta no es la forma óptima de lograrlo. Los símbolos públicos de NTDLL, cuando se cargan, exponen un símbolo llamado `g_dwLastErrorToBreakOn`. La mejor estrategia es editar este valor en la memoria y pasar el valor de error deseado para interrumpir:

```
0:000> ep ntdll!g_dwLastErrorToBreakOn 5  
0:000>g  
(2a0.2228): Excepción de instrucción de interrupción: código 80000003 (primera oportunidad)  
ntdll!RtlSetLastError+0x21:  
00007ff8'4c444df1cc entero 3  
0:000> !gle  
LastErrorValue: (Win32) 0 (0) - La operación se completó correctamente.  
LastStatusValue: (NTSTATUS) 0xc0000034 - Nombre del objeto no encontrado.
```

Errores de script

Si se produce un error durante la ejecución de un script de depuración, se cancelará todo el script después de que aparezca el mensaje de error. Considere un archivo de script con el siguiente contenido:

```
.printf "Script iniciado\n";  
comando inválido;  
.printf "Fin del script\n";
```

Cuando se ejecuta este script, producirá un error:

```
Guion iniciado  
Guion iniciado  
  
^ Error de sintaxis en '.printf "Script iniciado'  
,  
0:000>
```

Para evitar que el script se cancele, puede utilizar el token de comando .catch :

```
.printf "Script iniciado\n";
.atrapar
{
    comando no válido; .printf
    "!! no será alcanzado !!\n";

} .printf "Después de la captura\n";
```

El error hará que el script salga del bloque .catch y muestre el error, pero continuará ejecutando el script después de ese bloque:

Guion iniciado ^ Error de sintaxis en ': comando inválido; '
Después de la captura

Cuando se está dentro de un bloque .catch , se puede salir explícitamente de él con el token de comando .leave .

Curiosamente, .leave se puede utilizar para emular una “interrupción”, como en un bucle:

```
r $t0=0;
.atrapar
{
    .si (por(@$ip) == 0xb9) {
        .printf "se encontró MOV ECX, ... \n"; r $t0 =
        dwo(@$ip+1); .leave;

    } .elseif (por(@$ip) == 0xb8) {
        .printf "se encontró MOV EAX, ... \n"; r $t0 =
        dwo(@$ip+1); .leave;

    } $$ hacer algún otro análisis... .printf "No se pudo
    encontrar el código de operación correcto\n"; $$ hacer más cosas...

}
```

\$\$ Se alcanza después de que finaliza el bloque catch, se ha producido un error o se utiliza un .leave

Estructuras de repetición

DbgEng admite cuatro estructuras de repetición, que se describen en las siguientes secciones.

El comando .break se puede utilizar para salir de un bucle. De manera similar, el comando .continue se puede utilizar para pasar a la siguiente iteración dentro de la estructura de repetición encapsulante.

NOTA En el caso de una condición de repetición errónea (el script o comando se ejecuta sin fin), puede interrumpirlo presionando Ctrl+C en cualquiera de los depuradores de la consola (kd, cdb, ntsd) o Ctrl+PauseBreak en WinDbg.

El bucle for

El token de comando .for tiene la siguiente sintaxis:

```
.for (Comandoinicial; Condición; ComandosIncrementales) { Comandos }
```

El siguiente script de ejemplo vuelca los controladores de la tabla de descriptores de interrupciones (IDT) mediante un bucle for . Primero, ejecutamos el comando dt para inspeccionar la estructura de un IDTENTRY en un sistema de 32 bits en una sesión de depuración en modo kernel:

```
kd> dt _ENTRADA KIDT
jntdll!_KIDTENTRY
+0x000 Desplazamiento :Uint2B
Selector +0x002 :Uint2B
+0x004 Acceso :Uint2B
+0x006 Desplazamiento extendido: Uint2B
```

El guión es el siguiente:

```
para (r $t0=0; 1;r $t0=@$t0+1)
{
    $$ Lleva un puntero tipado a la siguiente entrada IDT
    r? $t1 = @@@c+(((_KIDTENTRY *)@idtr) + @$t0);

    $$ ¿Última entrada?
    .si (@@@c+(@$t1->Selector) == 0)
    {
        $$ Salir corriendo
        .romper;
    }

    $$ Resolver la dirección completa
    r $t2 = @@@c+((largo)((largo sin signo)@$t1->ExtendedOffset << 0x10) + (largo sin signo)@$t1->Offset);

    .printf "IDT[%02x] @ %p\n", @$t0, @$t2
    $$ .printf "IDT[%02x] @ %p\n", @$t2
}
```

Algunos aspectos importantes del guión a tener en cuenta:

- La condición del bucle for se establece en 1, por lo que se repite indefinidamente. Saldremos condicionalmente del interior del cuerpo del bucle con el comando .break .
- La r? se utiliza para asignar un valor tipificado a \$t1.
- El pseudo-registro \$t1 es un puntero a _KIDTENTRY. Cuando se le agrega \$t0 , este avanzará a la ubicación de memoria apropiada (teniendo en cuenta el tamaño de _KIDTENTRY).
- Usted determina el final de las entradas IDT examinando el campo Selector y salir del bucle en consecuencia.
- La dirección base completa del controlador IDT se calcula combinando los campos ExtendedOffset y Offset .
- Convierte \$t2 en long para que tenga el signo extendido correctamente (como pseudo-registros son siempre valores de 64 bits).
- Mostrar el resultado.

Si considera que el uso de pseudoregistros como \$t0 como un contador de bucle for es un poco inusual y, en su lugar, desea utilizar un nombre como i, j o k, por ejemplo, cree un alias con nombre de usuario llamado i que sea equivalente a @\$t0:

```
como yo @$t0;

.bloquear
{
    .para (r ${i} = 1; ${i} <= 5; r ${i} = ${i}+1)
    {
        .printf "i=%d\n", ${i}
    }
}
```

El bucle while

El bucle while es una forma simplificada de un bucle for que no tiene ni un comando inicial ni un comando de incremento:

```
.while (Condición) { Comandos }
```

Según la expresión de la condición, es posible que el cuerpo del bucle while no se ejecute en absoluto. A continuación, se muestra un script de ejemplo que rastrea 200 instrucciones en un proceso recién iniciado:

```
$$ Ir al punto de entrada (omitar la inicialización del proceso NT)
.printf "Yendo al punto de entrada\n";
```

```
g@Sexentry:  
  
.printf "Se inició el rastreo...\n";  
  
$$ Reiniciar el contador  
r $t0 = 0;  
  
.mientras (@$t0 <= 0x200) {  
  
.printf "ip -> %p; ntrace=%d\n", @$ip, @$t0; r $t0 = @$t0 + 1;  
  
es;  
}  
  
.printf "Condición satisfecha\n"; u @$ip L1;
```

Tenga en cuenta que esta no es la forma ideal de realizar un seguimiento condicional. Los comandos t y j utilizados juntos son un mejor enfoque.

El bucle do-while

El bucle do-while tiene la siguiente sintaxis:

```
.do { Comandos } (Condición)
```

A diferencia del bucle while , el cuerpo del bucle do se ejecutará al menos una vez antes del bucle. La condición se evalúa:

```
hacer  
{  
.si (por(@$ip) == 0xb8) {  
  
.printf "Se encontró MOV EAX, ...\\n"; .break;  
  
} $$ hacer otras cosas $$ ....  
$$....  
}  
(0); .printf "Continuar haciendo otra cosa...\\n";
```

DbgEng también proporciona el comando z para ejecutar comandos mientras se cumple una determinada condición:

```
Comando [ Comando ; [Comando ....] ] ; z(Expresión )
```

236 Capítulo 4 ■ Depuración y automatización

En el siguiente ejemplo, \$t0 se utiliza como contador para rastrear cinco (5) ramificaciones. instrucciones:

```
0:000> r $t0=1
0:000> th;r $t0=@$t0 + 1; z (@$t0 <= 5);
rehacer [1] th;r $t0=@$t0 + 1; z (@$t0 <= 5);
rehacer [1] th;r $t0=@$t0 + 1; z (@$t0 <= 5);
rehacer [1] th;r $t0=@$t0 + 1; z (@$t0 <= 5);
rehacer [1] th;r $t0=@$t0 + 1; z (@$t0 <= 5);
0:000> ? @$t0
Evaluando expresión: 6 = 00000006
```

Como en el ejemplo anterior, se pueden especificar uno o más comandos a la izquierda del comando z .

El bucle foreach

El bucle foreach es muy útil y se puede utilizar para enumerar tokens leídos desde un archivo, desde la salida de un comando o desde una cadena proporcionada por el usuario.

Se pueden pasar dos opciones comunes (por separado o juntas) como primeros parámetros al token del comando .foreach :

- /pS ExpressionValue: número inicial de tokens que se omitirán cuando comience el bucle. Esto es equivalente a inicializar el contador con un valor distinto de cero en un bucle for .
- /ps ExpressionValue: la cantidad de tokens que se omitirán después de cada iteración. Esto es equivalente a la parte de incremento del bucle for donde el programador puede especificar el valor de incremento del contador.

Tokenización a partir de una cadena

La sintaxis general es la siguiente:

```
.foreach [Opciones] /s (TokenVariableName "InString" ) { Comandos de salida }
```

Por ejemplo, supongamos que busca símbolos relacionados con *CreateFile* en los siguientes tres módulos: ntdll, kernelbase y kernel32. Esta es una forma de hacerlo:

```
.foreach /s (token "ntdll kernel32 kernelbase") { x ${token}!CreateArchivo; }
```

En el siguiente ejemplo, supongamos que desea tokenizar el contenido de un determinado Cadena ASCII en memoria:

```
como /mu STR 0x8905e8
r $t0 = 0;
.bloquear
{
.foreach /s (token "$(STR)") {
```

```
    .printf "token_{i}=%d, token_val=${token}\n", @$t0;
    $t0 = @$t0 + 1;
}
}
```

El \${} se utiliza para evaluar el valor de la variable de token. Esto solo es necesario si el token no está rodeado por el carácter de espacio en el momento de la evaluación. acción. El .block se utilizó para hacer que se evaluará el alias STR.

Tokenización a partir de la salida de un comando

La sintaxis general es la siguiente:

```
.foreach [Opciones] ( Variable { InCommands } ) { Comandos de salida }
```

Este uso de .foreach es el más común porque permite extraer información. información a partir de la salida de un comando y su uso en su script.

A modo de demostración, imagine un script que necesita asignar memoria en el espacio de proceso del depurado y luego usa esa memoria para leer un archivo. contenido al mismo.

Primero, examine la salida del comando de asignación de memoria .dvalloc:

```
0:000>.dvalloc 0n4096
Se asignaron 1000 bytes a partir de 00620000
```

La salida se puede convertir en seis tokens; por lo tanto, el bucle foreach debe usar el indicador /pS para omitir los primeros cinco tokens y comenzar directamente con el último token (que es la dirección de memoria recientemente asignada):

```
0:000>.foreach /pS 5 (token {.dvalloc 0x1000}) { r $t0 = ${token}; .break; }
0:000>? @$t0
Evaluar expresión: 8323072 = 007f0000
```

El guión completo queda así:

```
## Establecer el nombre del archivo de imagen
como nombre de archivo @"c:\temp\shellcode.bin"
.atrapar
{
    ## Establezca el tamaño de asignación para que sea igual al archivo que queremos leer
    r $t0 = 0n880;

    foreach /pS 5 (token {.dvalloc @$t0}) {

        r $t1 = token;
        .romper;
    }

    ## Leer el archivo
    .readmem "$nombreArchivo" @$t1 L@$t0;
```

```
.printf "Se cargó ${fileName} @ %pin", @$t1  
}
```

NOTA Recuerde liberar la memoria con el comando .dvfree .

El siguiente ejemplo analiza la salida de lm1m (que, por diseño, devuelve un resultado simple). salida fija para usar con .foreach):

```
0:000>lm1m  
Imagen00400000  
ESCUELA  
NUCLEO32  
comctl32  
usuario32  
Ntdll
```

El bucle foreach debería verse así:

```
0:000> .foreach (modulename { lm1m; }) { .printf "Nombre del módulo: modulename \n"; }
```

Tokenización desde un archivo

La sintaxis general es la siguiente:

```
.foreach [Opciones] /f ( Variable "InFile" ) { Comandos de salida }
```

Supongamos que existe un archivo llamado lines.txt con el siguiente contenido:

```
Esta es la linea 1  
Esta es la linea 2  
Esta es la linea 3
```

Se tokenizará de la siguiente manera:

```
0:000> .foreach /f (linea "c:\temp\lines.txt") { .printf ">${linea}<\n" }  
>Esto<  
>es<  
>línea<  
>1<  
>Esto<  
>es<  
>línea<  
>2<  
>Esto<  
>es<  
>línea<  
>3<
```

Bucles foreach proporcionados por la extensión

Existen algunos otros comandos foreach proporcionados por extensiones que no forman parte del lenguaje de programación. Estos comandos foreach se implementan dentro de varias extensiones de DbgEng:

- **!for_each_frame**—Ejecuta un comando para cada cuadro en la pila de El hilo actual
- **!for_each_function**—Ejecuta un comando para cada función en una función determinada. módulo que coincide con el patrón de búsqueda
- **!for_each_local**—Ejecuta un comando para cada variable local en la marco actual
- **!for_each_module**—Ejecuta un comando para cada módulo cargado
- **!for_each_process**—Ejecuta un comando para cada proceso (esta extensión sion funciona únicamente en la depuración del kernel)
- **!for_each_thread**—Ejecuta un comando para cada hilo (depuración del núcleo). (solo ging)

NOTA Utilice el comando .extmatch *for_each* para enumerar todos los foreach comandos de extensión.

Cada uno de esos comandos de extensión expone variables especiales al comando que ejecutan. Consulte el manual del depurador para saber qué variables se exponen para cada comando de extensión específico.

El siguiente ejemplo enumera todos los módulos y muestra información sobre ellos:

```
!for_each_module .printf /D "%16p %16p: ${(@#ModuleName)} @<link cmd="u %p">%p</link>\n", ${(@#Base)}, ${(@#End)}, $imento(0x${(@#Base)}), $imento(0x${(@#Base)})\n\n400000          408000: imagen00400000 @00406800\n74b70000        74c78000: gdi32 @74b7afc5\n75130000        75270000: KERNEL32 @7514a5cf\n755d0000        75656000: comctl32 @755d1e15\n75670000        757bf000: usuano32 @75685422\n759b0000        76b28000: shell32 @759b108d\n76d00000        76cbe000: msrvct@76d0a9ed\n76fa0000        77017000: ADVAPI32 @76fa1005
```

El punto de entrada se calculó con el operador \$imento(). Además, se utilizó el lenguaje de marcado del depurador (DML) para que se pueda hacer clic en el punto de entrada. Al hacer clic, se desensamblan las instrucciones en el punto de entrada.

El siguiente ejemplo busca todas las funciones en ntdll que contienen la subcadena Archivo en su nombre:

```
!para_cada_función -m:ntdll -p:"Archivo" -c:.echo @#NombreDeSímbolo
```

NOTA: Para ejecutar más comandos, enciérralos entre comillas o simplemente utilice uno de los comandos que ejecutan archivos de script.

Archivos de script

Se pueden utilizar varios comandos para indicarle a DbgEng que ejecute scripts. Estos comandos se dividen en dos categorías principales:

- Comandos que abren el archivo de script, reemplazan todas las líneas nuevas con un punto y coma (el separador de comandos) y concatenan todo el contenido en un solo bloque de comandos. Estos comandos tienen el siguiente formato: \$><.
- Comandos que abren el archivo de script e interpretan cada línea por separado. Estos comandos tienen el siguiente formato: \$<.

El primero es muy útil cuando se utiliza un comando de depuración que acepta otros comandos como argumentos. Por ejemplo, el comando bp realiza una acción de punto de interrupción, que puede ser un comando simple o un comando que ejecuta un archivo de script (que contiene varios comandos dentro de él).

Este último interpreta el contenido del archivo de script línea por línea; cada línea puede contener varios comandos separados por punto y coma. Cada comando ejecutado también se reflejará en la salida del depurador.

Algunos comandos del depurador interpretan la línea completa, sin tener en cuenta si hay un punto y coma (;) o no. Esto significa que el uso de los comandos relacionados con \$>< no funcionará para dichos scripts. Considere el siguiente script de ejemplo:

```
r eax;r ebx  
r $.u0 = Esto es solo una línea  
.printf "$u0"  
rcx:r edx
```

Ejecutar este script con \$>< no funciona como se esperaba:

```
0:000> $><prueba.wds  
eax=00000000  
ebx=00000000  
0:000> ? $u0  
  
No se pudo resolver el error en 'Esto es solo una línea;.printf ""';r ecx;r edx'
```

Por el contrario, ejecutar este script en particular con \$< funciona bien:

```
0:000> $<prueba.wds
0:000> r eax;r ebx
eax=00000000
ebx=00000000
0:000> r $.u0 = Esto es solo una línea
0:000> .printf "$u0"
Esto es solo una linea0:000> r ecx;r edx
ecx=f7fc0000
edx=00000000
```

La razón de este comportamiento es que al asignar un valor a un alias de nombre fijo, los puntos y comas también serán parte de la asignación. Esto explica por qué en la primera salida, el script parece haberse detenido de repente; es porque \$>< concatenará todas las líneas y las separará con un punto y coma.

Por la misma razón, si utiliza un comando que crea bloques y las llaves ({ y }) se utilizan en líneas separadas en el archivo de script, \$< no funcionará correctamente:

```
.si (1 == 2)
{
    .printf "¡De ninguna manera!\n";
}
.demás
{
    .printf "Eso es lo que pensé";
}
```

Al ejecutarse lo anterior devuelve el siguiente error:

```
(1)
0:000> $<bloque de prueba.wds
0:000> .si (1 == 2)
          ^
          Error de sintaxis en '.if (1 == 2)'
0:000> {
          ^
          Error de sintaxis en '{'
0:000>     .printf "¡De ninguna manera!";
¡De ninguna manera!0:000> }
          ^
          Error de sintaxis en '}'
0:000> .de lo contrario
          ^
          Error de sintaxis en '.else'
0:000> {
          ^
          Error de sintaxis en '{'
0:000>     .printf "Eso es lo que pensé";
Eso es lo que pensé0:000> }
          ^
          Error de sintaxis en '}'
(2)
0:000> $><p:\libro\scripts\l_blocktest.wds
Eso es lo que pensé
```

242 Capítulo 4 ■ Depuración y automatización

NOTA: Cuando los comandos de ejecución del script tienen como prefijo \$ adicional, el nombre o la ruta del archivo del script ya no pueden contener punto y coma. Cuando se encuentra un punto y coma después de \$\$>< o \$\$<, entonces lo que viene después se interpreta como otro conjunto de comandos.

Para ejecutar un archivo de script con su contenido concatenado en un solo bloque de comando, utilice \$\$>< o \$\$><:

```
$$><ruta_al_script.wds; r eax; al; bl;
```

Debido a que se utiliza \$\$><, el punto y coma permite ejecutar los comandos subsiguientes.

Pasar argumentos a archivos de script

Es posible pasar argumentos a los scripts usando el comando \$\$>a< :

```
$$>a<ruta_al_script.wds arg1 arg2 ...
```

A continuación, se puede acceder a los argumentos en el script a través de los alias \$argN . El alias \$arg0 contiene el nombre del script (como en argv[0] de C).

Si pasa UDPF como argumentos, no se ampliarán ni evaluarán antes de pasarlo al script. Esta es una situación complicada y puede dar lugar a diversos comportamientos inesperados. Por ejemplo, supongamos que llama a un script como este:

```
$$>a<script.wds @$t1 @$t2
```

Al script anterior se le pasarán los valores @\$t1 y @\$t2 como \${\$arg1} y \${\$arg2}, respectivamente. Para resolver este problema, asigne los pseudo-registros a un alias nombrado por el usuario y luego llame al script desde un .block. Esto garantizará la expansión de los valores del alias antes de que se pasen al script:

```
como /x val1 @$t0
como /x val2 @$t1
.bloquear
{
    $$>a<script.wds ${val1} ${val2}
}
.anuncio /q val1
.anuncio /q val2
```

Para comprobar si hay un argumento presente, utilice .if" con "\${/d:...}":

```
.atrapar
{
    .si ${/d:$arg1} == 0 o ${/d:$arg2} == 0
    {
        .printf "Uso: ${$arg0} dirección de memoria len\n";
        .dejar;
    }
    r $t0 = ${$arg1};
```

```

.si $vvalid($t0, 1) == 0
{
    .printf "Dirección de memoria no válida especificada\n";
    .dejar;
}

r $t1 = @$t0 + ${$arg2} - 1;
.printf "Suma de bytes de memoria de %x a %x\n", @$t0, @$t1;
.para (r $t3 = 0; @$t0 <= @$t1; r $t0 = @$t1 + 1)
{
    r $t3 = @$t3 + por(@$t1);
}
.printf "El resultado es %x\n", @$t3;
}

```

Utilizamos algunos trucos que merecen una breve explicación:

- .catch y .leave se utilizaron para simular el inicio y el “retorno” de una función.
Me gusta el comportamiento.
- .if y \${/d:\$arg1} se usaron para verificar si el primer argumento estaba definido.
Como no cambiamos explícitamente la sintaxis del evaluador, el motor de scripts evaluará utilizando MASM; por lo tanto, los operadores utilizados deben ser todos válidos en la sintaxis MASM. Encerrar una expresión con @@c++(expression) evaluará la expresión usando la sintaxis C++.
- El operador \$vvalid() se utiliza para verificar si la dirección de memoria pasada es válido.
- El token de comando .for se utiliza para recorrer el contenido de la memoria, y cada byte en esa ubicación se desreferencia mediante el operador by() de MASM .

En la siguiente salida, se le pasan al script varios argumentos:

```

(1)
0:000> $$>a<script.wds
Uso: script.wds dirección de memoria len
(2)
0:000> $$>a<script.wds 0xbadf00d
Dirección de memoria no válida especificada
(3)
0:000> $$>a<script.wds @eip 2
Suma de bytes de memoria de 76f83bc5 a 76f83bc6
El resultado es eb

```

En el marcador 1, el script se ejecuta sin ningún argumento y muestra sus argumentos correctamente. En el marcador 2, se le pasa al script una dirección de memoria no válida. Finalmente, en el marcador 3, el script se llama correctamente y se devuelve la suma de los bytes.

NOTA: La extensión de archivo .wds no es necesaria. Es solo una convención utilizada por varios programadores de scripts y significa archivo WinDbg Script.

Uso de scripts como funciones

No hay forma de definir funciones en el lenguaje de programación de DbgEng. Sin embargo, es posible utilizar varios archivos de script como si fueran funciones. Un script puede llamarse a sí mismo recursivamente o llamar a otro script con otro conjunto de argumentos, y esos argumentos serán diferentes en el contexto de cada script.

Las UDPR son muy útiles al escribir un script. Cuando un script llama a otro script, esas UDPR serán comunes a todos los scripts y, por lo tanto, no se pueden usar exclusivamente dentro de cada script sin alterar el estado de los otros scripts que llaman, a menos que, por supuesto, el script las guarde y restaure en sus puntos de entrada y salida.

NOTA: Debe pensar en la necesidad de preservar las UDPR en términos de registros en programas X86 o AMD64, donde el compilador se asegura de emitir código que preserva ciertos registros de propósito general en la entrada y salida de cada función mientras que (dependiendo de la convención de llamada) dedica ciertos registros para la entrada/salida de la función.

Con esto en mente, es importante diseñar un mecanismo que nos permita de manera fácil y sin problemas, y con la menor repetición posible, guardar/restaurar ciertas UDPR cada vez que un script vaya a llamar a otro.

El alias del archivo de script @call

En la sección anterior, describimos la necesidad de tener una forma de guardar o restaurar los UDRP. Por ese motivo, ideamos dos scripts simples que hacen justamente eso. Esta sección ilustra los scripts init.wds y call.wds y explica cómo funcionan.

El script init.wds se utiliza para configurar el entorno de scripting y crear alias cortos que actúen como nombres de funciones:

```
(1)
anuncio /q *;
(2)
como ${/v:SCRIPT_PATH} @"p:\libro\guiones";
.bloquear
{
    $$ Scripts invocables (usando @call) (3)
```

```

como ${/v:#sigma}           @"${RUTA_DE_SCRIPT}\sigma";
como ${/v:#pi}              @"${RUTA_DE_SCRIPT}\pi";
$$ Alias de llamadas de script (4)
como ${/v:@dvalloc} como $  @"$$>a<${RUTA_DE_SCRIPT}\dvalloc.wds";
{/v:@call}                  @"$$>a<${RUTA_DE_SCRIPT}\call.wds";
}

r $t19 = 0; (5)
r $t18 = 1; (6)

```

El script init.wds diseña dos convenciones de nombres de alias para nombres de usuario:

- Los nombres con el prefijo @ denotan alias del comando \$\$>a< (ejecutar un script con argumentos). Normalmente, se trata de scripts que son autosuficientes (no necesitan conservar las UDPR y no necesariamente se llaman a sí mismos ni a otros scripts).
- Los nombres con el prefijo # designan un alias que se puede llamar con el alias @call . Esos scripts pueden ser recursivos y pueden asumir con seguridad que se guardarán todos los UDPR que no sean los designados como valores de retorno. restaurado antes/después de que se llame/regrese un script.

El marcador 1 elimina todos los alias definidos previamente. En el marcador 2, se define la ruta base del script. (Observe el uso de @ para especificar una cadena literal). En el marcador 3, definimos dos alias nombrados por el usuario con el prefijo # definido. Estos se pueden llamar a través del alias @call y se evalúan como la ruta completa del script sin la extensión .wds . (El script de llamada agregará la extensión). Para fines de demostración, se definen dos scripts invocables: sigma y pi. En el marcador 4, definimos dos alias nombrados por el usuario con el prefijo @. Estos alias simplemente se resuelven en \$\$>a< seguido de la ruta completa del script. El alias @call es lo que hace posible la invocación de scripts como una función. @dvalloc es un contenedor alrededor del comando .dvalloc . El marcador 5 define la UDPR \$t19 , que es utilizada internamente por el script call.wds para recordar el nivel de anidamiento de las llamadas al script. El nivel de anidamiento se utiliza para formar un alias que guardará todas las UDPR por nivel de anidamiento. En el marcador 6, definimos la UDPR \$t18, que es utilizada internamente por call.wds para determinar cuántas UDPR a partir de \$t0 debe omitirse al restaurar las UDPR guardadas después de una llamada de script (más sobre esto en la siguiente explicación).

Aquí está el script call.wds :

```

anuncio /q ${/v:_tn_} (1)
.atrapar
{
.si ${/d:$arg1} == 0 (2)
{
.printf "No se especificó ningún script para llamar";
.dejar;
}

```

246 Capítulo 4 ■ Depuración y automatización

```

}

$$ Calcular el nombre de alias de los registros guardados de la llamada anterior aS /x ${/v:_ln_} @$t19; (3)

.bloquear
{
    $$ Eliminar el nombre de alias de los registros guardados de la ejecución anterior ad /q _sr_${_ln_};

} r $t19 = @$t19 + 1; $$ Incrementa el nivel de anidamiento (4)

$$ Calcular el nombre de alias de los registros guardados para la ejecución actual aS /x ${/v:_ln_} @$t19;
(5)

$$ Guardar todos los pseudo-registros
.bloquear
{
(6)
    aS /c _sr_${_ln_} "r $t0,$t1,$t2,$t3,$t4,$t5,$t6,$t7,$t8,$t9,
$t10,$t11,$t12,$t13,$t14,$t15,$t16,$t17";
}

$$ Llamar al script
.atrapar
(7)
    $$>a<"${$arg1}.wds" ${/f$arg2} ${/f$arg3} ${/f$arg4}
${/f$arg5} ${/f$arg6} ${/f$arg7} ${/f$arg8} ${/f$arg9} ${/f$arg10} ${/f$arg11} ${/f$arg12} ${/f$arg13} ${/f$arg14} ${/f$arg15}
${/f$arg16} ${/f$arg17} ${/f$arg18} ${/f$arg19} ${/f$arg20};

}

$$ Restaurar los registros después de llamar
.bloquear
{
(8)
    $$ Calcular el nombre de alias de los registros guardados aS /x ${/v:_ln_} @$t19;
.bloquear
{
    $$ Restaura todos los registros excepto los primeros que $$ deben devolver un
    valor .foreach /pS @$t18 /s (X "_sr_${_ln_}" ) (9)
    {
        r $(X); (10)
    }
}

$$ Eliminar el nombre de alias de los registros guardados ad /q _sr_${_ln_}; (11)

$$ Disminuir el nivel de anidamiento r $t19 =
@$t19 - 1; (12)
}

) ad /q ${/v:_ln_}; (13)

```

Este script necesita dos UDPR para fines especiales. El primero es \$t19, que se utiliza para almacenar el nivel de anidamiento de llamadas. Se incrementa cada vez que se utiliza @call para ejecutar un script y se decrementa cuando el script finaliza su ejecución. Debido a que \$t19 se incrementa y decrementa, puede crear un alias con un nombre único por nivel de anidamiento para almacenar los valores UDPR.

La segunda UDPR es \$t18, que se utiliza para designar el recuento de UDPR que se utilizan para devolver valores (a partir de \$t0). De forma predeterminada, el valor 1 indica que \$t0 es el único registro que se utilizará como valor de retorno. Si el script devuelve más de un valor (por ejemplo, en \$t0 y \$t1), el autor de la llamada debe establecer \$t18 en 2. antes de llamar al script. Esto garantiza que ni \$t0 ni \$t1 volverán a sus valores originales (los valores anteriores a la llamada al script). La llamada script toma el nombre del script que se llamará como primer argumento, seguido por el resto de los argumentos (\$arg2 a \$argN).

Ahora explicamos brevemente cómo funciona el resto de este script antes de ponerlo en acción. En el marcador 1, eliminamos el alias con nombre de usuario _tn_ (usado para calcular un nombre de alias por nivel de anidamiento) antes de redefinirlo. El marcador 2 verifica si se pasó un parámetro al script. En los marcadores 3 y 4, asignamos al alias _tn_ el valor numérico del nivel de anidamiento (observe el uso de aS /x), y luego incrementamos el nivel de anidamiento UDPR \$t19. En el marcador 5, creamos un alias temporal que tiene el valor del nivel de anidamiento actual.

En el marcador 6, guardamos las UDPR \$t0 a \$t17 en un alias llamado _sr_{_tn_} mediante el uso de aS /c seguido del comando r y la lista de UDPR para devolver sus valores. Por ejemplo, si el nivel de anidamiento es 2, el nombre de alias del registro guardado será _sr_2 y contendrá los valores de todas las UDPR en cuestión. _sr_0x2 equivaldrá a \$t0=00000003 \$t1=00000000 \$t2=00000000 ... \$t17=00000000.

En los marcadores 7 y 8, se llama al script que se pasó a \$arg1 con el resto de los argumentos que se pasaron. Después de que el script regrese, vuelva a calcular el alias _tn_. (El alias podría haber sido sobrescrito por el script llamado).

En los marcadores 9 y 10, iteramos en el alias _sr_NESTING_LEVEL actual pero omitimos los tokens \$t18 (observe el cambio /pS) y luego restauramos cada UDPR con el dominio.

En los marcadores 11-13, limpiamos los registros guardados alias (_sr_NESTING_LEVEL), Disminuye el nivel de anidamiento y elimina el alias de nombre temporal.

El siguiente paso es ejecutar el script init.wds que creará los alias apropiados:

Alias	Valor
-----	-----
#pi	"p\libro\scripts\pi" "p\libro\scripts\sigma"
#sigma	"p\libro\scripts\test" \$\$>a<p:
#prueba	"libro\scripts\call.wds" \$\$>a<p:
@llamar	"libro\scripts\dalloc.wds" p\libro\scripts
@dalloc	
RUTA DEL GUION	

248 Capítulo 4 ■ Depuración y automatización

Otra forma de hacerlo es ejecutar WinDbg (o cdb) con el parámetro de línea de comandos -c :

```
c:\dbg\windbg.exe -c "ad /q *;$><p:\book\scripts\init.wds;al;" p:\test.exe
```

Ahora puede saber que tiene dos alias para @call y @dvalloc que son scripts que no requieren guardar/restaurar automáticamente las UDPR, y otros tres scripts que dependen de @call para guardar/restaurar automáticamente las UDPR y actúan como "funciones".

El script sigma.wds toma dos parámetros numéricos y devuelve la suma de términos entre el primer y el segundo argumento, devolviendo el resultado en \$t0:

```
.para (r $t0=0, $t1=$($arg1), $t2=$($arg2); @$t1 <= @$t2; r $t1 = @$t1 + 1)
{
    r $t0 = @$t0 + @$t1;
}
```

Para ejecutar sigma.wds, utilice @call #sigma start_num end_num, de la siguiente manera:

```
0:000> @call #sigma 1 4;.printf "El resultado es %d\n", @$t0
El resultado es 10
```

De manera similar, el script pi.wds devuelve el resultado de la multiplicación de los términos entre el primer y el segundo argumento:

```
.para (r $t0=1, $t1=$($arg1), $t2=$($arg2); @$t1 <= @$t2; r $t1 = @$t1 + 1)
{
    r $t0 = @$t0 * @$t1;
}
```

El script dvalloc.wds es un contenedor del comando .dvalloc . Cuando Se llama a @dvalloc y el resultado se devuelve en \$t0 para que pueda usarse en scripts:

```
.atrapar
{
    r $t0 = -1; $$ Establecer resultado no válido
    .si ${/d$arg1} == 0
    {
        .printf "Uso: dvalloc.wds tamaño-de-memoria\n";
        .printf "La memoria asignada se devuelve en t0\n";
        .dejar;
    }

    $$ Asignar memoria y establecer el resultado en $t0
    .foreach /pS 5 (t {.dvalloc ${$arg1}}) {

        .si $vvalid(${!t}, 1) == 1
        {
            r $t0 = ${!t}; .dejar;
```

```
    }  
}
```

Después de asignar memoria con .dvalloc, tokenizamos el resultado y lo analizamos. saca la dirección de memoria a \$t0.

Tanto sigma.wds como pi.wds son funciones de ejemplo que utilizan \$t1 y \$t2 UDP. Esto significa que si un script llama a sigma o pi, \$t1 y \$t2 no deben modificarse de ninguna manera al regresar de ambas funciones al llamador.

Puede verificar este comportamiento con el siguiente script test.wds :

```
r $t1 = 0x123;  
r $t2 = 0x456;  
  
.printf "Antes de llamar a sigma: t1=%x, t2=%x\n", @$t1, @$t2  
@llamada #sigma 1 3  
.printf "Después de llamar a sigma: el resultado es t0=%x, t1=%x, t2=%x\n",  
@$t0, @$t1, @$t2
```

El script anterior asigna valores a las UDP \$t1 y \$t2 y luego llama a #sigma 1 3, que modificará \$t1 y \$t2. Si @call funciona como se espera, entonces esas UDP se restauran justo después de la llamada:

```
0:000> @llamada #prueba  
Antes de llamar a sigma: t1=123, t2=456  
Después de llamar a sigma: el resultado es t0=6, t1=123, t2=456
```

Ejemplos de scripts de depuración

En esta sección harás uso de varios scripts útiles, poniendo en práctica todo lo aprendido hasta ahora.

Obtención de la base de imagen de un módulo específico

Una forma rápida de obtener la imagen base de un módulo es usar el comando lm (listar módulos) con el interruptor m para listar los módulos que coinciden con el patrón especificado:

```
0:000> lm kernel32  
comenzar      fin          nombre del módulo  
749e0000 74b20000 KERNEL32 (difendo)
```

A partir de la salida, puedes ver que en el quinto token tienes la base de la imagen. De esta manera, la base de la imagen se puede analizar fácilmente con .foreach omitiendo los primeros tokens, extrayendo el valor del quinto token y saliendo del bucle:

```
r $t0 = -1;  
.foreach /pS 4 ( imgbase { imm ${Sarg1}; } ) {
```

250 Capítulo 4 ■ Depuración y automatización

```
r @$10 = ${!imgbase}; .break;
}
```

Cómo escribir un desempaquetador UPX básico

Escribir un descomprimidor UPX es bastante simple y hay muchas maneras de hacerlo. El método que se utiliza aquí es elaborado para ejercitarse en varios comandos del depurador. Se supone que tiene conocimientos básicos sobre el formato de archivo PE para comprender correctamente el script.

La idea detrás del guión es la siguiente:

1. UPX empaqueta el programa y mueve el punto de entrada original (OEP) de la sección .text , que es la primera sección.
2. El script calcula los límites de la primera sección y comienza a rastrear.
3. Si el puntero de instrucción (EIP) está fuera de la imagen del programa, entonces el script emite un gu para regresar al llamador.
4. El seguimiento continúa hasta que EIP se encuentra dentro de la primera sección. En ese momento, se supone que el programa se ha descomprimido.

Aquí está el guión:

```
$$ Obtener la base de la imagen
$$ Obtener la base de la imagen
como /x IMG_BASE @@c++(@$peb->ImageBaseAddress); (1)
$$ Declarar algunos alias con nombre de usuario que equivalen a UDP
como SEC_START @$t19; (2)
como SEC_END @$t18;
como IMG_START @$t17;
como IMG_END @$t16;

$$ Ir al punto de entrada del programa
g @$sexentry

.atrapar
{
    $$ Obtener puntero a los encabezados NT
    (3)
    r $10 = ${!IMG_BASE} + @@c++(({!_ENCABEZADO_DOS_DE_IMAGEN *} ${!IMG_BASE})->e_lfanew)

    $$ Ahora, desde IMAGE_NT_HEADERS.FileHeader, obtenga el tamaño opcional
    encabezamiento
    (4)
    r $t1 = @@c++(({!_IMAGE_NT_HEADERS *} @$10)->FileHeader.SizeOfOptionalHeader )

    $$ Calcular la dirección de la primera sección
    $$ omitir firma, tamaño de encabezados de archivo y tamaño de encabezados opcionales
    r $t2 = @$10 + 4 + @@c++(tamano(de(ole32:_ENCABEZADO_DE_ARCHIVO_DE_IMAGEN)) + @$t1; (5)
```

En el marcador 1, tomamos la base de la imagen del programa que se está ejecutando actualmente del pseudoregistro tipificado \$peb accediendo a su campo ImageBaseAddress mediante el evaluador C++ y luego lo almacenamos en un alias llamado IMG_BASE.

En el marcador 2, creamos un conjunto de alias con nombre de usuario que corresponden a algunas UDPR. Este es un buen truco para dar nombres a esas UDPR. En el marcador 3, asignamos la dirección de _IMAGE_NT_HEADERS a la UDPR \$t0 agregando la base de la imagen al valor del campo en IMAGE_DOS_HEADER.e_lfanew.

En el marcador 4, recuperaremos el tamaño de los encabezados opcionales en la `UDPR $t1`. Esto será útil para omitir todos los encabezados de PE y llegar al encabezado de la sección de la primera imagen.

252 Capítulo 4 ■ Depuración y automatización

En el marcador 5, calculamos la dirección de la primera sección de la imagen en la UDPR \$t2 .

En el marcador 6, analizamos desde IMAGE_SECTION_HEADER tanto la dirección virtual de la sección (inicio de la sección) como el final de la sección (inicio de la sección + tamaño de la sección).

En el marcador 7, calculamos las direcciones de inicio y fin del programa. La dirección de inicio es la base de la imagen y la dirección de fin es la base de la imagen más el contenido del campo IMAGE_OPTIONAL_HEADER.SizeOfImage .

En el marcador 8, comenzamos a hacer un bucle infinito usando un bucle for , \$t0 como contador, y el valor 1 como condición.

En los marcadores 9 a 11, utilizamos el comando t para rastrear una sola instrucción. No rastrear si el EIP no está dentro de los límites de la imagen y dejar de rastrear si el EIP está dentro de los límites de la primera sección.

Aunque este método es demasiado largo, ilustra cómo escribir un texto más complejo.

seguimiento de script y lógica en caso de que el proceso de desempaquetado sea más sofisticado.

La siguiente es una versión más simple del descompresor que busca un patrón de código que se ejecuta justo antes de que el programa esté a punto de pasar al punto de entrada original (OEP):

```
## Desempaquetado UPX con patrón
##UPX1: 0107D7F539C4
## UPX1:0107D7F7 75 FACTURA
## UPX1:0107D7F9 83 CE 80 ##
##UPX1:0107D7FC E9 ?? ?? ??
##                                especialmente, eax
##                                loc_107D7F3 corto
##                                Subtituto cmp jne esp, -80h
##                                saltar      cerca de pir word_103FC62

## Ir al punto de entrada del programa (no al punto de entrada original, sino al paquete
## uno)
## solo si no se especificaron argumentos
.si ${/d:$arg1} == 0
{
    g@$Sexentry;
}

##Patrón no encontrado!
r $t0 = 0; (1)
.foreach (dirección { s -{1}b @$ip L200 39 c4 75 fa 83 EC}) (2)
{
    ##Patrón encontrado!
    r $t0 = 1;
    r $t1 = ${dirección} + 7; (3)
    .printf /D "El JMP a OEP @<link cmd=l"u %x">%x</link>\n",@$t1,@$t1;
(4)
    es @$t1;
(5)
    tú;
    .romper;
}

.si $t0 == 0
{
    .printf "No se pudo encontrar el patrón de salto OEP. ¿El programa está empaquetado por UPX?\n";
}
```

En el marcador 1, usamos la UDPR \$t0 como una variable booleana para indicar si Se encontró el patrón.

En el marcador 2, buscamos el patrón a partir del punto de entrada y durante un máximo de 200 bytes utilizando la bandera 1 con el comando de búsqueda s. Esto devolverá solo la dirección donde se produjo la coincidencia. Si no se encuentra ninguna coincidencia, se devuelve una cadena vacía y, por lo tanto, el .foreach no tiene nada que tokenizar.

En el marcador 3, saltamos siete bytes más allá de la ubicación del patrón coincidente para señalar el salto relativo largo (que vuelve al OEP). Almacenamos esa dirección en \$t1.

En los marcadores 4 y 5, ejecutamos el programa hasta que se alcanza la instrucción JMP OEP (se utilizó el comando ga , por lo que se utiliza un punto de interrupción de hardware en lugar de un punto de interrupción de software) y luego trazamos una vez sobre la instrucción JMP OEP y así llegamos a la primera instrucción del programa descomprimido.

Cómo escribir un monitor de archivos básico

Este ejemplo crea un script que ilustra cómo utilizar scripts en combinación con puntos de interrupción condicionales para rastrear todas las llamadas a las versiones ASCII y Unicode de varias funciones de API de E/S de archivos: CreateFile, DeleteFile, GetFileAttributes, CopyFile, etc.

El script está diseñado para ser llamado una vez con el parámetro init para inicializarlo y luego varias veces como comando a los puntos de interrupción que crea cuando se inicializa.

Los siguientes parámetros se pasan cuando se llama al script desde el punto de interrupción:

- ApiName: se utiliza únicamente con fines de visualización.
- IsUnicode: pase cero para especificar que ésta es la versión ASCII de la API y pase uno para especificar que es la versión Unicode.
- FileNamePointerIndex: el número de parámetro en la pila que contiene el puntero al búfer de nombre de archivo
- ApilD: un ID de su elección, este parámetro es opcional. Esto es útil si desea agregar lógica adicional cuando se produce este punto de interrupción. En este script, CreateFile[A|W] recibe el ID 5. Más tarde, verifique si se activa esta API y luego verifique a qué nombre de archivo se accede y actúe en consecuencia.

Aquí está el contenido del script bp_displayfn.wds :

```
.atrapar
{
    .si '$$arg1' == 'init' (1)
    {
```

254 Capítulo 4 ■ Depuración y automatización

```
(2)
bp kernelbase!CreateFileA @“$>a<${$arg0} CrearArchivoA 0 1 5”, bp kernelbase!CreateFileW @“$>a<${$arg0}
CrearArchivoW 1 1 5”;
```

```
(3)
bp kernelbase!EliminarArchivoA @“$>a<${$arg0} EliminarArchivoW 0 1”, bp kernelbase!EliminarArchivoW
@“$>a<${$arg0} EliminarArchivoW 1 1”, bp kernelbase!BuscarPrimerArchivoA @“$>a<${$arg0}
BuscarPrimerArchivoA 0 1”, bp kernelbase!BuscarPrimerArchivoW @“$>a<${$arg0} BuscarPrimerArchivoW 1 1”, bp
kernel32!MoverArchivoA @“$>a<${$arg0} MoverArchivoA 0 1”, bp kernel32!MoverArchivoW @“$>a<${$arg0}
MoverArchivoW 1 1”, bp kernelbase!GetFileAttributesA @“$>a<${$arg0} ObtenerFileAttributesA
0 1”; bp kernelbase!GetFileAttributesExA @“$>a<${$arg0} ObtenerFileAttributesExA 0 1”; bp
kernelbase!GetFileAttributesExW @“$>a<${$arg0} ObtenerFileAttributesExW 1 1”; bp kernel32!CopyFileA @“$>a<${$arg0}
CopiarFileA 0 1”, bp kernel32!CopyFileW @“$>a<${$arg0} CopiarFileW 1 1”;
```

\$\$ Ignorar algunos eventos de depuración (para disminuir la contaminación de salida) sxí id;

\$\$ Muestra la lista de los puntos de interrupción recién instalados bl; (4)

.dejar;

} (5)

\$\$ Mostrar nombre de API .printf “\$
{\${\$arg1}}: >”, (6)

\$\$ Obtener el puntero del nombre del archivo r \$t0 =
poi(@\$csp + 4 * \${\$arg3});

(7)

\$\$ ¿Es un puntero de cadena Unicode? .if \${\$arg2} == 1 {

(8)

.printf “%mu<\n”, @\$t0;

}

.demás

{

\$\$ Mostrar como ASCII SZ (9) .printf “%ma<\n”,
@\$t0;

}

\$\$ ¿Conjunto de parámetros ApilD? (10) .si \${/d:

\$arg4} == 1 {

\$\$ ID de la API CreateFile? (11) .if \${/arg4} == 5

```

    {
        $$ Toma el nombre del archivo para que lo comparemos
        aS /mu ${/v:NOMBRE_DE_ARCHIVO} @$t0; (12)
        .bloquear
        {
            (13)
            .si $sicmp(@"${NOMBRE_DE_ARCHIVO}", @"c:\temp\eb.txt") == 0
            {
                .dejar; (14)
            }
        }
        anuncio /q ${/v:NOMBRE_DE_ARCHIVO};
    }
}

$$ Continuar después del punto de interrupción
gc; (15)
}

```

En el marcador 1, verificamos si el script se llama con init; si es así, inicializamos el script (marcadores 2 a 4) y salimos del script. En el marcador 2, creamos dos puntos de interrupción para CreateFileA/W y establecemos la condición para que sea el script en sí, y pasamos ApiID = 5.

En los marcadores 3 y 4, añadimos puntos de interrupción para el resto de las API sin pasar el argumento ApiID y luego regresamos del script. En los marcadores 5 y 6, imprimimos el nombre de la API y luego asignamos a \$t0 el puntero del nombre de archivo (usando el índice del parámetro pasado). En los marcadores 7 a 9, verificamos si el script se llama para la versión ASCII o Unicode de la API y luego usamos apropiadamente el especificador de formato %mu o %ma. En los marcadores 10 a 12, verificamos si se pasó un ApiID y es el CreateFile ApiID.

En los marcadores 12-14, extraemos el nombre del archivo en un alias llamado FILE_NAME, creamos un bloque para que el alias se expanda correctamente y luego comparamos el FILE_NAME alias contra una ruta de archivo deseada. (Observe el uso de @ para indicar expansión de cadena literal). Si la ruta coincide con lo que estamos buscando, el script finaliza y suspende la ejecución. Finalmente, en el marcador 15, el script reanudará la ejecución después de que se alcance cualquiera de los puntos de interrupción definidos.

Para utilizar este script, ejecútelo primero con el parámetro init :

```
0:000> $$>a<P:\libro\scripts\bp_displayfn.wds init; g;
```

Cómo escribir un descifrador de cadenas básico

Este script implementa una rutina de descifrado simple. Imagine que la rutina de descifrado en C es la siguiente:

```

void descifrar(unsigned char *p, tamaño_t sz)
{
    para (tamaño_t i=0;i<sz;i++, ++p)

```

NOTA: La rutina de descifrado puede ser más sofisticada. Si la rutina implica el uso de tablas y demás, recuerde que tiene acceso a esas tablas porque el script tiene acceso total a la memoria del programa depurado.

La siguiente es la misma rutina implementada utilizando el lenguaje de programación de DbgEng. Observe cómo utiliza el evaluador @@c++ para imitar fácilmente el algoritmo original:

El contenido de la memoria codificada es el siguiente:

```
0.000>db0x4180a4L30  
004180a4 bb 9e 8a 8f 9f 85 88 8d-87 cc 99 89 9d 89 99 9f .....  
004180b4 8e cc 8e 82 8c 85 85 89-8e 9e 82 82 8c ec eb ec .....  
004180c4 eb ec .....  
.....
```

Para descifrar, ejecute el script:

Uso del SDK

Hasta ahora hemos explicado cómo automatizar tareas mediante las funciones de creación de scripts que ofrecen las herramientas de depuración. El SDK que se incluye con las herramientas de depuración proporciona otra forma de automatizar o ampliar el depurador. Se entrega con archivos de encabezado, archivos de biblioteca para vincular la extensión y varios ejemplos que muestran cómo utilizar DbgEng de forma programática.

El SDK se encuentra en el subdirectorio sdk donde se encuentran las herramientas de depuración instalado. Tiene la siguiente estructura de directorios:

- **Ayuda:** contiene referencias a la biblioteca DbgHelp.
- **Inc:** contiene las inclusiones necesarias al utilizar el SDK.
- **Lib:** contiene los archivos de biblioteca adecuados que se utilizan durante la etapa de creación de vínculos. Contiene bibliotecas para WOA (Windows en ARM), AMD64 e i386.
- **Ejemplos:** contiene ejemplos de varios ejemplos escritos con los diferentes marcos de trabajo que se pueden usar para escribir extensiones del depurador. También hay ejemplos sobre cómo usar DbgEng en lugar de escribir una extensión para él.

Aunque cubrir el SDK está fuera del alcance de este capítulo, las siguientes secciones explican brevemente cómo utilizar el SDK para escribir extensiones de DbgEng para el depurador. El material cubierto debería ser suficiente para darle una ventaja, lo que le permitirá comprender fácilmente las extensiones de muestra y comenzar a aprender y escribir las suyas propias.

Para comenzar, debes saber que el SDK proporciona tres marcos con en las que puedes escribir extensiones:

- **Marco de extensión WdbgExts:** son las extensiones originales de WinDbg . Para interactuar con DbgEng, requieren exportar algunas devoluciones de llamadas para trabajar con las API de extensión de WinDbg en lugar de la interfaz de cliente de depuración. El programador puede adquirir posteriormente una interfaz de cliente de depuración u otras interfaces a pedido si se requiere más funcionalidad.
- **Marco de extensión DbgEng:** estos nuevos tipos de extensiones pueden proporcionar funcionalidad adicional al escritor de extensiones. Los comandos de extensión tienen acceso a una instancia de interfaz de cliente del depurador que les permite adquirir otras interfaces e interactuar más con DbgEng.
- **Extensiones EngExtCpp:** desarrolladas sobre el marco de extensión DbgEng, estas extensiones se crean subclasificando la clase base ExtExtension . La clase ExtExtension proporciona una variedad de funciones de utilidad que permiten que la extensión realice tareas complejas.

Las siguientes secciones ilustran brevemente cómo escribir extensiones utilizando el marco de extensión WdbgExts. Tenga en cuenta que escribir extensiones utilizando cualquiera de los otros marcos es bastante sencillo y se puede hacer siguiendo los ejemplos del SDK que se incluyen con el paquete de herramientas de depuración.

Conceptos

En esta sección se describen dos métodos para acceder a las API de DbgEng:

- A través de las interfaces del depurador, que se pueden recuperar utilizando una instancia de objeto de cliente de depuración.
- A través de una estructura pasada a la devolución de llamada de inicialización de la extensión WdbgExts . La estructura contiene un conjunto de punteros de función API que la extensión puede utilizar.

Las interfaces del depurador

DbgEng proporciona siete interfaces base para que las utilice el programador. Con el tiempo, se han añadido más funciones y, para preservar la compatibilidad con versiones anteriores, se han introducido nuevas versiones de esas interfaces. Por ejemplo, en el momento de redactar este artículo, IDebugControl es la primera versión de la interfaz y IDebugControl4 es la última versión de esta interfaz.

A continuación se muestra la lista de interfaces y una breve explicación de su propósito y algunas de las funciones que proporcionan:

- **IDebugClient5**: esta interfaz proporciona varias funciones útiles para iniciar o detener una sesión de depuración y establecer las devoluciones de llamadas DbgEng necesarias (entrada/salida/eventos). Además, su método QueryInterface se utiliza para recuperar las interfaces de las interfaces restantes.
 - **CreateProcess/AttachProcess**: crea un nuevo proceso o lo adjunta a uno existente:
 - **AttachKernel**: se conecta a un depurador de kernel en vivo.
 - **GetExitCode**: Devuelve el código de salida de un proceso.
 - **OpenDumpFile**: inicia una sesión de depuración desde un archivo de volcado.
 - **SetInputCallbacks/SetEventCallbacks**: establece la entrada/salida devoluciones de llamadas.
- **IDebugControl4**: esta interfaz proporciona funciones relacionadas con el control de procesos:
 - **AddBreakpoint**: agrega un punto de interrupción.
 - **Ejecutar**: ejecuta un comando del depurador.

- SetInterrupt: indica a DbgEng que ingrese al objetivo.
- WaitForEvent: espera hasta que se produzca un evento del depurador. Esto es similar a la API Win32 WaitForDebugEvent() .
- SetExecutionStatus: establece el estado de DbgEng. Esto permite al programador reanudar la ejecución, solicitar un paso hacia adelante o hacia atrás, etc.
- IDebugDataSpaces4: esta interfaz proporciona información relacionada con la memoria y los datos.
 - Funcionalidad:
 - ReadVirtual: lee la memoria desde la memoria virtual del destino.
 - QueryVirtual: equivalente a VirtualQuery() de Win32 , esta función consulta la memoria virtual del espacio de direcciones virtuales del destino.
 - ReadMsr: lee el valor del registro específico del modelo.
 - WritePhysical: escribe en la memoria física.
 - IDebugRegisters2: proporciona introspección de registros (enumeración, consulta de información) y funcionalidad de configuración/obtención. DbgEng asigna un índice a los registros. Para trabajar con un registro nombrado, primero debe averiguar su índice:
 - GetDescription: devuelve una descripción del registro (tamaño, nombre, tipo, etc.).
 - SetValue/GetValue: establece/obtiene el valor de un registro.
 - GetIndexByName: busca un índice de registro dado su nombre.
 - IDebugSymbols3: proporciona funcionalidad para gestionar símbolos de depuración, información de línea de origen, tipos de consulta, etc.:
 - GetImagePath: devuelve la ruta de la imagen ejecutable.
 - GetFieldName: devuelve el nombre de un campo dentro de una estructura.
 - IDebugSystemObjects4: proporciona funcionalidad para consultar información de los destinos depurados y el sistema en el que se ejecutan: ■ GetCurrentProcessId:
 - devuelve el ID del proceso DbgEng del proceso depurado actualmente.
 - GetCurrentProcessHandle: devuelve el identificador del sistema del proceso actual. proceso.
 - SetCurrentThreadId: cambia el hilo actual según su ID de DbgEng. Esto es equivalente al comando ~Nk .
- IDebugAdvanced4: proporciona más funcionalidad que no necesariamente está presente en las otras interfaces:
 - GetThreadContext/SetThreadContext: obtiene/establece el contexto del hilo.
 - GetSystemObjectInformation: devuelve información sobre el objeto deseado. objeto del sistema.

260 Capítulo 4 ■ Depuración y automatización

Para utilizar las API a través de las interfaces, es necesario tener una instancia de la interfaz `IDebugClient` (cliente del depurador) o cualquiera de sus interfaces derivadas. En el siguiente fragmento de código, la instancia de la interfaz `IDebugClient5` se pasa a la función de utilidad `CreateInterfaces`. Esta última llama a `QueryInterface` de forma repetitiva para recuperar las interfaces necesarias:

```
bool CreateInterfaces(IDebugClient5 *Cliente) {  
  
    // ¿Ya se crearon interfaces? if (Control != NULL)  
  
    devuelve verdadero;  
  
    // Obtener la interfaz del cliente de depuración si (Cliente  
    == NULL)  
    {  
        m_LastHr=m_pDebugCreate(__uuidof(IDebugClient5),(void**)&Client); si (m_LastHr != S_OK) devuelve falso;  
  
    }  
  
    // Consulta de otras interfaces que necesitaremos.  
    hacer  
    {  
        m_LastHr = Cliente->QueryInterface( __uuidof(IDebugControl4),  
            (void**)&Control); si (m_LastHr != S_OK)  
            romper;  
  
        m_LastHr = Cliente->QueryInterface( __uuidof(IDebugSymbols3),  
            (void**)&Symbols); si  
        (m_LastHr != S_OK)  
            romper;  
        m_LastHr = Cliente->QueryInterface( __uuidof(IDebugRegisters2),  
            (void**)&Registers);  
  
        si (m_LastHr != S_OK) romper;  
  
        m_LastHr = Cliente->QueryInterface( __uuidof(IDebugSystemObjects4),  
            (void**)&SystemObjects); si (m_LastHr !=  
            S_OK)  
            romper;  
        m_LastHr = Cliente->QueryInterface( __uuidof(IDebugAdvanced3),  
            (void**)&Advanced); si  
        (m_LastHr != S_OK) romper;  
  
        m_LastHr = Cliente->QueryInterface( __uuidof(IDebugDataSpaces4),  
            (void**)&DataSpace); } mientras  
        (falso);
```

```
    devuelve EXITOSO(m_LastHr);
}
```

Las variables de interfaz se definen así:

Espacios de datos de depuración 4	*Espacio de datos;
IDebugRegisters2	*Registros;
Símbolos de depuración de ID3	*Símbolos;
Control de errores 4	*Control;
IDebugSystemObjects4 *Objetos del sistema;	
IDebugAdvanced3	*Avanzado;

Para adquirir una interfaz de cliente de depurador (IDebugClient), utilice DebugCreate o la función DebugConnect (conectarse a un host remoto). El siguiente ejemplo adquiere una interfaz de cliente de depurador mediante DebugCreate:

```
Estado HRESULT;
IDebugClient *Cliente;
si ((Estado = DebugCreate(__uuidof(IDebugClient),
                           (void**)&Cliente)) != S_OK)
{
    printf("DebugCreate falló, 0x%X\n", Estado);
    devuelve -1;
}
// Bien, ahora estamos listos para consultar otras interfaces...
```

API de extensión de WinDbg

Las extensiones del depurador reciben un puntero a una estructura WINDBG_EXTENSION_APIS a través de la rutina de devolución de llamada de inicialización de la extensión WinDbgExtensionDlInit .

La estructura tiene los siguientes punteros de API:

```
// wdbgexts.h
tipo de definición de estructura _WINDBG_EXTENSION_APIS {
    ULONG nTamaño;
    RUTINA DE SALIDA PWINDBG;
    PWINDBG_OBTENER_EXPRESSION Rutina de salida Ip;
    PWINDBG_OBTENER_SÍMBOLO Rutina de obtención de expresiones Ip;
    PWINDBG_DISASM lpGetSymbolRoutine;
    CONTROL DE VERIFICACIÓN PWINDBG_C lpRutinaDeDesasmo;
    PWINDBG_RUTINA_DE_MEMORIA_DE PROCESO DE LECTURA lpCheckControlCRoutine;
    PWINDBG_RUTINA_DE_MEMORIA_DE PROCESO DE ESCRITURA lpWriteProcessMemoryRoutine;
    PWINDBG_RUTINA_OBTENCIÓN DE CONTEXTO DE HILO lpReadProcessMemory;
    PWINDBG_ESTABLECER_RUTINA_DE CONTEXTO DE HILO lpGetThreadContext;
    RUTINA PWINDBG_IOCTL lpSetThread;
    RUTINA DE SEGUIMIENTO DE PILAS PWINDBG lpIoctlRutina;
    RUTINA DE SEGUIMIENTO DE PILAS PWINDBG lpStack;
} API DE EXTENSIÓN DE WINDBG, *APIS DE EXTENSIÓN DE WINDBG;
```

262 Capítulo 4 ■ Depuración y automatización

Cuando la extensión recibe esta estructura, debe copiarla y almacenarla en una variable global, preferiblemente denominada ExtensionApis. La razón para elegir este nombre de variable en particular es que el archivo de encabezado wdbgexts.h define algunas macros que hacen referencia a ExtensionApis para acceder a los punteros de la API:

```
extern WINDBG_EXTENSION_APIS ExtensionApis;

#define dprintf          (ExtensionApis.lpOutputRoutine)
#define ObtenerExpresión (Rutina de obtención de expresiones de extensión APIs.lp)
#define CheckControlC   (ExtensionApis.lpCheckControlCRoutine)
#define Obtener contexto (Rutina de obtención de subprocessos de extensión APIs.lp)
...
#define LeerMemoria     (ExtensionApis.lpReadProcessMemoryRoutine)
#define EscrituraMemoria (ExtensionApis.lpWriteProcessMemoryRoutine)
#define el seguimiento de pila (ExtensionApis.lpStackTraceRoutine)
```

Estas macros permiten a los escritores de extensiones llamar directamente a StackTrace o WriteMemory, por ejemplo, en lugar de usar pExtension.lpStackTraceRoutine o pExtension.WriteMemory.

A parte de poder utilizar únicamente las funciones declaradas en WINDBG_Estructura EXTENSION_APIS , también es posible utilizar toda una gama de otras funciones que se basan en la función ExtensionApis.lpIoctlRoutine .

Por ejemplo, ReadPhysical() es una función en línea que llama a IoCtl() con el código de control IG_READ_PHYSICAL mientras le pasa los parámetros apropiados.

Consulte el archivo de ayuda de DbgEng para obtener una lista de funciones que puede utilizar dentro de las extensiones WdbgExts.

Cómo escribir extensiones para herramientas de depuración

En la sección anterior aprendiste los conceptos detrás del SDK; ahora estás listo para profundizar en más detalles sobre cómo se ve una extensión WdbgExts y cómo escribir una extensión muy básica.

Una extensión de depurador es simplemente una DLL de Microsoft Windows. La DLL debe exportar dos funciones obligatorias que necesita DbgEng y luego exportar tantas funciones como la extensión le proporcione al depurador.

La primera función que se debe exportar es WinDbgExtensionDllInit .

Se llama cuando el depurador carga su extensión:

```
ANULAR WinDbgExtensionDllInit(
    PWINDBG_EXTENSION_APIS lpExtensionApis,
    USHORT Versión principal,
    USHORT Versión menor)
{
    ExtensionApis = *lpExtensionApis; // Tomar una copia

    // Opcionalmente también guarde la información de la versión
    VersiónMayorGuardada = VersiónMayor;
```

```
VersiónMinorGuardada = VersiónMinor;  
  
devolver;  
}
```

Tenga en cuenta que guarda el contenido del puntero lpExtensionApis que se pasa. Las variables de información de versión que se pasan indican el tipo de compilación y el número de compilación de Microsoft Windows, respectivamente. Opcionalmente, guarde esas variables si desea comprobar sus valores en los comandos de extensión más adelante.

La segunda función que se debe exportar es ExtensionApiVersion. DbgEng la llama cuando desea consultar la información de la versión de su extensión:

```
EXT_API_VERSION Versión API = {  
  
    5, // Mayor  
    1, // Menor  
    EXT_API_VERSION_NUMBER64, // Revisión  
    0 // Reservado  
};  
  
LPEXT_API_VERSION Versión API de extensión (VOID)  
{  
    devuelve &ApiVersion;  
}
```

Ahora que se han definido las funciones obligatorias (o devoluciones de llamadas), proceda declarando los comandos de extensión.

Un comando de extensión tiene la siguiente declaración:

```
CPPMOD VOID miextensión(  
    MANEJAR hProcesoActual,  
    MANEJAR hHilo actual,  
    ULONG dwCurrentPc,  
    ULONG Procesador dw,  
    PCTR-ES args)
```

Los argumentos pasados más notables son los siguientes:

- dwProcessor: el índice del procesador actual
- dwCurrentPc: el puntero de instrucción actual
- args—Los argumentos pasados (si los hay)

Otra forma preferida de declarar una función de extensión es utilizar DECLARE_MACRO API(api_s) :

```
DECLARE_API(prueba)  
{  
    dprintf("Esta es una rutina de extensión de prueba");  
}
```

NOTA: En cualquier momento, cualquier comando de extensión puede llamar a DebugCreate() y luego obtener cualquier interfaz que desee para obtener funcionalidad adicional.

El paso final es exportar las dos funciones obligatorias y los comandos de extensión que planeas exponer a DbgEng. La forma habitual es crear un archivo .def y llamar al enlazador con un modificador adicional /DEF:filename.def . Así es como se ve el archivo DEF para la extensión de prueba que escribimos:



Coloque la DLL resultante en el directorio de herramientas de depuración (o en el directorio winext). subdirectorio) o en el directorio del sistema de Windows. Utilice !load extname para cargar la extensión compilada y luego !extension_command o !extname.ext_command para ejecutar el comando de extensión.

Extensiones, herramientas y recursos útiles

A continuación se muestra una breve lista de extensiones, herramientas y recursos útiles que pueden mejorar su experiencia de depuración:

- **narly** (<https://code.google.com/p/narly/>) : A extensión útil que enumera los controladores /SAFESEH , muestra información sobre /GS y DEP, busca gadgets ROP y proporciona otros comandos diversos.
- **SOS**: esta extensión, que se entrega con el Kit de controladores de Windows (WDK), facilita la depuración de código administrado.
- **!analyze**: una extensión muy útil (se incluye con DbgEng) que muestra información sobre la excepción o comprobación de errores actual.
- **VirtualKd** (<http://virtualkd.sysprogs.org/>): Este es una herramienta que mejora la velocidad de depuración del kernel cuando se utiliza con VMWare o VirtualBox.
- **windbg.info**: este sitio web proporciona una guía muy completa sobre WinDbg/ Referencia de comandos DbgEng y foro de discusión para usuarios.
- **kdext.com**: este sitio web ofrece un par de extensiones DbgEng. Una extensión notable es la extensión de resaltado de sintaxis de ensamblado y mejoras de la interfaz de usuario.

- Scripts WinDbg de SysecLabs (www.laboskopia.com/download/SysecLabs-WinDbg-Script.zip)—A conjunto de scripts que le ayudan a inspeccionar el kernel. Especialmente útil para la caza de rootkits.
- lexploitable (<http://msecdbg.codeplex.com/>)—Un extensión que proporciona análisis automatizado de fallos y evaluación de riesgos de seguridad.
- Qb-Sync (<https://github.com/quarkslab/qb-sync>): una ingeniosa extensión de WinDbg de Quarkslab que permite sincronizar la vista de desmontaje o gráfico de IDA Pro con WinDbg.
- Pykd (<http://pykd.codeplex.com/>) : un Extensión de Python para acceder a la Inglés:

Ofuscación

La ingeniería inversa del código generado por el compilador es un proceso difícil y que requiere mucho tiempo . La situación empeora aún más cuando el código ha sido reforzado, construido deliberadamente para resistir el análisis. Nos referimos a estas técnicas para reforzar los programas bajo el paraguas general de la ofuscación. Algunos ejemplos de situaciones en las que se podría aplicar la ofuscación son los siguientes:

- Malware: evitar el escrutinio de los motores de detección antivirus y de los ingenieros inversos es el motivo principal de los delincuentes que emplean malware en sus operaciones y, por lo tanto, esta ha sido una aplicación tradicional de ofuscación durante muchos años.
- Protección de la propiedad intelectual: muchos programas comerciales cuentan con algún tipo de protección contra la duplicación no autorizada. Algunos sistemas emplean una mayor ofuscación con el fin de ocultar los detalles de implementación de ciertas partes del sistema. Algunos buenos ejemplos son Skype, iMessage de Apple o incluso el cliente Dropbox, que protegen sus formatos de protocolo de comunicación con ofuscación y criptografía.

- Gestión de derechos digitales: los sistemas DRM suelen proteger determinados datos cruciales (por ejemplo, claves criptográficas y protocolos) mediante ofuscación. FairPlay de Apple, Media Foundation Platform de Microsoft y su DRM PlayReader, por citar sólo dos, son ejemplos de aplicaciones de ofuscación . Actualmente, esta es la principal aplicación contemporánea de ofuscación.

En términos abstractos, la “ofuscación” puede verse en términos de transformaciones de programas . El objetivo de tales métodos es tomar como entrada un programa y producir como salida un nuevo programa que tenga el mismo efecto computacional que el programa original (formalmente hablando, esta propiedad se llama equivalencia semántica o equivalencia computacional), pero al mismo tiempo es “más difícil” de analizar.

La noción de “dificultad de análisis” ha sido definida informalmente desde hace mucho tiempo, sin ningún rigor matemático que la respalde. Por ejemplo, se cree ampliamente que, en lo que respecta a un analista humano, el tamaño de un programa es un indicador de la dificultad para analizarlo. Un programa que consume 20.000 instrucciones para realizar una sola operación podría considerarse “más difícil” de analizar que uno que necesita una instrucción para realizar la misma operación. Tales suposiciones son dudosas y han atraído el escrutinio de los teóricos (como el de Mila Dalla Preda¹⁷ y Barak et al.²).

Se han propuesto varios modelos para representar un ofuscador y (de forma dual) un desofuscador. Estos modelos son útiles para mejorar el diseño de herramientas de ofuscación y razonar sobre su robustez, a través de criterios adaptados. Entre ellos, dos modelos revisten especial interés.

El primer modelo es adecuado para el análisis de mecanismos criptográficos, en el contexto de los denominados ataques de caja blanca. Este modelo define al atacante como un algoritmo probabilístico que intenta deducir una propiedad pertinente de un programa protegido. Más precisamente, intenta extraer información distinta a la que se puede deducir trivialmente del análisis de las entradas y salidas del programa. Esta información es pertinente en el sentido de que permite al atacante eludir una función de seguridad o representarse a sí misma como datos críticos del programa protegido. De doble manera, un ofuscador se define en este modelo como un generador probabilístico de una caja negra virtual, un ofuscador ideal que asegura que el análisis del programa protegido no proporcione más información que el análisis de sus distribuciones de entrada y salida.

Otra forma de formalizar a un atacante es definir la acción de ingeniería inversa como una interpretación abstracta de la semántica concreta del programa protegido. Naturalmente, esta definición es adecuada para el análisis estático del flujo de datos del programa, que es el primer paso antes de la aplicación de transformaciones de optimización. De doble manera, un ofuscador se define en el modelo de interpretación abstracta como un compilador especializado, parametrizado por algunas propiedades semánticas que no se conservan.

El objetivo de estos intentos de modelado es obtener algunos criterios objetivos relativos a la robustez efectiva de las transformaciones de ofuscación. De hecho, muchos problemas

Los problemas que antes se consideraban difíciles se pueden abordar de manera eficiente mediante la aplicación juiciosa de técnicas de análisis de código. Muchos métodos que han surgido en el contexto de temas más convencionales en la teoría de lenguajes de programación (como compiladores y verificación formal) se pueden reutilizar con el fin de derrotar la ofuscación.

Este capítulo comienza con un estudio de las técnicas de ofuscación existentes que se encuentran comúnmente en situaciones del mundo real. Luego, cubre los diversos métodos y herramientas disponibles desarrollados para analizar y posiblemente descifrar el código de ofuscación. Por último, ofrece un ejemplo de un esquema de ofuscación moderno y difícil, y detalla cómo evitarlo utilizando técnicas de análisis de última generación.

Un estudio de las técnicas de ofuscación

Para simplificar la presentación, comenzaremos dividiendo las ofuscaciones en dos categorías: ofuscación basada en datos y ofuscación basada en control. Más adelante verá que las dos se combinan de maneras complejas y difíciles y que, de hecho, son inseparables. Sin embargo, antes de adentrarnos en estos temas, comenzaremos con un ejemplo representativo de los tipos de código que uno podría encontrar en la ofuscación del mundo real. Tenga en cuenta que el ejemplo es particularmente simple porque involucra solo ofuscaciones basadas en datos, no basadas en control.

La naturaleza de la ofuscación: un ejemplo motivador

Cuando se trabaja con un procesador x86, los compiladores tienden a generar instrucciones extraídas de un subconjunto particular y minúsculo del conjunto de instrucciones disponible, y la estructura de control del programa generado sigue convenciones predecibles. Con el tiempo, el ingeniero inverso desarrolla un estilo de análisis adaptado a estos patrones de código estructurado. Cuando uno se enfrenta a un código no conforme, la velocidad del análisis puede verse afectada enormemente.

Este fenómeno se puede ilustrar de forma sencilla con un ejemplo concreto. Como uno de los objetivos de un optimizador de compiladores es reducir la cantidad de recursos computacionales necesarios para realizar una tarea, y 50 años de investigación les han dado formidables capacidades para lograr este objetivo, no es habitual detectar ineficiencias obvias en la traducción del código fuente original al lenguaje ensamblador. Por ejemplo, si el código fuente dictara que alguna variable se incremente en cinco (por ejemplo, debido a una declaración como `x += 5;`), un compilador probablemente generaría un código ensamblador similar a uno de los siguientes:

```
01: agregar eax, 5 02:  
agregar dword ptr [ebp-10h], 5 03: lea ebx, [ecx+5]
```

En un código ofuscado, uno podría encontrar un código como el siguiente, suponiendo que EAX corresponde a la variable x, y que el valor de EBX puede sobrescribirse (o “modificarse”):

```
01: xor ebx, eax
02: xor eax, ebx
03: xor ebx, eax
04: inc. eax.
05: ebx negativo
06: agregar ebx, 0A6098326h
07: cmp eax, esp
08: movimiento eax, 59F67CD5h
09: xor eax, 0FFFFFFFh
10: sub ebx, eax
11: rcl eax, cl
12: empuje 0F9CBE47Ah
13: agregar dword ptr [esp], 6341B86h
14: sbb eax, ebp
15: subdword [esp], ebx
16: empuje
17: empujador
18: pop eax
19: añadir esp, 20h
20: prueba ebx, eax
21: pop eax
```

En este ejemplo se puede observar una variedad de técnicas de ofuscación en acción:

- Las líneas 1 a 3 utilizan el “ truco de intercambio XOR ” para intercambiar el contenido de dos ubicaciones; en este caso, los registros EAX y EBX .
- La línea 4 muestra una asignación al registro EAX que en realidad es “basura” (ya que EAX se sobrescribe con una constante en la línea 8).
- En las líneas 5 y 6, el registro EBX se niega y se agrega a la constante 0A6098326h: EBX = -EAX + 0A6098326h.
- En la línea 7, se compara EAX con ESP. La instrucción CMP modifica únicamente los indicadores, y estos se sobrescriben en las líneas subsiguientes antes de volver a usarse, por lo que este código es basura.
- Las líneas 8 y 9 mueven la constante 59F67CD5h al registro EAX y la XOR con -1h (que, en binario, son todos bits uno). La XOR con todos los bits uno es equivalente a la operación NOT ; por lo tanto, el efecto de esta secuencia es mover la constante 59F67CD5h al registro EAX y la XOR con -1h (que, en binario, son todos bits uno). constante 0A609832Ah en EAX.
- La línea 10 resta la constante en EAX de EBX: EBX = - EAX + 0A6098326h - 0A609832Ah, o EBX = - EAX - 5, o EBX = -(EAX + 5).
- La línea 11 modifica EAX mediante el uso de la instrucción RCL . Esta instrucción es basura porque EAX se sobrescribe en la línea 18.

- Las líneas 12 y 13 introducen la constante 0F9CBE47Ah y luego le suman la constante 6341B86h , lo que da como resultado el valor 0h en la parte inferior de la pila.
- La línea 14 modifica EAX mediante el uso de la instrucción SBB , que involucra el registro extraño EBP. Esta instrucción es basura, ya que EAX se sobrescribe en la línea 18.

- La línea 15 resta EBX del valor que se encuentra actualmente en la parte inferior de la pila (que es 0h). Por lo tanto, dword ptr [ESP] = 0 - -(EAX + 5), o dword ptr [ESP] = EAX + 5.
- Las líneas 16 a 19 demuestran operaciones que involucran la pila: se insertan nueve dwords, se introduce una en EAX y luego se ajusta el puntero de la pila para que apunte a la misma ubicación que apuntaba antes de que se ejecutara la secuencia.
- La línea 20 prueba EBX contra el registro EAX y establece los indicadores en consecuencia. Si los indicadores se redefinen antes de su próximo uso, esta instrucción no se ejecuta.
- La línea 21 coloca el valor en la parte inferior de la pila (que contiene EAX + 5) en el registro EAX .

En resumen, el código calcula $EAX = EAX + 5$.

No hace falta decir que el código ofuscado no se parece en nada al código generado por el compilador y resulta muy difícil determinar la funcionalidad del fragmento. En este ejemplo se utilizan varias técnicas de ofuscación:

- Ofuscación basada en patrones
- Despliegue constante
- Inserción de código basura
- Ofuscación basada en pila
- El uso de instrucciones poco comunes, como RCL, SBB, PUSHF y PUSHAD

En consecuencia, se puede utilizar una variedad de transformaciones del compilador existentes para representar el código en una forma más cercana al original:

- Optimización de mirillas ■
- Plegado constante ■
- Eliminación de declaraciones muertas
- Optimización de la pila

La interacción entre el flujo de datos y el flujo de control

Considere la siguiente secuencia de instrucciones:

```
01: movimiento eax, dword ptr [ebp-10h]  
02: jmp eax
```

Supongamos que desea construir un gráfico de flujo de control clásico “correcto” para un programa que contiene secuencias como ésta. Para determinar cuál será la siguiente instrucción después de que se haya ejecutado la línea 2 (o, quizás, el conjunto de posibles instrucciones sucesoras), debe determinar el conjunto de valores posibles para el registro EAX en esa ubicación. En otras palabras, el flujo de control para este fragmento depende del flujo de datos en lo que respecta a la ubicación [EBP-10h] en el punto de programa l1 (línea 1). Sin embargo, para determinar el flujo de datos con respecto a [EBP-10h], debe determinar el flujo de control con respecto a la ubicación de la línea 1: debe conocer todas las posibles instrucciones de transferencia de control (y el flujo de datos asociado que conduce a esas ubicaciones) que posiblemente podrían tener como objetivo la ubicación de la línea 1. No tiene sentido hablar sobre el flujo de control sin hablar simultáneamente sobre el flujo de datos, o viceversa.

La situación es aún más difícil de lo que parece a primera vista. El análisis de programas intenta responder a preguntas como “¿Qué valores podría asumir la ubicación [EBP- 10h] en cualquier circunstancia posible?”. Para combatir la intratabilidad y la indecidibilidad, muchas formas de análisis de programas emplean aproximaciones del espacio de estados. Algunas aproximaciones son finas (por ejemplo, aproximar el conjunto {1,3} por {1,2,3}), y otras son burdas (por ejemplo, aproximar ese mismo conjunto por {0,1,...,232 –1}).

(Fino y grueso no son términos técnicos en este párrafo). Si no puede aproximarse con precisión al conjunto de valores potenciales de la ubicación [EBP-10h] (por ejemplo, si debe asumir que la ubicación podría tomar cualquier valor posible), entonces no sabe a dónde apuntará el salto, por lo que debe asumir que podría apuntar a cualquier ubicación dentro del espacio de direcciones. Luego, los datos del flujo de datos de la ubicación de la línea 2 deben propagarse a los de todas las demás ubicaciones. En entornos prácticos , una decisión de este tipo afectará gravemente el análisis, lo que probablemente hará que concluya de manera conservadora que todos los estados son posibles en todas las ubicaciones, lo cual es correcto pero inútil.

Peor aún, si alguna vez debe asumir que un salto podría tener como objetivo cualquier ubicación, entonces, debido a la codificación de instrucciones de longitud variable en x86, muchas de estas transferencias se realizarán en ubicaciones que no corresponden al comienzo de una instrucción adecuada. Es probable que estas instrucciones falsas causen estragos en cualquier análisis, especialmente cuando se combinan con las observaciones del párrafo anterior.

Los trabajos académicos en esta área, como los de Kinder30 y Thakur et al.⁴¹, buscan construir sistemas que puedan devolver respuestas correctas para todas las entradas posibles. Estos sistemas prefieren decirles a los usuarios que no pueden determinar información precisa, devolver resultados correctos pero extremadamente imprecisos o morir en el intento (por ejemplo, agotando toda la memoria disponible o no logrando terminar debido a problemas de manejabilidad), en lugar de dar una respuesta que no está completamente justificada. Este objetivo es loable, dada la motivación que dio origen a estas disciplinas: asegurar la absoluta corrección de los programas y análisis. Sin embargo, no está en línea con nuestras motivaciones como investigadores de la ofuscación.

La desofuscación es una criatura de un tipo diferente a la verificación formal o Análisis de programas, aunque preferimos utilizar técnicas desarrolladas en esos contextos . Mientras que un ofuscador transforma un programa Porig en un programa Pobf, buscamos un traductor de Pobf a Porig o suficiente información sobre Pobf para responder a preguntas próximas a algún esfuerzo de ingeniería inversa. Dudamos en utilizar métodos poco sólidos, pero preferimos los resultados reales cuando el día haya terminado, por lo que podemos emplear dichos métodos, aunque sea de forma consciente y a regañadientes.

Ofuscaciones basadas en datos

Comenzamos analizando las técnicas de ofuscación que se pueden describir mejor en términos de su efecto sobre los valores de los datos y los cálculos no relacionados con el control. En particular, supongamos que los fragmentos presentados se encuentran dentro de un único bloque básico del gráfico de flujo de control del programa. Los análisis de las ofuscaciones basadas en el control y su combinación con las ofuscaciones basadas en datos se posponen para secciones posteriores.

Despliegue constante

El plegado de constantes es una de las primeras y más básicas optimizaciones del compilador. El objetivo de esta optimización es reemplazar los cálculos cuyos resultados se conocen en tiempo de compilación con esos resultados. Por ejemplo, en la declaración de C $x = 4 * 5;$, la expresión $4 * 5$ consiste en un operador aritmético binario (*) que se suministra con dos operandos cuyos valores se conocen estáticamente (4 y 5, respectivamente). Sería un desperdicio para el compilador generar código que calculara este resultado en tiempo de ejecución, ya que puede deducir cuál será el resultado durante la compilación. El compilador puede simplemente reemplazar la asignación con $x = 20;$.

El desdoblamiento de constantes es una ofuscación que realiza la operación inversa. Dado un valor constante que se utiliza en algún lugar del programa de entrada, el ofuscador puede reemplazar la constante por algún proceso computacional que produzca la constante . Ya se ha encontrado con esta ofuscación en el ejemplo motivador:

01: empuje 0F9CBE47Ah
02: agregar dword ptr [esp], 6341B86h

Despreciando las modificaciones que esta secuencia tiene sobre las banderas, esto fue Se encontró que era equivalente a empujar 0h.

Esquemas de codificación de datos

El defecto fundamental de esta técnica es que las constantes deben decodificarse dinámicamente (es decir, exponerse, al igual que la función de decodificación) en tiempo de ejecución antes de ser procesadas. Tenemos la función de codificación $f(x) = x - 6341B86h$, cuyo resultado $f(x)$ se coloca en la pila y luego se aplica la función decodificada: $f^{-1}(x) = x + 6341B86h$.

Esta construcción es trivial; la desofuscación se realiza simplemente aplicando la optimización de plegado constante del compilador estándar.

Se han hecho esfuerzos para reforzar estas afirmaciones y proponer esquemas de codificación más resistentes. Algunas técnicas, como la codificación polinómica y la codificación de residuos, se han descrito en la patente US6594761 B1 de Chow, Johnson y Gu11. Los mapas afines también se utilizan comúnmente.

¿Qué sucedería si se pudiera encontrar una codificación tal que no fuera obligatorio decodificar las variables para manipularlas (se pudiera definir una operación equivalente sobre las variables codificadas)? Esta propiedad, llamada homomorfismo, ha sido discutida en una perspectiva orientada a la ofuscación, así como en un refinamiento de la técnica de codificación de residuos en trabajos como los de Zhu y Thomborson.⁴⁴

En álgebra abstracta, un homomorfismo es una aplicación que preserva las operaciones entre dos estructuras algebraicas. Consideremos, por ejemplo, dos grupos, G y H, equipados respectivamente con las operaciones $+g$ y $+h$. Queremos construir una aplicación f entre los conjuntos subyacentes a G y H, y queremos que su aplicación respete las operaciones $+g$ y $+h$. En particular, debemos tener que $f(x+g y) = f(x) + h f(y)$.

La noción de homomorfismo se puede generalizar más allá de los grupos a estructuras algebraicas arbitrarias. Por ejemplo, se pueden considerar homomorfismos de anillos que conserven simultáneamente los operadores de adición y multiplicación. A diferencia de las aplicaciones que conservan solo una de las operaciones del anillo y no la otra, o que imponen restricciones a los operadores o a su uso, las aplicaciones sin restricciones se consideran completamente homomórficas.

Las asignaciones completamente homomórficas tienen una aplicación natural en la ofuscación. Si el álgebra de origen es el dominio no codificado y el álgebra de destino es el codificado, entonces una asignación homomórfica nos permite realizar cálculos directamente sobre los datos codificados sin tener que decodificarlos de antemano y volver a codificarlos después.

En el momento de escribir este artículo, el tema de la criptografía homomórfica todavía está en pañales. Se ha demostrado que existen criptosistemas totalmente homomórficos que permiten el cálculo de programas cifrados a partir de datos cifrados. Es decir, se pueden hacer afirmaciones rigurosas sobre la dificultad de determinar detalles específicos sobre el programa que se está ejecutando y sobre qué datos está operando. En la actualidad, los esquemas son demasiado ineficientes para su uso práctico y la mejor manera de aplicar la tecnología a programas informáticos arbitrarios es una pregunta abierta.

Inserción de código muerto

Otra optimización común del compilador se conoce como eliminación de código muerto, que se encarga de eliminar las sentencias del programa que no tienen ningún efecto en el funcionamiento del programa. Por ejemplo, considere la siguiente función de C:

```
entero f()
{
    entero x, y;
```

```

x = 1;           // esta asignación a x está muerta // y no se vuelve a
y = 2; x =       usar, por lo que está muerta // x arriba no está activa
3;
devuelve x; // x está activo
}

```

Finalmente, la función devuelve el número 3. Lo hace después de varios cálculos sin sentido que no afectan el resultado de la función. Se dice que las primeras asignaciones a x e y están inactivas, ya que no tienen efecto en los cálculos en vivo.

Los ofuscadores realizan la operación inversa, insertando código muerto con el fin de hacer que el código sea más difícil de seguir: el ingeniero inverso tiene que decidir manualmente si una instrucción dada participa en el cálculo de algún resultado significativo. La capacidad de insertar código "muerto" requiere que el ofuscador sepa qué registros están "activos" en cada punto del programa; por ejemplo, si EAX contiene un valor importante (está activo) y EBX no (está muerto), entonces se pueden insertar instrucciones que modifiquen EBX.

La desofuscación de esta construcción se realiza simplemente aplicando la optimización de eliminación de declaraciones muertas del compilador estándar, lo que se puede realizar en un solo bloque básico o en todo un gráfico de flujo de control.

Sustitución aritmética mediante identidades

Se pueden hacer afirmaciones matemáticas que relacionen los resultados de ciertos operadores con los resultados de combinaciones de otros operadores. Ya ha visto un ejemplo de este fenómeno general en el ejemplo motivador, cuando se encontró con la instrucción XOR EAX, 0xFFFFFFFFh (donde la representación binaria de 0xFFFFFFFFh es todos los bits uno). Como 0 XOR 1 = 1 y 1 XOR 1 = 0, esta instrucción en realidad invierte cada uno de los bits en EAX; en otras palabras, es sinónimo del operador NOT . De manera similar, puede hacer las siguientes afirmaciones:

- $-x = \sim x + 1$ (por definición de complemento a dos) ■ rotar a la izquierda(x,y)
- $= (x << y) | (x >> (bits(x)-y))$
- rotar a la derecha(x,y) = $(x >> y) | (x << (bits(x)-y))$
- $x-1 = \sim x$
- $x+1 = -x$

Ofuscación basada en patrones

La ofuscación basada en patrones, un elemento básico de muchas protecciones contemporáneas, tiene un concepto subyacente simple. El autor de la protección construye manualmente transformaciones que asignan una o más instrucciones adyacentes a una secuencia de instrucciones más complicada que tiene el mismo efecto semántico. Por ejemplo, un patrón podría convertir la secuencia

01: empuje reg32

276 Capítulo 5 ■ Ofuscación

en esta secuencia (que llamaremos #1):

```
01: empujar imm32  
02: mov dword ptr [esp], reg32
```

O bien, podría convertir esa misma secuencia en esta secuencia (#2):

```
01: lea esp, [esp-4]  
02: mov dword ptr [esp], reg32
```

O este (#3):

```
01: subtítulo especial, 4  
02: mov dword ptr [esp], reg32
```

Los patrones pueden ser tan complejos como se desee. Un ejemplo más complejo podría sustituir el patrón:

```
01: subtítulo especial, 4
```

Para este patrón (#4):

```
01: empuje reg32  
02: movimiento reg32, esp  
03: xchg [esp], reg32  
04: pop especialmente
```

Algunas protecciones tienen cientos de patrones. La mayoría de las protecciones aplican patrones aleatoriamente a la secuencia de entrada, de modo que dos ofuscaciones del mismo fragmento de código dan como resultado una salida diferente. Además, los patrones se aplican de forma iterativa. Considere la siguiente entrada:

```
01: empujar ecx
```

Imaginemos que se transforma mediante la sustitución nº3:

```
01: subtítulo especial, 4  
02: mov dword ptr [esp], ecx
```

Ahora supongamos que el ofuscador se ejecuta una segunda vez y la primera instrucción se reemplaza de acuerdo con el patrón n.º 4:

```
01: empuje ebx  
02: movimiento ebx, esp  
03: xchg [esp], ebx  
04: pop especialmente  
05: mov dword ptr [esp], ecx
```

Este proceso se puede aplicar de forma indefinida, lo que da como resultado una secuencia de salida de cualquier tamaño. Con suficientes patrones, se puede transformar una instrucción en millones de instrucciones.

Tenga en cuenta algunas cosas sobre estas sustituciones. #1 y #2 preservan la equivalencia semántica: después de que se ejecuten esas secuencias, la CPU estará en el mismo estado.

que hubiera sido si se hubiera ejecutado el original en su lugar. #3 no conserva la equivalencia semántica, porque utiliza la subinstrucción que cambia los indicadores, mientras que el push original no lo hace. En cuanto a la secuencia #4, el original sí cambia los indicadores, mientras que la sustitución no lo hace; además, mientras que el original no modifica la memoria en absoluto, la sustitución escribe el valor de ESP en la parte inferior de la pila (por lo tanto, también podría considerar esto como igual a la instrucción PUSH ESP).

Estas consideraciones ilustran la dificultad de ofuscar el código ensamblador después de la compilación. La protección sólo es segura para ejecutar la sustitución n.º 3 si se sabe que los indicadores modificados por la instrucción no se utilizan antes de la siguiente modificación de esos indicadores. La sustitución n.º 4 es igualmente segura si los indicadores están inactivos y si el código resultante es indiferente al contenido de [ESP] después de la operación SUB ESP, 4 original . Para garantizar la actividad de los indicadores es necesario construir el gráfico de flujo de control de la función, lo que puede ser difícil debido a las ramificaciones indirectas. Para garantizar que la modificación de la memoria de pila sea segura sería extremadamente difícil debido al alias de memoria. Es poco probable que estas preocupaciones específicas afecten a las funciones normales generadas por un compilador para el que se pueden generar gráficos de flujo de control, pero se espera que ilustren los peligros de aplicar transformaciones semánticamente no equivalentes al código compilado.

Debido a las complejidades de ofuscar el lenguaje ensamblador compilado, las protecciones suelen aplicar estas transformaciones al código correspondiente a la protección en sí, en lugar de al código del objetivo. De esta manera, los autores de la protección pueden garantizar que el código de entrada no tenga en cuenta aquellas transformaciones que no conserven una equivalencia semántica estricta.

La desofuscación de este tipo de ofuscación es sencilla, aunque escribir el desofuscador puede llevar mucho tiempo . Se pueden construir sustituciones de patrones inversos , que en su lugar asignan las secuencias de destino a las originales. De hecho, esto corresponde a una optimización rutinaria del compilador conocida como optimización de mirilla. Trabajos académicos, como el de Jacob et al.²⁵ o Bansal,¹ han analizado la construcción automatizada tanto de ofuscadores de patrones como de optimizadores de mirillas.

Esto nos lleva de nuevo a la cuestión de los resultados prácticos frente a los académicos.

Supongamos que estamos tratando con un ofuscador basado en patrones que contiene errores (por ejemplo, sustituciones de patrones erróneas que no preservan la equivalencia semántica).

Supongamos además que usted, como investigador de desofuscación, es consciente de los errores y puede corregirlos en el momento de la desofuscación. Esto significa que su desofuscador tampoco preservará la equivalencia semántica y, por lo tanto, es "incorrecto" en términos absolutos en lo que respecta a la transformación, pero en realidad produce resultados "correctos" con respecto al código preofuscado. ¿Deberíamos hacer la sustitución? Los partidarios de la corrección formal dirían que no; nosotros responderíamos afirmativamente.

Ofuscación basada en control

Al realizar ingeniería inversa del código generado por el compilador, los ingenieros inversos pueden confiar en la previsibilidad de las traducciones del compilador de las construcciones de flujo de control. De esta manera, pueden determinar rápidamente la estructura del flujo de control del código original a un nivel de abstracción superior al del lenguaje ensamblador. A lo largo del proceso, la ingeniería inversa se basa en una serie de suposiciones sobre cómo los compiladores generan código. En un programa compilado puro, todo el código de un bloque básico estará ubicado en la mayoría de los casos de manera secuencial (las optimizaciones intensas del compilador pueden hacer que esta premisa básica sea nula y sin valor). Los bloques relacionados temporalmente también lo estarán. Una instrucción CALL siempre corresponde a la invocación de alguna función. La instrucción RET también casi siempre significará el final de alguna función y su retorno a quien la llamó. Los saltos indirectos, como los que se usan para implementar sentencias switch, aparecen con poca frecuencia y siguen esquemas estándar.

La ofuscación basada en el control ataca estos pilares de la ingeniería inversa estándar, de una manera que complica tanto los análisis estáticos como los dinámicos. Las herramientas de análisis estático estándar hacen suposiciones similares a las de los ingenieros inversos humanos, en particular:

- La instrucción CALL solo se utiliza para invocar funciones, y una función comienza en la dirección a la que se dirige la llamada.
- La mayoría de las llamadas retornan y, si lo hacen, regresan a la ubicación inmediatamente posterior a la instrucción CALL ; las declaraciones ret y RETN connotan límites de función.
- Al encontrar un salto condicional, los desensambladores asumen que fue incluido en el código “de buena fe”, en particular que:
 - Ambos lados de la rama podrían ser tomados de manera factible.
 - El código, no los datos, se encuentra a cada lado de la rama.
- Podrán determinar fácilmente los objetivos de los saltos indirectos.
- Los saltos y llamadas indirectas solo se generarán para construcciones estándar como conmutadores e invocaciones de punteros de función.
- Todas las transferencias de control apuntan a ubicaciones de códigos, no a ubicaciones de datos.
- Las excepciones se utilizarán de formas predecibles.

Con respecto a las transferencias de control, los desensambladores asumen un modelo de “normalidad ” basado en los patrones del código compilado estándar. Crean funciones explícitamente en los destinos de llamada, las terminan en las instrucciones de retorno, continúan desensamblando después de una instrucción de llamada, recorren ambos lados de todas las ramas condicionales, asumen que todos los destinos de las ramas son código, usan la coincidencia de patrones sintácticos para resolver el esquema de salto indirecto y, en general, ignoran el flujo de control excepcional. Violar los supuestos establecidos anteriormente conduce a un desensamblaje muy deficiente. Esta es una

es una espina constante en el costado de los investigadores de la ofuscación y un tema de investigación abierto (como se discutió anteriormente) en materia de verificación.

El análisis dinámico tiene más facilidad con respecto a las transferencias de control indirectas, ya que puede seguir explícitamente el flujo de ejecución. Sin embargo, el atacante aún enfrenta preguntas que involucran la determinación de los objetivos de las transferencias indirectas y sufre la falta de localidad secuencial inducida por el llamado código espagueti. Las siguientes secciones explican en detalle lo que sucede cuando se cuestionan estas suposiciones.

Funciones de revestimiento interior y exterior

El gráfico de llamadas de un programa contiene gran parte de su lógica de alto nivel. Jugar con la noción de función puede romper algunas de las suposiciones del inversor. Es posible:

- **Funciones en línea:** el código de una subfunción se fusiona con el código de su llamador. El tamaño del código puede crecer rápidamente si la subfunción se llama varias veces.

- **Funciones de esquema:** se extrae una subparte de una función y se transforma en una función independiente y se reemplaza por una llamada a las funciones recién creadas.

La combinación de estas dos operaciones en un programa da como resultado un gráfico de llamadas degenerado sin lógica aparente. No hace falta decir que también se puede jugar con los prototipos de funciones para reordenar argumentos, agregar argumentos adicionales y falsos, etc., y contribuir a la oscuridad de la lógica.

Destrucción de la localidad secuencial y temporal

Como se ha dicho, y como lo entienden intrínsecamente quienes realizan ingeniería inversa de código compilado, las instrucciones dentro de un único bloque básico compilado se encuentran en una secuencia de línea recta. Esta propiedad se denomina localidad secuencial. Además, los optimizadores de compiladores intentan poner bloques básicos que están relacionados entre sí (por ejemplo, un bloque y sus sucesores) cerca, con el fin de maximizar la localidad de la caché de instrucciones y reducir el número de ramas en la salida compilada. Llamamos a esta propiedad la localidad secuencial del código relacionado temporalmente. Cuando se realiza ingeniería inversa de código compilado, estas propiedades suelen ser verdaderas. Uno aprende al analizar dicho código que todo el código responsable de una sola unidad de funcionalidad estará perfectamente contenido en una sola región, y que los vecinos del flujo de control próximos estarán cerca y ubicados secuencialmente de manera similar.

Una técnica muy antigua de ofuscación de programas consiste en introducir ramificaciones incondicionales para destruir este aspecto de familiaridad que los ingenieros inversos obtienen de manera orgánica a través de esfuerzos típicos. A continuación se muestra un ejemplo sencillo:

```
01: instrucción_1:  
02: título de desplazamiento de empuje  
03:     instrucción jmp_4
```

280 Capítulo 5 ■ Ofuscación

```

04;
05: instrucción_2;
06: llamar a MessageBoxA
07:     instrucción jmp_5
08:
09: instrucción 3:
10: empujar 0 jmp instr_2
11:
12:
13: inicio:
14:     Empujar
15:     0 jmp instr_1
16:
17: instrucción_4:
18: desplazamiento de inserción digtxt
19:     instrucción jmp_3
20:
21: instrucción 5:
22:; ...

```

Este ejemplo muestra la falta de localidad secuencial para las instrucciones dentro de un bloque básico, y no la localidad temporal de múltiples bloques básicos. En la práctica, grandes cantidades del código del programa se entrelazarán de esa manera (normalmente con más de una instrucción en un bloque básico determinado, a diferencia del ejemplo anterior).

Desde una perspectiva formal, esta técnica ni siquiera merece ser llamada “trivial”, ya que no tiene ningún efecto semántico en el programa. Construir un gráfico de flujo de control y eliminar las ramas incondicionales espurias derrotará por completo este esquema. Sin embargo, en términos de análisis realizado manualmente por un humano, la capacidad de seguir el código se ha ralentizado drásticamente.

Indirección de control basada en procesador

Para la mayoría de los procesadores, dos primitivas de desplazamiento esenciales son la bifurcación de tipo JMP y el puntero y la bifurcación de instrucción de guardado de tipo CALL. Estas primitivas se pueden ofuscar utilizando direcciones de bifurcación calculadas dinámicamente o emulándolas. Una de las técnicas más básicas es la pareja PUSH-RET. utilizado como instrucción JMP :

```

01: introducir la dirección de destino
02: ret

```

Esto es (casi) semánticamente equivalente a lo siguiente:

```

01: dirección_objetivo jmp

```

La instrucción CALL es un blanco fácil para los ofuscadores porque la mayoría de las deshabilitaciones Los ensambladores asumen lo siguiente acerca de su semántica de alto nivel:

- La dirección de destino es un punto de entrada de subfunción.
- Una llamada retorna (es decir, la instrucción después de que se ejecuta la LLAMADA).

En realidad, es fácil romper con estas suposiciones. Consideremos el siguiente ejemplo:

```
01: llamar a target_addr 02:  
< código basura > 03:  
target_addr: 04: agregar  
esp, 4
```

La llamada se utiliza como un JMP; nunca regresará a la línea 2. La pila se fija (la dirección de retorno se descarta de la pila) en la línea 3. A continuación, considere estos dos elementos:

```
01: bloque_básico_a:  
02: añadir [esp], 9  
03: ret
```

y

```
01: bloque_básico_b:  
02: llamar al bloque básico a  
03: < código basura >  
04: dirección de retorno verdadera:  
05: no
```

La instrucción CALL de la línea 2 de basic_block_b apunta a basic_block_a, que en realidad es sólo un trozo que actualiza (ver la línea 2 de basic_block_a) la dirección de retorno almacenada en la parte superior de la pila antes de que la instrucción RET la utilice (línea 3 de basic_block_a). En estos dos ejemplos, el resultado es un intervalo entre las direcciones de retorno naturales (esperadas) y efectivas de CALL ; un ofuscador puede (y lo hará) aprovecharse para insertar código que frustre a los desensambladores y genere confusión.

El siguiente ejemplo es un Enriquecimiento interesante del estándar PUSH-RET

utilizado como JMP anteriormente:

```
01: enviar addr_branch_default  
02: empuje ebx  
03: empujar edx  
04: movimiento ebx, [esp+8]  
05: movimiento edx, addr_branch_jmp  
06: cmovz ebx, edx  
07: movimiento [esp+8], ebx  
08: educación pop  
09: pop-ebx  
10: ret
```

La base de esta construcción es en realidad un PUSH-RET. La línea 7 escribe la dirección de destino en la pila; la utiliza el RET en la línea 10. La dirección insertada proviene de EBX (línea 7), que se actualiza condicionalmente mediante la instrucción CMOVZX en la línea 6. Si se cumple la condición (se prueba el indicador Z), la instrucción actúa como un MOV estándar (EBX se sobrescribe con EDX, que contiene la dirección de destino de la bifurcación); de lo contrario, actúa como un NOP (por lo tanto, EBX contiene la dirección de bifurcación predeterminada).

Al final, se puede ver claramente que este patrón representa un salto condicional (JZ).

Indirección de control basada en el sistema operativo

El programa puede hacer uso de primitivas del sistema operativo (aunque esto pueda implicar una pérdida de portabilidad). El manejador de excepciones estructurado (SEH), el manejador de excepciones vectorizado (VEH) y el manejador de excepciones no controladas, en Windows, y los manejadores de señales y las funciones setjmp/longjmp , en Unix, se utilizan comúnmente para ofuscar el flujo de control.

El algoritmo básico se puede descomponer de la siguiente manera:

1. El código ofuscado desencadena una excepción (utilizando un puntero no válido, una operación no válida, una instrucción no válida, etc.).
2. El sistema operativo llama a los controladores de excepciones registrados.
3. El controlador de excepciones envía el flujo de instrucciones de acuerdo con su lógica interna y devuelve el programa a un estado limpio.

El siguiente ejemplo se ha visto miles de millones de veces en binarios x86:

```
01: empuja la dirección_se_manejador_de_la_dirección
02: empujar fs:[0]
03: movimiento fs:[0], esp
04: xor eax, eax
05: movimiento [eax], 1234h
06: <código basura>
07: controlador de dirección_seh:
08: <continuar la ejecución aquí>
09: grupo de personas:[0]
10: añadir esp, 4
```

Las líneas 1 a 3 configuran el SEH. Luego se activa una excepción en forma de violación de acceso cuando la línea 5 intenta escribir en 0x0. Suponiendo que el programa no esté depurado, el sistema operativo transferirá la ejecución al controlador SEH . Tenga en cuenta también que cuando se llama a un controlador SEH , este recibe una copia del contexto del hilo como uno de sus argumentos y el valor del registro del puntero de instrucción se puede modificar para ofuscar aún más la redirección del flujo de control.

NOTA: Esta técnica también actúa de manera eficiente como un antidepurador. Básicamente, el trabajo de un depurador es manejar excepciones. Estas excepciones deben pasarse al objetivo de depuración; de lo contrario, se modificará el comportamiento del objetivo y se detectarán las alteraciones.

Más interesante aún, el concepto también puede invertirse. ¿Qué sucede si una protección inserta excepciones en el programa original y las captura con su propio depurador adjunto? El programa protegido consta de un depurador y un depurador. Un ejemplo bien conocido de esto es la característica namomites de Armadillo. Los namomites realmente reemplazan los saltos (condicionales) por la instrucción INT 3. La excepción es capturada por el depurador de la protección, que actualiza el contexto del depurado de manera apropiada para emular los saltos (condicionales). No se puede simplemente separar el depurador del depurado; de lo contrario, las excepciones no se manejarían y el programa se bloquearía. Deroko ha propuesto una implementación de este concepto.¹⁹

Predicados opacos

Un predicado opaco (introducido por Collberg en "A Taxonomy of Ofuscating Transformations"¹² y "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs"¹³) es una construcción condicional especial (expresión booleana) que siempre se evalúa como verdadera o falsa (respectivamente anotadas como PT y PF). Su valor se conoce solo en el momento de la compilación/ofuscación y debería ser desconocido para un atacante, así como computacionalmente difícil de probar, para cumplir con un grado suficiente de resiliencia. Utilizado en combinación con una instrucción de salto condicional, introduce una rama espuria adicional, es decir, un borde adicional en el gráfico de flujo de control (CFG). Esta rama muerta se puede utilizar para insertar código basura o propiedades especiales como ciclos en el CFG para reforzar el análisis. Sin embargo, la rama espuria tiene que parecer lo suficientemente real como para escapar a la detección simple por parte de un atacante humano (por ejemplo, solo una de las dos ramas contiene inicializaciones de variables necesarias).

Tiene la apariencia de un salto condicional pero su semántica es la de un salto incondicional. Se pueden utilizar problemas matemáticos computacionalmente complejos para implementar predicados opacos. También se pueden utilizar algunas variables ambientales cuyos valores sean constantes y conocidos en el momento de la compilación/ofuscación. Esta última técnica puede ser menos resistente porque hay un conjunto limitado y finito de variables candidatas, lo que limita la diversidad potencial.

Diseñar predicados opacos resistentes es una tarea difícil. Son fragmentos superfluos de código mezclados con código existente que tiene su propia lógica/estilo; si no se toma un cuidado especial, son fácilmente detectables. Una buena práctica es crear dependencias entre el predicado y el estado/las variables del programa. Un atacante humano (usted) suele ser bastante eficiente en la detección de patrones dudosos. El uso de un predicado absurdamente complejo puede frustrar eficazmente una herramienta de análisis estático, pero probablemente un atacante humano lo detectará fácilmente.

Una variación interesante del concepto original utiliza un predicado que devuelve aleatoriamente verdadero o falso (observado como P?). Como ambas ramas se ejecutan potencialmente en tiempo de ejecución, deben ser semánticamente equivalentes. En la mayoría de los casos, eso equivale a clonar (y posiblemente diversificar) un bloque básico (o un fragmento de código más grande), lo que produce una construcción "similar a un diamante".

Ofuscación simultánea del flujo de control y del flujo de datos

Para mayor claridad, hasta ahora hemos disociado la ofuscación del flujo de control y del flujo de datos. Sin embargo, en la práctica, ambos están íntimamente relacionados. En esta sección se presentan técnicas basadas en esta interacción.

Inserción de código basura

Esta técnica está íntimamente ligada a la ofuscación del flujo de control. Básicamente consiste en insertar un bloque de código muerto (es decir, nunca ejecutado) entre dos bloques de código válidos. El objetivo es frustrar totalmente un desensamblador que ya ha sido engañado para que siga una ruta no válida (normalmente un caso de predicados opacos). Las instrucciones contenidas en el código basura pueden ser parcialmente inválidas o pueden crear ramas a direcciones inválidas (por ejemplo, en medio de instrucciones válidas) para complicar demasiado el CFG.

El ejemplo más trivial de inserción de código basura podría ser el siguiente:

```
01: etiqueta jmp
02: <basura> 03:
etiqueta:
04: < código real >
```

Aquí hay algo un poco más elaborado, utilizando un predicado opaco ficticio:

```
01: empujar eax
02: xor eax, eax
03: 9 de julio
04: <inicio de código basura>
05: partido 4
06: inc esp
07: ret
08: <fin del código basura>
09: pop eax
```

El salto condicional en la línea 3 (dirección) siempre es verdadero porque el registro EAX se pone a cero mediante la instrucción XOR en la línea 1. Eso significa que tienes seis bytes de código basura. Este bloque basura utiliza instrucciones que influirán en el desensamblador , creando una nueva rama y aparentemente insertando un final de función (la instrucción RET en la línea 9).

Cuando se generan de forma adecuada, los bloques de código basura pueden resultar bastante difíciles de detectar a primera vista. La mayoría de las veces, se eliminan del alcance del desensamblador como un efecto secundario de la desofuscación del flujo de control (consulte http://www.openrce.org/blog/vista/1672/Desofuscación_del_flujo_de_control_a_través_de_Interpretación_Abstracta). En el último ejemplo, si el predicado opaco se detecta como tal, entonces no hay más caminos que conduzcan al bloque de código basura. Como todas las demás técnicas, si no se diferencia lo suficiente (por ejemplo, utilizando una base de datos limitada de patrones estáticos), su resistencia y fortaleza tienden a ser mínimas.

Aplanamiento de gráficos de flujo de control

La idea básica detrás del aplanamiento de grafos es reemplazar todas las estructuras de control con una única sentencia switch, conocida como el despachador. Se selecciona un subgrafo del grafo de flujo de control del programa (las implementaciones a menudo funcionan a nivel de funciones) y se transforma, momento en el cual los bloques básicos pueden ser reelaborados (divididos o fusionados). Cada bloque básico es entonces responsable de actualizar el contexto del despachador (es decir, el estado del subprograma) de modo que el despachador pueda vincularse al siguiente bloque básico (ver Figura 5-1). Las relaciones entre bloques básicos ahora están "ocultas" dentro de las operaciones de manipulación del contexto del despachador. Los saltos condicionales (como en el bloque d) pueden ser fácilmente emulados usando pruebas de indicadores e instrucciones IMUL, o instrucciones CMOV simples.



Figura 5-1

No hace falta decir que gran parte de la resistencia de esta técnica frente al análisis estático se basa en la capacidad de ocultar las manipulaciones y transiciones del contexto. Se pueden implementar varias funciones para complicar el problema, como las relaciones entre procedimientos, el alias de punteros, la inserción de estados ficticios, etc.

De la misma manera que los predicados opacos, el aplanamiento de CFG también se puede utilizar para insertar rutas de código inactivas y bloques básicos espurios. Se puede decir mucho sobre el aplanamiento de grafos y cómo reforzar una implementación. El grafo resultante no ofrece pistas sobre la estructura del algoritmo, y el código de manipulación de contexto y de envío también agrega una sobrecarga que contribuye a ocultar el código protegido. Esta técnica es conceptualmente la misma que la virtualización de código (virtual

máquina); puede verse como una virtualización parcial que apunta (virtualiza) solo el flujo de control (no el flujo de datos).

Si desea ver el código aplanado, simplemente obtenga una copia de un complemento de Flash (como NPSWF32.dll), desmonte el archivo y busque las funciones con el tamaño más grande. Las funciones aplanadas son fácilmente reconocibles.

Máquinas virtuales

Las máquinas virtuales (VM) son una clase potente de protección de software y una transformación especialmente compleja. Una VM consta básicamente de un intérprete y un código de bytes. El lenguaje admitido por el intérprete queda a discreción de la protección. En tiempo de compilación, se compilan partes seleccionadas del código con respecto a la arquitectura de destino de la VM (se redirigen) y luego se insertan en el programa protegido junto con el intérprete asociado. En tiempo de ejecución, el intérprete asume la ejecución del código de bytes (es decir, la traducción de la arquitectura de destino a la arquitectura original). Las VM suelen tener una sobrecarga considerable en términos de rendimiento (en particular, tiempo de CPU), por lo que normalmente solo se virtualizan partes específicas y seleccionadas del programa original.

Entre los ejemplos de protecciones conocidas centradas en las máquinas virtuales se incluyen VMProtect y CodeVirtualizer. Más adelante profundizaremos en la agradable actividad del análisis de máquinas virtuales. Por ahora, basta decir que un atacante tiene que entender al intérprete para poder analizar el código de bytes y eventualmente crear un compilador desde la arquitectura de destino a la arquitectura nativa (desvirtualización).

Criptografía de caja blanca

Cuando la aplicación a proteger no puede basar su seguridad en el uso de un componente de hardware o en un servidor de red, se debe plantear la hipótesis de un atacante capaz de ejecutar la aplicación en un entorno que controla perfectamente . El modelo de atacante que se corresponde con esta situación, denominado contexto de ataque de caja blanca (WBAC), impone una implementación de software particular de primitivas criptográficas clásicas .

Estos mecanismos están diseñados a medida para garantizar la confidencialidad de una clave secreta dentro de un algoritmo. Una transformación de este tipo (ocultar una clave en un algoritmo de cifrado, con o sin la ayuda de la interacción con el entorno) se puede formalizar como una transformación de ofuscación.

En esta sección se describen algunos resultados negativos y positivos relacionados con el código. ofuscación y su impacto en este problema clave de gestión.

Un algoritmo probabilístico O es un ofuscador si satisface las siguientes propiedades , dadas por Barak et al.2 :

- P y O(P) calculan la misma función.
- El crecimiento del tiempo y el espacio de ejecución de O(P) es como máximo polinomial en respecto al tiempo y espacio de ejecución del programa P.

- Para cualquier algoritmo probabilístico de tiempo polinomial A, existe un algoritmo probabilístico de tiempo polinomial S y una función despreciable m (una función despreciable es una función que crece mucho más lentamente que la inversa de cualquier polinomio), como la siguiente: para todos los programas P,

$$| p[A(O(P))=1] - p[SP(1|P)|=1] | \leq m(|P|)$$

La propiedad de caja negra virtual expresa el hecho de que la distribución de salidas de cualquier algoritmo de análisis probabilístico A aplicado al programa ofuscado O(P) es casi en todas partes igual a la distribución de salidas de un simulador S que hace acceso de oráculo al programa P. (El programa S no tiene acceso a la descripción del programa P, pero para cualquier entrada x, se le da acceso a P(x) en tiempo polinomial con respecto al tamaño de P. Un acceso de oráculo al programa P es equivalente a un acceso únicamente a las entradas/salidas del programa P.)

Intuitivamente, la propiedad de caja negra virtual simplemente estipula que todo lo que se puede calcular a partir de la versión ofuscada O(P) también se puede calcular a través del acceso del oráculo a P.

Uno de los puntos principales de un ofuscador ideal es que no existe. La prueba se basa en la construcción de un programa que no puede ser ofuscado. Este resultado de imposibilidad demuestra que no existe un generador de caja negra virtual (que podría proteger el código de cualquier programa impidiéndole revelar más información que la revelada por sus entradas/salidas). Este resultado de imposibilidad conduce naturalmente a resultados importantes para los diseñadores de mecanismos de ofuscación (adaptados al contexto WBAC).

Consideremos una aplicación práctica de la ofuscación que consiste en transformar un cifrado simétrico en un cifrado asimétrico, mediante la ofuscación del esquema de cifrado de clave privada. Un esquema de cifrado de clave privada que no se pueda ofuscar existe si existe un esquema de cifrado de clave privada. Esto indica claramente que no todos los esquemas de cifrado de clave privada son adecuados para la ofuscación.

Obsérvese que este resultado no prueba que no exista algún esquema de cifrado de clave privada que nos permita proporcionar al atacante un circuito que calcule el algoritmo de cifrado sin pérdida de seguridad. Sin embargo, sí prueba que no existe un método general que permita transformar cualquier esquema de cifrado de clave privada en un sistema de cifrado de clave pública mediante la ofuscación del algoritmo de cifrado.

El problema de construir un esquema de cifrado de clave privada que verifique la propiedad de la caja negra virtual (y por lo tanto resistente en el contexto WBAC) sigue siendo de interés para los investigadores de criptografía, incluso si el resultado de la imposibilidad de una forma genérica de gestionarlo puede parecer desalentador. Las propuestas de implementación de DES y AES de caja blanca ilustran este interés.

La ofuscación mediante el uso de una red de tablas de búsqueda codificadas permite obtener versiones de los algoritmos DES y AES que son más resistentes en el contexto de ataques de caja blanca. Sin embargo, el criptoanálisis efectivo de las implementaciones de caja blanca de DES (como el realizado por Goubin24) y AES (por Billet5) ha

estableció que el problema de construir un esquema de cifrado de clave privada que verifique la propiedad de la caja negra virtual sigue sin resolverse.

El modelo ideal de un ofuscador capaz de transformar cualquier programa en una caja negra virtual no se puede implementar. En particular, no existe ninguna transformación general que permita, a partir de un algoritmo de cifrado y una clave, obtener una versión ofuscada de ese algoritmo que pueda publicarse sin filtrar información sobre la clave que contiene.

Sin embargo, este formalismo no establece que sea imposible ocultar una clave en un algoritmo para transformar un algoritmo de clave privada en un cifrado de clave pública.

Se ha publicado un método (por Chow9) para dificultar la extracción de la clave en el contexto de caja blanca. El principio es implementar una versión especializada del algoritmo DES que incorpora la clave K y que es capaz de realizar solo una de las dos operaciones, cifrar o descifrar. Esta implementación es resiliente en un contexto de caja blanca porque es difícil extraer la clave K observando las operaciones realizadas por el programa y porque es difícil falsificar la función de descifrado a partir de la implementación de la función de cifrado, y viceversa.

La idea principal es expresar el algoritmo como una secuencia (o red) de tablas de búsqueda y ofuscar estas tablas codificando su entrada/salida. Todas las operaciones del cifrador de bloques, como la adición módulo 2 de la clave circular, están incrustadas en estas tablas de búsqueda. Estas tablas están aleatorizadas para ofuscar su funcionamiento.

La ofuscación de AES (descrita por Chow10) se realiza de manera similar a DES. El objetivo sigue siendo incrustar las claves circulares en el código del algoritmo, para evitar almacenar la clave en la memoria estática o cargarla en la memoria dinámica en el momento de la ejecución. La técnica utilizada para incrustar de forma segura estas claves es (como en el caso de DES) representar AES como una red de tablas de búsqueda y aplicar codificaciones de entrada/salida para ocultar las claves.

Lograr la seguridad mediante la oscuridad

Hasta ahora, ha visto una gran cantidad de técnicas de ofuscación. La mayoría de ellas son transformaciones simples que parecen bastante débiles a primera vista, y en realidad son débiles consideradas individualmente. ¿Cómo se puede generar seguridad o confianza a partir de tales primitivas? La fortaleza de un sistema de ofuscación (u ofuscador) proviene de las aplicaciones iterativas y combinadas de un conjunto de estas técnicas. Cada aplicación sucesiva de una técnica simple se acumula en una fuerte transformación global indiscernible (bueno, al menos ese es el objetivo). Jakubowski et al.²⁶ propusieron una analogía interesante entre la criptografía basada en rondas y la ofuscación iterada. La ronda de un algoritmo criptográfico está compuesta por operaciones aritméticas básicas (suma, o exclusiva, etc.) que realizan transformaciones triviales en las entradas. Considerada individualmente, una ronda es débil y propensa a

Existen múltiples formas de ataques. Sin embargo, aplicar un conjunto de rondas varias veces puede dar como resultado un algoritmo relativamente seguro. Ese es el objetivo de un ofuscador. **El objetivo del atacante es discernir las rondas de la forma global ofuscada y atacarlas en sus puntos más débiles.**

Tenga en cuenta que, incluso si el ofuscador no es perfecto, tan pronto como eleve el nivel necesario para acceder al código protegido en una cantidad suficiente, esto puede ser suficiente para el defensor. Por ejemplo, si se requieren algunas semanas o meses para acceder a una nueva versión de software, el defensor puede aprovechar ese período para trabajar en nuevas protecciones, actualizaciones de protocolos, etc., y así estar siempre un paso por delante.

Un estudio de las técnicas de desofuscación

Ahora que comprende mejor la ofuscación de código, la pregunta es ¿cómo puede usted, como ingeniero inverso, aceptar el desafío? ¿Qué medios y herramientas tiene a su disposición para acceder al código ofuscado? El análisis manual del código ofuscado es una tarea tediosa, si no imposible; querrá reducir el problema a un análisis de código limpio.

Debido a que un enfoque manual utilizando herramientas de análisis de programas estándar es tedioso , y considerando la amplia variedad de mecanismos de ofuscación a los que un analista puede enfrentarse, es necesario encontrar algunos modelos y criterios para diseñar y evaluar algoritmos de desofuscación. Esta sección proporciona una breve descripción general del problema desde una perspectiva más teórica y describe algunos métodos formales bien estudiados que se pueden utilizar para diseñar herramientas de desofuscación más genéricas y automatizar tanto como sea posible las tareas realizadas por un analista.

La naturaleza de la desofuscación: inversión de la transformación

Para deshacer la transformación de la ofuscación, existen varias técnicas de análisis de software. Esta sección cubre lo siguiente:

- La noción de aproximación decidable
- Algunos métodos, ya sean estáticos o dinámicos, que se pueden utilizar y ventajas que se pueden obtener de los métodos híbridos estático-dinámicos (algunos de ellos se presentan más adelante mediante el uso de herramientas especializadas)
- Algunos criterios que siempre se pueden aplicar para evaluar un algoritmo de análisis y de los cuales es posible derivar algunos criterios de seguridad sobre la robustez de la ofuscación (y de doble vía una eficiencia de transformación de desofuscación)

- Problemas abiertos y nuevas tendencias en relación con el análisis híbrido dinámico/estático y la formalización de la desofuscación

El tema es amplio y aún no existe un consenso sobre la terminología para las diversas áreas especializadas de investigación en la literatura. El objetivo de esta sección es, por tanto, proporcionar a los lectores algunas palabras clave que permitan una visión global y algunas referencias útiles para los lectores interesados que quieran complementar sus conocimientos en este ámbito.

En el campo del análisis de software se pueden observar varias dicotomías. Algunas técnicas de análisis se describen como estáticas o dinámicas, aunque a veces esta distinción parezca bastante artificial (esta distinción la analizan Yannis Smaragdakis y Christoph Csallner³⁸). Por lo demás, los algoritmos de análisis se califican de sólidos, o completos, pero estas características importantes pueden tener diferentes significados en la literatura. Finalmente, los análisis de programas se describen como sobreaproximación o subaproximación, pero esta distinción también parece algo artificial porque algunos métodos de análisis parecen utilizar tanto sobreaproximación como subaproximación.

El resto de esta sección analiza tanto la “sinergia” como la “dualidad” del análisis estático y dinámico (también analizado por Michael D. Ernst²⁰), primero introduciendo el modelo formal de interpretación abstracta y luego proporcionando varios ejemplos de análisis en relación con la desofuscación.

Encontrar una aproximación decidible de la semántica concreta

El propósito de cualquier análisis de programas es verificar si el programa satisface una determinada propiedad. Desafortunadamente, la cuestión es generalmente indecidible para cualquier propiedad no trivial, es decir, no se puede diseñar un algoritmo para determinar si la propiedad se cumple para el programa. Para superar esta dificultad, una solución es abstraer los comportamientos concretos del programa en una aproximación decidible. El propósito de la interpretación abstracta es formalizar esta idea de aproximación en un marco unificado. (Los lectores pueden consultar el artículo de Patrick Cousot y Radia Cousot.¹⁴)

La semántica de un programa representa todo su posible comportamiento concreto, incluida su interacción con cualquier entorno de sistema informático posible. Entre las semánticas más precisas (concretas) se encuentra la llamada semántica de trazas. Esta semántica incluye todas las secuencias finitas e infinitas de estados y transiciones . Donde X es el conjunto de trazas de ejecución (finitas e infinitas), se puede expresar la semántica de trazas como la solución mínima (para el ordenamiento parcial computacional) de una ecuación de punto fijo $X=F(X)$.

Un dominio abstracto es una abstracción de una semántica concreta. El objetivo de la interpretación abstracta es proporcionar aproximaciones computables y precisas de dominios abstractos, definiendo así una semántica abstracta computable. Obviamente, cuanto más burda sea la semántica abstracta, menos preguntas podrá responder.

Todas las abstracciones de una semántica pueden organizarse en una jerarquía (descrita por Cousot¹⁶), desde la más precisa a la más burda. Más precisamente, las semánticas abstractas pueden colocarse en una red, y la ordenación parcial aproximada de esta red

se puede utilizar para caracterizar la concreción (o precisión) de la semántica abstracta y, por tanto, los conjuntos de preguntas que pueden responder.

La interpretación abstracta se aplica generalmente al análisis estático, a través de una sobreaproximación de la semántica concreta. Se puede observar que, de manera dual y según el "principio dual" de la teoría de retículas, también debería aplicarse al análisis dinámico, aunque actualmente no haya muchos trabajos sobre este tema.

Verá en la siguiente sección que las relaciones y la sinergia entre el análisis estático y dinámico conducen a métodos híbridos dinámicos/estáticos prácticos, lo que hace posible que un enfoque dinámico gane en cobertura y que un enfoque estático gane en precisión.

Los análisis dinámicos y estáticos forman un continuo

El análisis estático es la disciplina que infiere automáticamente información sobre programas informáticos sin ejecutarlos (por lo tanto, se aplica a una representación "estática" del programa). El análisis estático intenta derivar propiedades (invariantes) que se cumplan para todas las ejecuciones del programa, mediante una sobreaproximación conservadora de su semántica concreta.

Un ejemplo de este tipo de análisis estático es el algoritmo de propagación de constantes , que tiene como objetivo determinar para cada instrucción del programa si una variable tiene un valor constante siempre que el flujo de control llegue a esa instrucción. La información sobre las constantes es útil en el contexto de la compilación, optimización y recompilación de programas. Se utiliza, por ejemplo, para la eliminación de código inactivo y rutas de ejecución inactivas (al reemplazar todos los usos de variables constantes por sus valores constantes, es posible que pueda identificar ramas condicionales constantes, que están condicionadas por predicados constantes).

Entre las muchas técnicas de optimización, las técnicas de evaluación parcial (descritas por Beckman et al.3) deben tenerse en cuenta en el contexto de la ingeniería inversa. Un evaluador parcial especializa un programa con respecto a una parte de sus datos de entrada. Se espera que la semántica concreta del programa se conserve mediante el proceso de especialización y que la representación sintáctica del programa resultante esté optimizada para la clase de datos de entrada utilizados y, como resultado, sea más sencilla de entender.

Otra clase importante de técnicas de optimización incluye las técnicas de segmentación (descritas por Weiser42), que también tienen como objetivo simplificar el programa en cuestión, pero en este caso eliminando aquellas partes del programa que son irrelevantes según un criterio proporcionado por el analista. Un criterio de segmentación estático incluye un conjunto de variables y un punto de interés elegido. Un criterio de segmentación dinámico completa un criterio estático con la información correspondiente a alguna ejecución concreta. La segmentación es de gran interés en el contexto de la ingeniería inversa, porque es representativa de la forma en que un inversor segmenta mentalmente un programa cuando intenta comprender su funcionamiento interno.

A diferencia del análisis estático, el análisis dinámico es la disciplina que infiere automáticamente información sobre un programa informático en ejecución. El análisis dinámico deriva propiedades que se cumplen para una o más ejecuciones de un programa mediante una subaproximación precisa.

Un método común de análisis dinámico es la prueba dinámica, que ejecuta un programa con varias entradas y verifica la respuesta del programa. Por lo general, los casos de prueba exploran solo un subconjunto de las posibles ejecuciones del programa.

Para ampliar la cobertura de las pruebas dinámicas, el principio de ejecución simbólica (descrito por Boyer⁶) utiliza valores simbólicos en lugar de entradas concretas. En cualquier punto durante la ejecución simbólica, se actualiza un estado simbólico del programa. Este estado simbólico consta de un almacén simbólico y una restricción de ruta.

El almacén simbólico contiene los valores simbólicos y la restricción de ruta es una fórmula que registra el historial de todas las ramas condicionales tomadas hasta el instrucción actual.

En una instrucción dada del programa, puede utilizar un solucionador de restricciones (solucionador SMT o SAT) para determinar la restricción de ruta correspondiente. Una asignación satisfactoria proporciona entradas concretas con las que el programa llega a la instrucción del programa. Al generar nuevas pruebas y explorar nuevas rutas, puede aumentar la cobertura de las pruebas dinámicas.

Lamentablemente, las restricciones generadas durante la ejecución simbólica pueden ser demasiado complejas para el solucionador de restricciones. Si el solucionador de restricciones no puede calcular una asignación satisfactoria, no podrá determinar si una ruta es factible o no.

La ejecución concólica (descrita por Godefroid²³ y Sen³⁷) proporciona una solución a este problema en muchas situaciones. La idea es realizar tanto la ejecución simbólica como la ejecución concreta de un programa. Cuando la restricción de ruta es demasiado compleja para el solucionador de restricciones, puede utilizar la información concreta para simplificar la restricción (normalmente, reemplazando algunos de los valores simbólicos por valores concretos). A continuación, puede esperar encontrar una asignación satisfactoria de esta restricción simplificada.

Debido a que la ejecución simbólica no puede manejar un bucle sin límites, lo que da como resultado rutas de ejecución simbólica infinitas, debe aproximarse por debajo de la semántica concreta del programa. Puede realizar esta simplificación fijando un límite de bucle arbitrario. Otra solución es utilizar la ejecución simbólica junto con un análisis estático que infiera invariantes de bucle.

Parece que los enfoques de análisis dinámico y estático forman un continuo. A modo de ejemplo, las pruebas dinámicas, la ejecución simbólica y la interpretación abstracta son tres formas de aproximarse a la semántica concreta de un programa. El análisis dinámico utiliza valores concretos y explora un subconjunto de transiciones concretas. La ejecución simbólica se sitúa claramente entre las pruebas dinámicas y el análisis estático. Se basa en una semántica más abstracta, pero también en una subaproximación. Un intérprete abstracto sobreaproxima la semántica concreta del programa.

Sin embargo, la frontera entre esos enfoques de análisis no es tan fácil de definir. Por ejemplo, la ejecución simbólica puede definirse como un intérprete lógico abstracto que opera sobre el dominio abstracto de las fórmulas lógicas.

En conclusión, muchos métodos de análisis estático se mejoran mediante el uso de un refinamiento basado en el análisis dinámico. Por el contrario, la cobertura de muchos métodos de análisis dinámico se puede aumentar mediante el uso de métodos de análisis estático tradicionales. Por ello, la investigación de enfoques híbridos dinámicos/estáticos resulta de gran interés, especialmente en el contexto de la ingeniería inversa. Los criterios de solidez y completitud pueden utilizarse para captar esta sinergia.

Solidez y completitud

Cualquier problema de análisis de programas se puede formular como una verificación de que el programa satisface una propiedad. Se pueden utilizar dos conceptos fundamentales para caracterizar un algoritmo de análisis: su solidez y su completitud. Estos conceptos, que tradicionalmente se aplican a los sistemas lógicos, también se pueden aplicar al análisis de programas. Desafortunadamente, debido a su naturaleza dual (solidez y completitud corresponden a implicaciones inversas en lógica), aún no hay consenso respecto de su aplicación a las diversas áreas especializadas de investigación en la literatura.

Dada una propiedad, un análisis de un programa de sonido identifica todas las violaciones de la propiedad. Sin embargo, debido a que se aproxima demasiado a los comportamientos del programa, también puede informar violaciones de la propiedad que no pueden ocurrir. Por ejemplo, un algoritmo de detección de errores de sonido detecta todos los errores posibles, aunque algunos de ellos pueden no ocurrir en tiempo de ejecución.

Un buen algoritmo de evaluación parcial preserva la semántica concreta del programa original, en el sentido de que el programa especializado no produce ningún valor de salida que no sea producido por el programa original (aunque no pueda producirlos todos).

Una ejecución simbólica sólida garantiza que, debido a que una ruta de restricción simbólica es satisfacible, debe haber una ruta de ejecución concreta que alcance el estado concreto correspondiente (incluso si algún estado concreto alcanzable no tiene un estado simbólico correspondiente).

Un buen intérprete abstracto preserva la semántica concreta del programa. Si afirma que es posible una transformación de optimización para un programa, entonces la optimización se puede aplicar sin romper la semántica del programa. Sin embargo, observe que puede ser incapaz de responder la pregunta para algunas optimizaciones. Puede afirmarse que una optimización no es segura incluso si de hecho es posible aplicar la transformación (sin ningún efecto destructivo). Algunas optimizaciones potenciales no se aplicarán. La solidez del intérprete abstracto es relativa a las preguntas que puede responder correctamente, a pesar de la pérdida de información. En ese sentido, es conservador. Técnicamente, los puntos fijos mínimos calculados por un intérprete abstracto representan al menos todos los estados concretos que ocurren en tiempo de ejecución.

Por ejemplo, un algoritmo de propagación constante es válido cuando cualquier constante que detecte es, en efecto, una constante. Sin embargo, es posible que algunas constantes no se detecten. Dada una propiedad, un algoritmo de análisis completo informa una violación de la propiedad solo si hay una violación concreta de la propiedad. Sin embargo, debido a que no se aproxima lo suficiente a los comportamientos del programa, es posible que no se informen algunas violaciones concretas de la propiedad.

Un algoritmo de evaluación parcial completo genera un programa especializado que puede producir los mismos valores de salida que el programa original para los valores de entrada previstos. Si no es correcto, puede producir valores de salida inesperados (es decir, no producidos por el programa original).

Una ejecución simbólica completa cubre todas las transiciones concretas. Garantiza que si se puede alcanzar un estado de ejecución concreto, entonces debe haber un estado simbólico correspondiente. Debido a que la ejecución simbólica no puede manejar un bucle sin límites, lo que da como resultado rutas de ejecución simbólica infinitas, debe aproximarse a la semántica concreta del programa (normalmente proporcionando algún límite de bucle).

Por lo tanto, los algoritmos de ejecución simbólica suelen ser incompletos.

Un intérprete abstracto completo es el más preciso para responder a un conjunto determinado de preguntas. Técnicamente, esto significa que cada estado representado por el punto fijo mínimo es alcanzable para una entrada concreta. Por ejemplo, un algoritmo de propagación de constantes completo sería capaz de detectar todas las constantes de un programa.

Hemos presentado algunos criterios (solidez y completitud) que siempre se pueden aplicar para evaluar un algoritmo de análisis. Es posible derivar de ellos algunos criterios de seguridad sobre la robustez de la ofuscación (y, de doble vía, la eficiencia de la transformación de desofuscación).

La interpretación abstracta se puede utilizar para modelar cualquier transformación de un programa (consulte el artículo de Patrick y Radia Cousot¹⁵). Al considerar la sintaxis de un programa como una abstracción de su semántica concreta, podemos formalizar cualquier transformación sintáctica de un programa como una interpretación abstracta de la transformación semántica correspondiente.

Una aplicación particular de esto se refiere al modelado de transformaciones de ofuscación y desofuscación. Mila Dalla Preda y Roberto Giacobazzi¹⁸ investigan las transformaciones semánticas correspondientes a la inserción de predicados opacos. Al modelar la desofuscación como una interpretación de abstracción, observan que la ruptura de predicados opacos corresponde a una abstracción completa. El criterio de completitud resulta de especial interés en términos de calificar tanto la efectividad del desofuscador como la robustez de los predicados opacos.

En conclusión, muchos métodos ya utilizados en el análisis y compilación de programas son de interés en el contexto de la ingeniería inversa. Como se demostró anteriormente, la frontera entre el análisis estático y dinámico no es tan obvia. Actualmente, el modelo de interpretación abstracta parece ser lo suficientemente general como para aplicarse a ambos tipos de análisis. Los criterios de solidez y completitud son de especial interés cuando se modelan transformaciones de ofuscación y desofuscación en el

Marco de interpretación abstracta. Has visto que tanto la solidez como la completitud de un algoritmo se pueden definir para análisis estáticos y dinámicos (análisis de flujo de datos, evaluación parcial, segmentación, ejecución simbólica), que son buenos candidatos para representar las acciones que llevan a cabo los inversores cuando intentan simplificar la representación de un programa ofuscado. Utilizando el modelo de interpretación abstracta, los análisis estáticos y dinámicos parecen tener una naturaleza dual.

Esta dualidad y la ganancia que puede obtenerse de una sinergia entre métodos estáticos y dinámicos conducen a nuevas posibilidades que deben investigarse en el futuro, a través del estudio de métodos híbridos.

En esta sección se presentaron algunos modelos y criterios académicos, así como métodos de análisis dinámicos y estáticos, que pueden ser útiles para diseñar y evaluar algoritmos de desofuscación. También se destacó la importancia de los métodos híbridos. La siguiente sección presenta algunas de las herramientas actualmente disponibles para ayudar a deshacer las transformaciones de ofuscación.

Herramientas de desofuscación

En esta sección, analizamos algunas de las herramientas que puede utilizar para realizar ingeniería inversa de código ofuscado y, en especial, las funciones que ofrecen para facilitar su trabajo. Tenga en cuenta que esta lista no pretende ser exhaustiva de ninguna manera; se basa en la experiencia de algunos de los autores y busca presentar diferentes categorías de herramientas.

IDA

IDA es la herramienta de última generación para la ingeniería inversa de código binario. Lanzar el binario que se desea analizar a IDA es una reflexión común, por lo que probablemente no sea necesario presentar esta herramienta aquí; de lo contrario, los lectores pueden consultar el libro *The IDA Pro Book* de Chris Eagle (No Starch Press, 2011). En relación con el tema específico que nos interesa aquí, tratar con código ofuscado usando IDA es problemático (aunque no imposible) por algunas razones:

- Ese no es el propósito principal de IDA. El código ofuscado es un caso muy particular y manejar cada situación o truco específico sería una tarea interminable; por lo tanto, es mejor no comenzar por ese camino.
- Tenemos muy poco control sobre el desensamblador, un punto que nos impide mucho cuando nos encontramos con esquemas de ofuscación que rompen/interrumpen/destroyer el gráfico del flujo de control. El desensamblador de IDA es muy fácil de confundir y a menudo uno termina con el problema del huevo y la gallina: para recuperar el flujo de control uno necesita limpiar el flujo de datos, pero para limpiar el flujo de datos uno necesita el flujo de control.
- IDA en sí no ofrece ningún tipo de representación intermedia (IR) o al menos semántica de instrucciones, por lo que el análisis avanzado de su salida no es trivial.

En 2008, en el taller ICAR (http://www.hex-rays.com/products/ida/support/ppt/caro_obfuscation.ppt), Ilfak Guilfanov ofreció algunos consejos útiles sobre cómo utilizar características específicas de IDA:

- Fusión de bloques a nivel de gráfico para simplificar el CFG
- Modificación sobre la marcha y basada en eventos del gráfico mediante ganchos como `grcode_changed_graph` (ver `graph.hpp` en el SDK)
- Desarrollar complementos específicos

IDA se puede ampliar mediante scripts (IDC o IDAPython) o complementos (consulte el SDK de IDA). Si tuviera que implementar algún análisis avanzado, ahí es donde podría interactuar.

Para tal fin, se han desarrollado algunos complementos como marcos de desofuscación (por ejemplo, el complemento Optimice de Branko Spasojevic, <http://optimice.googlecode.com>). Para intentar abordar algunos de los problemas mencionados anteriormente, incluida la semántica de instrucciones (basada en la Referencia de instrucciones y códigos de operación x86 (<http://ref.x86asm.net/>)), El complemento ofrece reducción de CFG, optimizaciones de mirilla y eliminación de código muerto.

Metasm

Metasm (<http://metasm.cr0.org>) es un marco de código abierto (publicado bajo la licencia GNU Lesser GPL v2) desarrollado por Yoann Guillot. Se define a sí mismo como una suite de manipulación de ensamblado. El marco, escrito en Ruby, ofrece funciones de ensamblador, desensamblador, compilador, enlazador y depurador para arquitecturas cruzadas. Los procesadores compatibles actualmente son Intel x86/x64, MIPS, PPC, SH4 y ARCompact. También se admiten los formatos de archivo más comunes, como MZ, PE/COFF, ELF, Mach-O, etc.

Devoluciones de llamadas del desensamblador

El comportamiento del desensamblador se puede modificar dinámicamente utilizando un conjunto de devoluciones de llamadas exportadas de la clase Disassembler. Las dos más útiles para la desofuscación son las siguientes:

- `callback_newaddr`: se llama cada vez que se descubre una ruta y está a punto de desensamblarse. En este punto, puede inspeccionar la ruta hacia atrás o hacia adelante para ver si presenta errores; lo más importante es que puede modificar el comportamiento del motor de desensamblado: eliminar una transferencia de control falsa, frustrar una trampa del desensamblador, etc.
- `callback_newinstr`: como sugiere su nombre, su devolución de llamada se llama cada vez que se desmonta una nueva instrucción.

Semántica de instrucciones

Una de las características clave del framework es el backtracking (considérelo como un corte de programa). Esta característica es la base de su motor de desensamblaje. Permite una recuperación muy precisa del flujo de control, a costa del rendimiento. Basada en esta característica, la API del framework también ofrece un método para calcular la semántica de un bloque básico. Sin embargo, Metasm no utiliza un lenguaje intermedio estricto; se basa en una descripción de la semántica de cada instrucción. La terminología asociada en el framework es vinculante. Metasm separa la codificación de la semántica del flujo de control y del flujo de datos. Se utilizan cuatro tipos para describir la semántica de una instrucción:

■ Valor numérico

■ Símbolo: cualquier cosa que no sea un valor numérico, según el tipo de símbolo de Ruby.

■ Expresión: Expresión[operando1, (operador), (operando2)]—Una

El operando puede ser cualquiera de los cuatro tipos.

■ Indirección: Indirección de memoria Indirección [objetivo, tamaño, origen]

El siguiente fragmento le presentará el enlace de instrucciones de Metasm:

```
# codificación: ASCII-8BIT
#!/usr/bin/env ruby
requiere "metasm" incluye
Metasm

# producir código x86 sc =
Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOS)
añadir eax, 0x1234
movimiento [eax], 0x1234
retirado
EOS

dasm = sc.init_desensamblador

# desensamblar el código del controlador
dasm.desmontar(0)

# obtener instrucción decodificada en la dirección 0
# entonces su bloque básico
bb = dasm.di_at(0).bloque

# mostrar código desensamblado
pone "\n[+] código generado:"
pone bb.list

# ejecutar a través de la lista de instrucciones decodificadas del bloque básico bb.list.each{|di| puts
"\n[+] #{di.instruction}" sem
= di.backtrace_binding()}
```

298 Capítulo 5 ■ Ofuscación

```

    pone    "flujo de datos:"
    sem.each{[clave, valor] pone           * #{clave} => #{valor}"}

    # ¿La instrucción modifica el puntero de instrucción?
    si di.opcode.props[:setip]
        pone    "flujo de control:"
        # y luego muestra la semántica del flujo de control
        pone    " * #{dasm.get_xrefs_x(di)}"
    fin
}

```

Para cada DecodedInstruction, llama al método backtrace_binding . Devuelve un hash. Cada par clave/valor representa una asignación de la clave según el valor y expresa resultados con respecto a las entradas. La ejecución de los scripts produce el siguiente resultado:

```

[+] código generado:
0 añadir eax, 1234h
5 mov dword ptr [eax], 1234h
0bh ret; fin de subpunto de entrada_0

[+] añadir eax, 1234h
flujo de datos:
* eax => eax+1234h
* eflag_z => ((eax+1234h)&0xffffffff)==0
* eflag_s => (((eax+1234h)>>1fh)&1)!=0
* eflag_c => ((eax&0xffffffff)+1234h)>0xffffffff
* eflag_o => (((eax>>1fh)&1)==0)&&((((eax>>1fh)&1)!=0) ==
((((eax+1234h)>>1fh)&1)!=0))

[+] movimiento dword ptr [eax], 1234h
flujo de datos:
* dword ptr [eax] => 4660

[+] retirar
flujo de datos:
* esp => esp+4+0
Flujo de control:
* [[Indirección[Expresión[:esp], 4, 0xb]]]

```

La instrucción RET es bastante representativa de la distinción entre flujo de datos y flujo de control. El método get_xrefs_x proporcionado por el objeto desensamblador devuelve una lista (un objeto Ruby Array) de posibles valores para el puntero de instrucción. Para esa instrucción específica, es una indirección cuyo objetivo es el ESP. registro y cuyo tamaño es 4 (para la arquitectura ia32), es decir, dword ptr [ESP]; 0xb es la dirección en el programa donde ocurre la indirección.

Retroceso y segmentación

Hasta ahora, ha visto cómo se describe la semántica de cada instrucción aislada . Ahora considere las instrucciones dentro de un flujo de control y cómo se describe la semántica de una instrucción.

Se puede utilizar el enlace. Para ello, el siguiente ejemplo muestra un patrón típico de cálculo de salto dinámico:

```
# codificación: ASCII-8BIT #!/usr/bin/env
ruby
requiere "metasma"
incluye Metasm

# produce el código x86 del controlador
sc = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOS)
entrada:
    movimiento ecx, 1
    error de shl, 0xA
    agregar edx, 0xBADC0FFE
    movimiento eax, 0x100000 lea
    eax, [ecx+eax]
    agregar ecx, 0xBADC0FFE jmp eax

EOS

# desensamblar el código del controlador
dasm = sc.init_disassembler.dasm.disassemble(0)

# obtener el bloque básico bb =
dasm.block_at(0)
objetivo = dasm.get_xrefs_x(bb.list.last).first.pone "[+] jmp objetivo: #{objetivo}"

# seguimiento inverso
valores = dasm.backtrace(objetivo, bb.lista.última.dirección,
    { :log => bt_log = [], :include_start => verdadero})
```

get_xrefs_x le indica cuál es el objetivo de la instrucción de salto final. Luego, se utiliza el método de retroceso para recorrer el flujo de control, siguiendo las dependencias de las variables, hasta que alcanza las asignaciones de variables o simplemente alcanza su límite de complejidad. Cada paso del retroceso se almacena dentro de la matriz bt_log. A continuación se agregan algunas líneas más para generar un buen resultado en el registro:

```
bt_log.each[entrada] tipo de caso
    = entrada.first
    cuando :inicio
        entrada, expr, addr = entrada.pone "[inicio]"
        retroceso expr #{expr} desde 0x#{addr.to_s('16)}"

    cuando :di
        entrada, a, desde, instr = entrada.pone "[actualizar] instr"
        #{instr}.ln -> actualizar expr desde #{desde} hasta
        #{tonelada}

    cuando :encontrado
        entrada, final = entrada
```

300 Capítulo 5 ■ Ofuscación

```

pone "[encontrado] valor posible: #{final.first}\n"

cuando :arriba
    entrada, a, desde, addr_down, addr_up = entrada
    poner "[arriba]"
    addr 0x#{addr_down.to_s(16)} -> 0x#{addr_up.to_s(16)}"
fin
}

```

Aquí está el resultado del ejemplo:

```

[+] objetivo jmp: eax
[inicio] retroceder expr eax desde 0x1c
[actualización] instr 13h lea eax, [ecx+eax],
    -> actualizar expr de eax a ecx+eax
[actualización] instr 0eh mov eax, 100000h,
    -> actualizar expr de ecx+eax a ecx+100000h
[actualización] instr 5 shl ecx, 0ah,
    -> actualizar expr de ecx+100000h a (ecx<<0ah)+100000h
[actualización] instr 0 mov ecx, 1,
    -> actualizar expr de (ecx<<0ah)+100000h a 100400h
[Encontrado] valor posible: 100400h

```

El motor de retroceso ha sido capaz de recorrer el flujo de instrucciones para calcular el valor final de la expresión retrocedida. Un motor de simplificación permite resolver (o al menos reducir) expresiones tanto a nivel simbólico como numérico.

Del registro de registro es posible incluso extraer una porción, es decir, el subconjunto mínimo del programa original que produce el efecto estudiado (el criterio de corte). En este caso, la porción contendrá todas las instrucciones involucradas en el cálculo del destino JMP :

```

# El objeto DecodedInstruction es el tercer elemento de :id entry slice = bt_log.select{|e|
e.first==:di}.map{|e| e[3]}.reverse puts slice

```

El corte es el siguiente:

```

0 movimientos ecx, 1
5 shl ecx, 0ah
0eh movimiento eax, 100000h
13h lea eax, [ecx+eax]

```

Observe cómo se han eliminado de esta lista los cálculos/asignaciones no significativos (por ejemplo, los que utilizan la constante 0BADC0FFEh) .

Ese ejemplo es un caso ideal: la expresión se puede reducir/resolver estáticamente en un valor numérico. Ahora, imagine que elimina la primera línea de ensamblaje (MOV ECX, 1) (dentro del ámbito de bloques básicos, ECX no está definido) y luego vuelve a realizar el análisis:

```

[+] objetivo jmp: eax
[inicio] retroceso de expr eax desde 0x17 [actualización]
instr 0eh lea eax, [ecx+eax],
    -> actualizar expr de eax a ecx+eax

```

```
[actualización] instr 9 mov eax, 100000h,
-> actualizar expr de ecx+eax a ecx+100000h
[actualización] instr 0 shl ecx, 0ah,
-> actualizar expr de ecx+100000h a (ecx<<0ah)+100000h
```

El valor de retorno es un objeto de tipo Expresión cuyo valor es $(ECX << 0Ah) + 100000h$.

Este ejemplo es bastante trivial. La capacidad del backtracker va mucho más allá de eso. A continuación se modifica el ejemplo anterior para incluir un gráfico de flujo de control más complejo:

```
# produce el código x86 del controlador sc =
Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOS)

entrada:
    movimiento ecx, 1
    prueba edx, edx jnz
    etiqueta inc cl
etiqueta:
    error de shl, 0xA
    agregar edx, 0xBADC0FFE
    movimiento eax, 0x100000
    lea eax, [ecx+eax] agrega ecx,
    0xBADC0FFE
    jmp-eax
EOS

# desensamblar el código del controlador
dasm = sc.init_disassembler dasm.disassemble(0)

# obtener el último bloque básico bblist =
dasm.instructionblocks.sort{|b1, b2| b1.address <=> b2.address}
bblist.each{|bb| pone "-n", bb.list}
bb = bblist.last
```

Básicamente, se ha insertado una instrucción (TEST EDX, EDX) que controla un salto condicional; en un caso se incrementa ECX , en el otro no. La salida actualizada es la siguiente:

```
[+] objetivo jmp: eax
[inicio] retroceso de expr eax desde 0x21 [actualización] instr 18h lea
eax, [ecx+eax],
-> actualizar expr de eax a ecx+eax [actualizar] instr 13h mov eax,
100000h,
-> actualizar expr de ecx+eax a ecx+100000h
[actualización] instr 0ah shl ecx, 0ah,
-> actualizar expr de ecx+100000h a (eax<<0ah)+100000h
[arriba] dirección 0xa -> 0x7
[dirección 0xa -> 0x7]
[arriba] [actualizar] instr 0 mov ecx, 1,
-> actualizar expr de (eax<<0ah)+100000h a 100400h
[Encontrado] valor posible: 100400h
```

302 Capítulo 5 ■ Ofuscación

```
[actualización] instr 9 inc ecx,
-> actualizar expr de (ecx<<0ah)+100000h a ((ecx+1)<<0ah)+100000h
dirección 0x9 > 0x7
[arriba] [actualizar] instr 0 mov ecx, 1,
-> actualizar expr de ((ecx+1)<<0ah)+100000h a 100800h
[Encontrado] valor posible: 100800h
```

El backtracker devuelve una matriz de dos valores posibles: 100400h o 100800h.

Nótese cómo se ha seguido el flujo de control sobre el CFG. (Se han seguido ambas ramas del condicional). Una etiqueta [up] indica el cruce de un bloque básico.

El retroceso es realmente el núcleo del desensamblador y produce un desensamblado más preciso. Obviamente, esta función conlleva graves penalizaciones de rendimiento (recuerde la compensación entre computabilidad y precisión).

Vinculación de código

Sabes cómo obtener la semántica de una instrucción aislada y cómo retroceder un valor y calcular una porción para ese valor en particular. ¿Qué sucedería si pudieras generalizar este proceso y calcular la semántica de un bloque básico?

Esta es otra característica muy potente de Metasm: el método code_binding , proporcionado por el objeto desensamblador. Depende totalmente de la función de retroceso. Aquí está su uso en el último bloque básico del ejemplo anterior:

```
# Calcular la semántica básica del bloque bbsem =
dasm.code_binding(bb.list.first.address, bb.list.last.address) puts "n[+] semántica básica del bloque"

bbsem.each{|clave, valor| pone "#{clave} => #{valor}"}
```

Su salida es la siguiente:

```
[+] semántica básica de bloques
* eax => ((ecx<<0ah)+100000h)
* ecx => ((ecx<<0ah)+badc0ffe)
* edx => (edx+badc0ffe)
```

Misma

Misma (<http://code.google.com/p/smiasm>) es un marco de ingeniería inversa desarrollado por Fabrice Desclaux que ofrece manipulación, ensamblaje y desensamblaje de PE/ELF (actualmente admite Ia32, ARM, PPC y bytecode Java). Al igual que Metasm, Miasm es de código abierto y se publica bajo la licencia GNU Lesser GPL v2, por lo que puedes explorar su motor para personalizarlo según tus necesidades. Los ejemplos que se ofrecen en esta sección se basan en la última revisión de MIASM disponible al momento de escribir este artículo (changeset:270:6ee8e9a58648).

El marco se basa en un lenguaje intermedio. Esto significa que la mayoría de las instrucciones comunes tienen su semántica codificada como una lista de expresiones. “Lista” debe entenderse en su significado en Python (es decir, un conjunto ordenado de objetos).

La gramática del IR de Miasm hace uso de nueve tipos de expresiones básicas, siendo los más importantes los siguientes:

- ExprInt—Valor numérico
- ExprId—Identificador/símbolo, cualquier cosa que no sea un valor numérico; por ejemplo, los registros se definen como ExprId
- ExprAff—Afectación $a = b$
- ExprCond—Operador ternario/condicional $a ? b : c$
- ExprMem: indirección de memoria
- ExprOp—Operación $op(a,b,...)$

También ofrece compatibilidad total con sectores (piense en él como un objeto para representar campos de bits) y composición de sectores. El IR permite cálculos simbólicos y está equipado con un motor de simplificación de expresiones.

Para cada procesador compatible, un archivo con el sufijo “sem” describe la semántica de las instrucciones más comunes. Consulte, por ejemplo, la semántica ADD tal como se define en “miasm/arch/ia32 sem.py”:

```
def agregar(info, a, b):
    mi= []
    c = ExprOp('+', a, b)
    e+=actualizar_indicador_arith(c)
    e+=actualizar_indicador_af(c)
    e+=actualizar_indicador_add(a, b, c)
    e.append(ExprAff(a, c))
    volver e
```

Esta función construye la semántica de las instrucciones en función de sus dos operandos (a y b). Se puede escribir fácilmente un fragmento de código para demostrar estas funciones:

```
#!/usr/bin/env python

desde miasm.arch.ia32_arch importar *
desde miasm.tools.emul_helper importar *

# instrucción de ensamblaje asm en la dirección indicada

def instr_sem(instr, dirección):
    imprimir "\n[+] Instrucción %s @ 0x%x" % (instr, dirección) binario = x86_mn.asm(instr) di =
    x86_mn.dis(binary[0])

    semántica = get_instr_expr(di, dirección) para expr en semántica:

        imprimir "%s" % expr
```

```
instr_sem("agregar eax, 0x1234", 0) instr_sem("mov
[eax], 0x1234", 0) instr_sem("ret", 0) instr_sem("je
0x1000", 0)
```

Aquí está el resultado:

```
[+] instrucción add eax, 0x1234 @ 0x0
zf = ((eax + 0x1234) == 0x0)
nf = ((0x1 == (((eax + 0x1234) >> 0x1F)) y 0x1)
pf = (paridad (eax + 0x1234)) af = (((eax + 0x1234) y 0x10) == 0x10)
cf = ((0x1 == (((eax ^ 0x1234) ^ (eax + 0x1234)) >> 0x1F)))
        (0x1 == (((((eax + 0x1234) y (! (eax ^ 0x1234))) >> 0x1F)) ^ 0x1234))) >> 0x1F))
de = (0x1 == (((eax ^ 0x1234) y (! (eax ^ 0x1234))) >> 0x1F))
eax = (eax + 0x1234)

[+] instrucción mov [eax], 0x1234 @ 0x0
@32[eax] = 0x1234

[+] instrucción ret @ 0x0
esp = (esp + (0x4 + 0x0)) eip = @32[esp]

[+] instrucción je 0x1000 @ 0x0 eip = (zf == 0x1)?
(0x1000,0)
```

La semántica de la instrucción ADD parece ser la más compleja debido a la actualización de los indicadores . Aquí se muestra la semántica de la instrucción MOV con un tipado explícito del objeto:

```
[+] instrucción mov [eax], 0x1234 @ 0x0
ExprAff( ExprMem(@32[ExprId(eax)]) = ExprInt(0x1234) )
```

Esta es una característica muy apreciable y poderosa. Basada en el IR, hay una característica de compilación justo a tiempo (JIT) mediante la cual el código primero se desensambló, se tradujo al IR y luego se volvió a generar como código nativo para su ejecución. La documentación y los ejemplos proporcionan casos de uso de Miasm para el análisis de empaquetadores/VM, así como para la instrumentación binaria.

Desnudarse

VxStripper es una herramienta de reescritura binaria, desarrollada por Sébastien Josse. Diseñada para el análisis de programas binarios protegidos y potencialmente hostiles, extrae dinámicamente una representación intermedia de un ejecutable binario y toda la información necesaria para aplicar ciertas simplificaciones, haciendo que el funcionamiento interno del binario sea más fácil de entender para el analista.

Una de las principales motivaciones detrás del diseño y las opciones de implementación de esta herramienta es sortear las limitaciones actuales de las soluciones de análisis de malware y programas binarios existentes. (Muchas herramientas vienen con su propia representación intermedia, no exportable, a veces propietaria, lo que dificulta su integración. Además, muchas de ellas no son adecuadas para el análisis de código hostil o protegido). El objetivo es obtener la mayor cantidad de información posible de

Un programa binario que utiliza todas las técnicas y herramientas disponibles para proteger esta información. La idea es instrumentar una unidad de procesamiento de computadora virtual y un sistema operativo invitado de manera no intrusiva para obtener dinámicamente la información necesaria para reconstruir el programa y simplificar su representación. Esta herramienta se basa en el motor de traducción binaria dinámica de QEMU y en la cadena de compilación LLVM.

LLVM (Low Level Virtual Machine) es una cadena de compilación que viene con un conjunto consecuente de optimizaciones que se pueden aplicar durante toda la vida útil de un programa. LLVM utiliza un conjunto de instrucciones de tipo RISC fuertemente tipado y una representación de asignación única estática (SSA) (al utilizar esta representación, cada variable temporal se asigna solo una vez). LLVM incluye muchos back-ends binarios (x86, x86-64, SPARC, PowerPC, ARM, MIPS, CellSPU, XCore, MSP430, MicroBlaze, PTX) y algunos back-ends de código fuente (C, C++). Los lectores pueden consultar el artículo de Lattner³¹ para obtener más detalles sobre LLVM.

El traductor binario dinámico (DBT) QEMU (Quick EMULATOR) se utiliza para traducir dinámicamente el código binario de la arquitectura de la CPU invitada a la arquitectura de la CPU host, mediante el uso de un IR llamado TCG (Tiny Code Generator).

Este lenguaje consta de instrucciones simples similares a RISC llamadas microoperaciones. La traducción binaria consta de dos etapas. El código binario invitado se traduce primero en secuencias de instrucciones TCG, llamadas bloques de traducción (interfaz DBT). Luego, los bloques de traducción se convierten en código ejecutable por la CPU anfitriona (interfaz DBT). La DBT de QEMU viene con muchas interfaces binarias (x86, x86-64, ARM, ETRAX CRIS, MIPS, Micro Blaze, PowerPC, SH4 y SPARC). Los lectores pueden consultar el artículo de Bellard⁴ para obtener más detalles sobre QEMU.

VxStripper hereda de QEMU los numerosos front-ends binarios y de LLVM los numerosos back-ends, lo que proporciona a un coste razonable un marco de reescritura binario completo. Las funciones de reescritura se implementan a medida que pasa LLVM.

Su diseño actual se basa en el trabajo ya realizado para convertir TCG IR a LLVM IR (LLVM-QEMU, descrito por Scheller³⁶, y S2E, descrito por Chipounov⁸), así como en los algoritmos de diseño presentados por Josse.^{27, 28}

Uno de los objetivos de esta herramienta es la colaboración con las numerosas herramientas de análisis de software basadas en la cadena de compilación LLVM, a través de una representación “exportada” del programa malicioso. Esta herramienta de análisis binario está especialmente diseñada para resolver el problema del análisis de programas hostiles. El objetivo es automatizar las tareas, a menudo fastidiosas y repetitivas, que lleva a cabo un analista.

Esta cadena de compilación se basa en una arquitectura modular y evolutiva, lo que permite aplicar las mismas transformaciones a una amplia variedad de arquitecturas de software y hardware. Se basa en una cadena de compilación moderna, que proporciona una representación intermedia y funcionalidades eficientes.

Vellvm (LLVM verificado), descrito por Zhao,⁴³ proporciona herramientas formales para razonar sobre las transformaciones que operan en la representación intermedia de LLVM. Vellvm se puede utilizar para extraer implementaciones formalmente verificadas de los pasos de desofuscación implementados en VxStripper.

Extensión DBT de QEMU

Ha visto que el motor DBT QEMU realiza la traducción dinámica del código binario desde la arquitectura del procesador invitado a la arquitectura del procesador host utilizando la representación intermedia TCG.

Usando un ejemplo simple, la siguiente instrucción demuestra cómo se ve este lenguaje:

```
0x0040104c: enviar 0xa
```

La instrucción anterior se traduce de la siguiente manera en la representación TCG de QEMU:

```
(1) movi_i32 tmp0,$0xa mov_i32
(2) tmp2,esp movi_i32
(3) tmp13,$0xffffffffc
(4) agregar_i32 tmp2,tmp2,tmp13
(5) qemu_st32 tmp0,tmp2,$0x1
(6) mov_i32 esp,tmp2
(7) movi_i32 tmp4,$0x40104e
(8) st_i32 tmp4,env,$0x30
(9) salida_tb $0x0
```

Este bloque de instrucciones TCG emula la ejecución de la instrucción push en la CPU del software. Las operaciones realizadas son las siguientes: El entero 0xa se almacena en la variable tmp0 (línea 1). Esta variable se almacena luego en la pila (líneas 2 a 6). La dirección de la instrucción que sigue a la instrucción actual se almacena en tmp4 (línea 7) y luego se almacena en el registro cc_op de la VPU de QEMU . La última instrucción (línea 9) indica el final del bloque TCG.

La herramienta modifica el mecanismo DBT de tal forma que la función de instrumentación de la CPU virtual se invoque sistemáticamente antes de la ejecución de un bloque de traducción. Para lograr esto, se agrega una microoperación extra (op_callback) que toma como operando la dirección de la función de instrumentación (vpu_callback).

El código TCG resultante es el siguiente:

```
(1) devolución de llamada de operación @vpu_callback
(2) movi_i32 tmp0,$0xa mov_i32
(3) tmp2,esp movi_i32
(4) tmp13,$0xffffffffc
(5) agregar_i32 tmp2,tmp2,tmp13
(6) qemu_st32 tmp0,tmp2,$0x1
(7) mov_i32 esp,tmp2
(8) movi_i32 tmp4,$0x40104e
(9) st_i32 tmp4,env,$0x30
(10) salida_tb $0x0
```

Este mecanismo le permite ejecutar su código de instrumentación en cada ciclo de ejecución de la CPU virtual. Con acceso a los registros de la VPU y a la memoria de la PC virtual, puede adquirir un contexto de proceso y extraer información sobre sus interacciones con el sistema operativo invitado.

Instrumentando también la carga y almacenando las instrucciones TCG, se puede extraer información sobre las interacciones del proceso de destino con la memoria del sistema invitado. Gracias a esta información, se puede recuperar la información de reubicación del proceso.

Ahora que ha visto cómo modificar la CPU virtual QEMU para permitir la invocación sistemática de su función de instrumentación, examinemos la traducción de la representación intermedia de TCG a la representación de LLVM. El resultado de traducir el bloque TCG anterior es el siguiente:

```
(1) %esp_v.i = carga i32* @esp_ptr
(2) %tmp2_v.i = suma i32 %esp_v.i, -4
(3) %4 = intoptr i32 %tmp2_v.i a i32*
(4) almacenar i32 10, i32* %4
(5) almacenar i32 %tmp2_v.i, i32* @esp_ptr
(6) almacenar i32 4198478, i32* %next.i
(7) almacenar i32 0, i32* %ret.i
```

El entero 0xa se almacena en la dirección indicada por la variable %4, lo que equivale a almacenarlo en la pila (líneas 1 a 4). La dirección de la instrucción que sigue a la instrucción actual se almacena en la variable %next.i (línea 6). La última instrucción (línea 7) finaliza el bloque LLVM.

Después del proceso de normalización, este bloque LLVM se compila de la siguiente manera:
Código ensamblador ing:

```
401269f mov dword ptr [esp-14h], 0ah
```

Ahora que tiene una descripción general de las principales modificaciones aplicadas al emulador QEMU, como se muestra esquemáticamente en la Figura 5-2, la siguiente sección describe la arquitectura general de la herramienta.

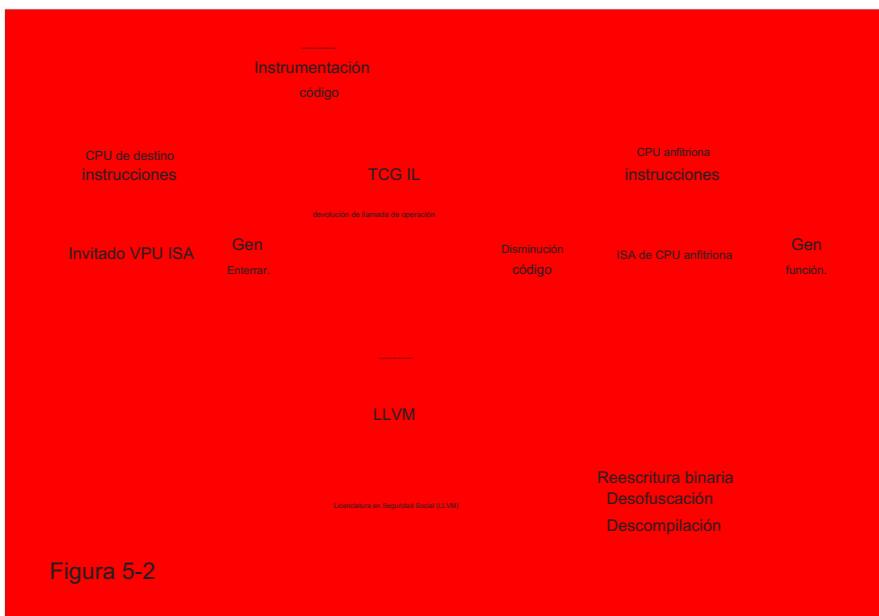


Figura 5-2

Arquitectura de VxStripper

VxStripper implementa un motor DBT extendido y varias funciones de análisis especializadas (ver Figura 5-3) para observar el programa de destino y su entorno de ejecución.



Figura 5-3

Un administrador de módulos maneja la activación y la colaboración entre estos análisis. funciones, implementadas como complementos.

Estas funciones de análisis extraen información semántica del programa de destino . Esta información puede ser el rastro de sus interacciones con las API del sistema operativo invitado, o la forma en que maneja los objetos y las estructuras del sistema operativo invitado o del núcleo, o simplemente su rastro de código de máquina.

La extracción de esta información se basa en una descripción del sistema operativo invitado, que puede ser proporcionada, por ejemplo, por un servidor de símbolos, como es el caso de la familia de sistemas operativos Windows.

Entre los módulos ya implementados se encuentran los siguientes:

- Un módulo de enganche de API ■

Un módulo de análisis forense ■ Un

módulo de desempaquetado

- Un módulo de normalización

Conexión de API

El módulo de enlace de API nativo y de Windows de VxStripper se basa en el análisis forense de la memoria del sistema operativo invitado, sin ninguna interacción con el sistema operativo invitado.

El ejecutivo de Windows mantiene un conjunto de estructuras que contienen información sobre los módulos cargados para un proceso determinado dentro de su espacio de memoria. Estas estructuras de datos se pueden recuperar utilizando el bloque de entorno de proceso (PEB), al que se puede acceder como un desplazamiento del registro de segmento FS. Los módulos de conexión de API nativos y de Windows de VxStripper utilizan esta información para localizar e instrumentar la API de Windows.

Análisis forense/root-kit

El módulo forense de VxStripper incluye funciones adicionales para supervisar y comprobar la integridad de muchas ubicaciones dentro de la plataforma invitada donde se puede instalar un ganchito. Recorre las estructuras ejecutivas del sistema operativo para identificar objetivos potenciales de un ataque de rootkit y supervisar los componentes de hardware que podrían verse dañados por un rootkit. Esta información es crucial para que el analista comprenda los ataques virales de bajo nivel.

Para los fines de este capítulo, puede considerar que estas características son similares a las que se esperan de un depurador de kernel. Puede adjuntar un proceso, ver su estado de CPU y código desensamblado, y rastrear la interacción del programa de destino con la API del sistema operativo. Esta inspección se realiza en un entorno seguro y controlado, sin ninguna interacción intrusiva con el sistema operativo invitado.

Las siguientes dos secciones analizan en profundidad el funcionamiento de los dos módulos de análisis más importantes de Vxstripper: el módulo de descompresión y el módulo de normalización.

Desembalaje del módulo

El módulo de desempaquetado localiza el punto de entrada original (OEP) del ejecutable de destino , obtiene información relativa a sus interacciones con la API del sistema operativo y extrae la información de reubicación.

La idea subyacente es una simple comprobación de la integridad del código ejecutable del programa de destino: para cada bloque de traducción del programa, se realiza una comparación entre su valor en la memoria virtual y su valor en el sistema de archivos del host. Mientras los valores sean idénticos, no se hace nada. Tan pronto como se identifica una diferencia , el bloque de traducción actual se escribe en el archivo sin procesar en lugar del bloque de traducción anterior. La primera instrucción del bloque de traducción recién generado se identifica como el OEP del programa protegido. Al final del análisis, las secciones de datos se escriben en el archivo sin procesar en lugar de las secciones de datos originales.

Se aplica el mismo algoritmo de monitoreo para cada bloque de traducción. El cargador de protección del ejecutable empaquetado puede tener varias capas de descifrado. Una vez que se ha alcanzado el último bloque de traducción descifrado, lo único que queda es...

310 Capítulo 5 ■ Ofuscación

Lo que hay que hacer es reparar el ejecutable de destino. Para recuperar la estructura PE (Portable Executable) de un ejecutable desprotegido, se deben llevar a cabo varias tareas : establecer el punto de entrada original, reconstruir las tablas de importaciones y reubicaciones y comprobar la coherencia del encabezado PE.

El método utilizado por el motor de descompresión para reconstruir la tabla de direcciones de importación (IAT) y las reubicaciones se basa en Win32 y en el enganche de API nativo. Durante el proceso de desempaquetado, se rastrean todas las llamadas a la API. En el momento de la carga, se inicializa una tabla ordenada de llamadas a la API mediante el recorrido de las estructuras ejecutivas de NT.

A continuación, una vez que se ha reanudado la ejecución del proceso, se realiza un seguimiento de cada llamada a la API. Esta tabla se actualiza periódicamente durante la ejecución del proceso de destino y se utiliza para resolver de forma dinámica los nombres de las funciones de la API. Por último, una vez que se completa un volcado del espacio de memoria del proceso de destino, esta tabla se utiliza para corregir el IAT en el ejecutable PE.

Gracias a la instrumentación de carga y almacenamiento de instrucciones TCG, es posible extraer dinámicamente la información de reubicación del programa, que también puede agregarse a una nueva sección del ejecutable.

Por ejemplo, aquí está la información (útil) extraída durante el desempaquetado.

Etapa de un programa que muestra un cuadro de diálogo (función MessageBoxA):

```
[[INFO] eip=0x00401000
[RELOC] valor=0x00403000 va=0x00401003
[RELOC] valor=0x0040300f va=0x00401008
[RELOC] valor=0x00402008 va=0x00401010
[APICALL] api_pc=0x77d8050b api_oep=0x77d8050b
    nombre_dll=C:\WINDOWS\system32\user32.dll
    nombre_función=MessageBoxA
    valor=0x00402008 va=0x00401010
```

La información de reubicación consta de pares (va, valor) que proporcionan la dirección virtual y el valor a reubicar, respectivamente. Tenga en cuenta que para este empaquetador, el prólogo de la función MessageBoxA no es emulado por la protección. De lo contrario, la dirección externa que efectivamente se llama (api_pc) es diferente del punto de entrada de la función API (api_oep).

Normalización

En la mayoría de los casos, después de la etapa de desempaquetado, se puede obtener (automáticamente) un binario sin su cargador de protección y sin ningún código reescribible. Lamentablemente, algunos mecanismos de ofuscación (aplanamiento del flujo de control, transformaciones de ofuscación basadas en VM , etc.) deben manejarse ahora para comprender completamente el funcionamiento interno de un malware.

En VxStripper se ha implementado un primer intento de proporcionar una solución a estos problemas mediante el uso de la representación intermedia LLVM. En lugar de intentar trabajar en el binario después de que se haya volcado su imagen de memoria, la idea es trabajar en su representación intermedia y aumentar la cantidad de información (que se ha recopilado dinámicamente) incorporándola en el módulo LLVM. Este tipo de representación es más adecuada para un análisis posterior.

El módulo de normalización utiliza la salida de análisis anteriores para generar la representación LLVM de los bloques de traducción, a los que se aplican varias transformaciones de optimización. Examinando esto con más detalle, durante la ejecución del programa de destino, el back end LLVM de QEMU TCG genera la representación LLVM de los bloques traducidos. Este código LLVM está vinculado con un módulo LLVM de inicialización (ver Figura 5-4).



Este módulo de inicialización implementa devoluciones de carga y almacenamiento, declara prototipos de API del sistema y establece una unidad de procesador virtual y su pila.

El módulo de normalización utiliza la información recopilada dinámicamente durante la ejecución del programa de destino para resolver importaciones, procesar reubicaciones y recuperar secciones de datos. La información de la tabla de importación se utiliza para crear instrucciones de llamada a la API de LLVM. El mapa de memoria de carga/almacenamiento se utiliza para aplicar reubicaciones e injectar datos desde el programa de destino al módulo LLVM.

Cuando se reconstruye el módulo LLVM, se aplican algunos pasos de optimización adicionales a su representación. A continuación, se puede compilar el LLVM en la arquitectura elegida, utilizando uno de los back-ends de LLVM disponibles. También se puede traducir a código C o C++.

Los primeros resultados muestran que la optimización estándar utilizada junto con la evaluación parcial inducida por la traducción dinámica del código de destino a su representación LLVM son suficientes para reducir y simplificar drásticamente el código bajo análisis.

312 Capítulo 5 ■ Ofuscación

Reflexiones finales

En esta sección se han analizado sólo cuatro herramientas (y con un sesgo hacia el análisis estático). Junto con los marcos Metasm y Miasm, podríamos haber citado el marco Radare (<http://www.radare.org/y/>), Por ejemplo, los esfuerzos de Rolf Rolles por ampliar IDA con su intérprete idaocaml (<https://code.google.com/p/idaocaml/>) También merecen atención. Hay muchos otros que no mencionamos o que solo mencionamos brevemente aquí, y lo alentamos a que los pruebe usted mismo.

Para poner estas herramientas en perspectiva, aunque IDA es un buen desensamblador, no puede ayudar mucho cuando se trata de lidiar con código ofuscado. Los frameworks Metasm y Miasm van un paso más allá, ofreciendo más control, un IR con el que jugar, etc. Herramientas como VxStripper van aún más allá. Probablemente puedas sentirlo: hay una carrera armamentista en marcha. Se invierte una enorme cantidad de esfuerzo en el desarrollo de ofuscadores, por lo que nuestras herramientas también tienen que evolucionar.

Como ingeniero inverso, desarrollar herramientas es una inversión que se realiza para cumplir con los objetivos y se espera algún tipo de retorno de la inversión. La mayoría de las herramientas más avanzadas pueden tardar semanas o incluso meses en desarrollarse y requieren mucho conocimiento.

Desofuscación práctica

Ahora verá cómo se pueden utilizar algunas de las herramientas presentadas anteriormente para la desofuscación práctica. Nuevamente, no se pretende ser exhaustivo en las siguientes secciones. En cambio, el objetivo es ilustrar algunos casos de uso comunes de las técnicas de desofuscación.

Desofuscación basada en patrones

Esta puede ser la desofuscación más simple y económica, que opera a nivel sintáctico y coincide con patrones conocidos. No olvide que los primeros patrones de ofuscación eran principalmente código creado manualmente y protegido (como algunos códigos de empaquetadores) y exhibían solo un conjunto limitado de patrones; por lo tanto, enumerarlos todos era "aceptable".

Esta técnica de desofuscación se reduce a un algoritmo de búsqueda y reemplazo a nivel binario (código de operación) (que eventualmente utiliza búsqueda con comodines). El principal inconveniente es que te deja con un binario plagado de instrucciones NOP.

Para ilustrar esto, observe el siguiente script antiguo de OllyDbg. Los empaquetadores son un ejemplo clásico de software que utiliza técnicas de ofuscación (en ese caso para proteger sus stubs). Durante años, OllyDbg ha sido (y probablemente aún lo sea) la herramienta favorita para desempaquetar, y se publicaron muchos scripts para ayudar en esa tarea. Este script antiguo (aleatorio) (2004, por loveboom, <http://tuts4you.com/download.php?view.601>) apunta a las versiones 2.0x de ASProtect (un empaquetador de uso común en ese momento).

Aprovecha los comandos REPL de OllyScript para buscar y reemplazar un conjunto de patrones. La definición de REPL es la siguiente:

```
repl addr, buscar, repl, longitud  
Reemplace find con repl iniciando att addr para len bytes.
```

Todos los patrones se basan en la misma técnica: se utiliza un salto incondicional dentro de la instrucción sucesora para confundir a los desensambladores. La diversidad se mejora ligeramente utilizando prefijos de instrucción (como REP o REPNE). La instrucción repl se utiliza para reemplazar estos patrones con NOP:

```
respuesta eip,#2EEB01??#,#90909090#,1000  
respuesta eip,#65EB01??#,#90909090#,1000  
responder eip,#F2EB01??#,#90909090#,1000  
respuesta eip,#F3EB01??#,#90909090#,1000  
respuesta eip,#EB01????#,#909090#,1000  
respuesta eip,#26EB02????#,#9090909090#,1000  
respuesta eip,#3EEB02????#,#9090909090#,1000
```

Aquí sólo operamos a nivel sintáctico. Considerando un objetivo con un conjunto limitado de patrones, esta técnica de desofuscación es trivial, pero eficaz:

- El coste de aplicación es limitado, si no insignificante.
- El coste de desarrollo también es casi nulo.

Por supuesto, al igual que con las firmas de virus y los motores antivirus, el polimorfismo y la diversidad lo hacen inútil. Se podría haber desarrollado un script equivalente utilizando las capacidades de scripting de IDA. Ese es el tipo de script que normalmente se puede crear cuando se analiza malware y/o empaquetadores trivialmente ofuscados, cuando la velocidad del análisis es prioritaria.

Desofuscación basada en análisis de programas

Ahora considere el siguiente ejemplo de código ofuscado (este es solo un extracto muy pequeño; el código ofuscado continúa así durante miles de instrucciones):

```
.texto:00405900 loc_405900:  
.texto:00405900 agregar edx, 67E37DA7h  
.texto:00405906           empujar      esi  
.texto:00405907           movimiento esi, 0D0B763Ah  
.texto:0040590C           empujar      fácil  
.texto:0040590D           movimiento eax, 15983FC8h  
.texto:00405912           neg eax  
.texto:00405914           incluye eax  
.texto:00405915           incluye eax  
.texto:00405916           jmp loc_4082AD  
.texto:004082AD loc_4082AD:  
.texto:004082AD no es fácil  
.texto:004082AF           y eax, 1D48516Ch  
.texto:004082B4           bateria auxiliar, 0ACE1B37Ah
```

314 Capítulo 5 ■ Ofuscación

```
.texto:004082B9 xor esi, eax
.texto:004082BB pop eax
.texto:004082BC xor edx, esi
.texto:004082BE pop esi
.texto:004082BF y ecx, edx
.texto:004082C1 jmp loc_407C54
```

Probablemente reconozcas, de las primeras secciones de este capítulo, algunas de las técnicas de ofuscación que se utilizan aquí, especialmente el despliegue constante. Ten en cuenta también que se insertan saltos incondicionales para dividir el código en bloques básicos que luego se distribuyen (reordenan aleatoriamente) en el binario. No hay ningún patrón obvio, al menos ninguno que puedas hacer coincidir de manera posible y efectiva a nivel sintáctico mediante firmas.

Es necesario avanzar al nivel semántico. Basándose en el resultado de un desensamblado, se podría empezar a trabajar en el flujo de control, fusionando los bloques básicos 405900h y 4082ADh. A continuación, se podría trabajar en el flujo de datos, teniendo en cuenta la Dos instrucciones:

```
.texto:0040590D movimiento eax, 15983FC8h
.texto:00405912 neg eax
```

Según la semántica de estas instrucciones, sabes que al registro EAX primero se le asigna un valor constante y luego se le aplica el valor negativo. Puedes calcular previamente la instrucción NEG y reescribirla de una forma más sencilla, asignando el valor negativo a EAX :

```
.texto:0040590D movimiento eax, EA67C038h
```

Reescribir programas de una forma más sencilla, precalcular valores que no dependen de la entrada del programa, eliminar código inútil... eso es la optimización de programas , y los compiladores lo han hecho casi desde su creación. Puede adaptar y reutilizar estas técnicas para sus propios fines, y hay una abundante bibliografía disponible sobre este tema. Algunas técnicas clásicas de optimización de compiladores incluyen las siguientes:

- Optimización de la mirilla
- Plegado/propagación constante
- Eliminación de tiendas muertas
- Operación de plegado
- Eliminación de código muerto
- Etc.

Este enfoque es exactamente el que propone el complemento de desofuscación Optimice para IDA. También se han presentado algunos trabajos anteriores, de Gazet y Guillot22 y de Josse29 , como complemento Metasm y complemento VxStripper, respectivamente. La idea es normalizar el código para obtener una forma reducida/optimizada/canónica, que sea más sencilla de analizar y más cercana al código original, desprotegido.

Estos intentos de proporcionar marcos/utilidades de desofuscación aún distan de ser perfectos. Además, no hay muchas herramientas disponibles para que el inversor promedio ataque programas ofuscados (por supuesto, existen algunas herramientas avanzadas privadas aquí y allá). El futuro de la desofuscación probablemente tomará la forma de herramientas elaboradas y plataformas de análisis basadas en IR formal. Rolf Rolles presentó algunos de sus resultados con su propio marco escrito en OCaml³⁴. Otros marcos populares que podrían usarse incluyen SecondWrite basado en LLVM (Smithson et al., 2007³⁹), S2e/revgen (Chipounov, 2001⁷) o BitBlaze (Song et al., 2008⁴⁰ y sus trabajos posteriores). Los esfuerzos en desofuscación tendrán que igualar los que se ponen en ofuscación.

Análisis complejo

En esta sección se analizan dos de las técnicas de ofuscación más impactantes: la virtualización y el aplanamiento del código. Para estas técnicas, es evidente que es necesario trabajar a nivel semántico.

Implementaciones de máquinas virtuales simples

Existen principalmente dos formas de implementación de VM. La forma más sencilla es desarrollar un emulador de procesador simple. Algorítmicamente hablando, incluiría los siguientes pasos:

1. Bucle:
 - a. Obtener : lee el flujo de código de bytes en el puntero de instrucción.
 - b. Decodificar: decodifica el código de operación de la instrucción y sus operandos.
 - c. Ejecutar: llama al controlador de código de operación apropiado.
2. Actualice el puntero de instrucción o salga del bucle.

En esta configuración, cada controlador de instrucciones es responsable de actualizar el contexto de la máquina virtual. El contexto representa la arquitectura emulada subyacente . Probablemente consta de un conjunto de registros y, eventualmente, de un área de memoria. Cada controlador implementa una instrucción distintiva del procesador emulado (un controlador para ADD, uno para SUB, etc.).

Esta forma se utiliza a menudo en las implementaciones más sencillas. Los controladores son totalmente independientes entre sí y el puntero de instrucción aumenta en función del tamaño de la instrucción (excepto en el caso de instrucciones que la modifican directamente).

Desde el punto de vista de un atacante, este tipo de implementación es fácilmente reconocible . A continuación, se detallan los pasos que generalmente se requieren para analizar una máquina virtual:

1. Comprender cómo se decodifica una instrucción a partir del código de bytes sin formato: qué parte codifica la operación (número de controlador), qué parte codifica el o los operandos, etc.
2. Deducir la arquitectura de la máquina virtual a partir de los operandos de las instrucciones: número de registros, diseño de memoria, interfaces de E/S , etc.

3. Realice un análisis del controlador. Una vez que se conoce la decodificación de los operandos, puede observar la forma en que cada controlador manipula los distintos operandos que posiblemente tome como argumentos. Este paso es la esencia del análisis de VM: cada controlador está asociado con su propia semántica.

Con todos estos conocimientos, finalmente podrá crear una herramienta similar a un desensamblador que le permita desensamblar el código de bytes de la máquina virtual.

En 2006, Maximus publicó dos excelentes artículos sobre la reversión de máquinas virtuales: "Reversing a simple virtual machine"³² y "Virtual machines re-building"³³. Uno de los objetivos que utilizó (HyperUnpackme2) también fue tratado en profundidad por Rolf Rolles el mismo año³⁵.

Aunque estas son contribuciones útiles de reversores talentosos, el análisis de VM sigue siendo un trabajo algo manual y repetitivo: hay que desarrollar un nuevo desensamblador para cada nueva instancia de VM. Además, los autores de protecciones también han reaccionado, reforzando sus implementaciones de VM.

Implementaciones avanzadas de máquinas virtuales

Las implementaciones de VM más avanzadas derivan del tipo simple, pero agregan características importantes para fortalecer las implementaciones y hacerlas más resistentes al análisis:

- Desenrollado de bucles: esta técnica clásica de optimización del compilador favorece el aspecto temporal (velocidad) de la relación espacio-tiempo de un programa. Reemplaza la estructura de bucles por las invocaciones secuenciales del cuerpo del bucle (por lo tanto, desenrollado). Aplicado a una máquina virtual, cada controlador se hace responsable de obtener y decodificar sus propios operandos y luego actualiza el contexto en consecuencia.
- Aplanamiento de código: el bucle de ejecución principal de la máquina virtual está aplanado. Esto significa que cada controlador es responsable de actualizar el puntero de instrucción (puntero en el código de bytes). En realidad, el aplanamiento de código y la ofuscación basada en la máquina virtual son básicamente lo mismo. El aplanamiento de código solo virtualiza/reorienta el flujo de control del código protegido, mientras que la ofuscación basada en la máquina virtual virtualiza/reorienta tanto el flujo de control como el flujo de datos. Puede utilizar casi los mismos algoritmos para seguir el contexto de un despachador de aplanamiento de código y un contexto de máquina virtual.
- Codificación/cifrado de bytecode: cada invocación de la VM depende de una clave de cifrado que se pasa a la VM como parte de la inicialización de su contexto . Cada controlador actualiza esa clave, lo que da como resultado una clave de activación. Los controladores dependen de la clave de activación para decodificar sus operandos a partir del bytecode codificado. Un atacante no puede comenzar a analizar la VM en un punto elegido, ya que el valor de la clave en este punto sería desconocido.
- Ofuscación de código: el código nativo de la máquina virtual se ofusca mediante técnicas como la descrita al principio de este capítulo. El simple hecho de observar el código de un controlador no proporciona ninguna pista sobre su semántica.

En resumen, las implementaciones reforzadas de máquinas virtuales son mucho más difíciles de analizar de forma estática. Para cada estado de la máquina virtual, un atacante debe conocer al menos los siguientes valores obligatorios:

- Puntero de código de bytes
- Puntero de instrucción, el valor del próximo controlador que se ejecutará
- Activación de la clave de cifrado

Las máquinas virtuales han alcanzado un nuevo nivel de complejidad, lo que hace necesario un nuevo nivel de ataque. Los controladores son más complejos de analizar; además, no se pueden analizar de forma aislada (por ejemplo, no se sabría el valor de la clave de cifrado). Un atacante quiere limitar el análisis manual al mínimo.

Sin embargo, la mayoría de las veces se requiere una revisión manual para “capturar” el comportamiento general de una máquina virtual y así inferir posibles ataques a ella.

Uno de los primeros lugares de interés es el stub de invocación de la VM, es decir, la transición entre el código nativo (no virtualizado) y la VM/intérprete. La inicialización del contexto indica la naturaleza de la asignación entre la arquitectura nativa y la arquitectura de la VM. Puede ser una copia exacta de los registros nativos a los registros de la VM o algo más complicado. Además, en este punto, es útil distinguir (tanto como sea posible) entre las variables de inicialización obligatorias (como la clave de una VM, el número de controlador o el punto de entrada) para las que se requiere un valor numérico, y las variables adicionales que se pueden mantener simbólicas.

El segundo lugar de mayor interés es el despachador de la máquina virtual (si existe). La mayoría de las veces, hay un único punto de despacho que básicamente recupera el siguiente controlador de una tabla de controladores en función de un índice almacenado en algún lugar dentro del controlador de la máquina virtual.

Contexto. Una pregunta a responder es, ¿cuál es la condición de interrupción de este bucle de ejecución? En esta configuración es posible considerar la VM como una generalización del aplanamiento de código. El aplanamiento de código virtualiza solo el flujo de control, mientras que la VM virtualiza tanto el flujo de control como el de datos. Además, el aplanamiento de código generalmente solo opera a nivel de función única, mientras que la VM opera a nivel de programa. La otra posibilidad es un envío distribuido, por el cual cada controlador es responsable de actualizar el puntero de instrucción de la VM y vincularlo al siguiente controlador.

Éstas son ideas generales; cada inversor tiene sus propios trucos y abstracciones del problema.

Usando Metasm

El enfoque que vamos a explorar se basa en el marco Metasm.

Se basa en la ejecución simbólica para hacer que la máquina virtual (es decir, el intérprete) procese el bytecode (con respecto a los datos estáticos) y calcule el programa residual. Por un lado, existe un programa (el intérprete); por otro lado, existen sus datos estáticos (el bytecode); especializaremos el programa con respecto a sus datos estáticos.

318 Capítulo 5 ■ Ofuscación

Considerar un programa ofuscado y sus datos como un todo sería demasiado complejo, la solución es dividir este complejo problema en múltiples subproblemas más simples.

Considerar el nivel de manejadores de instrucciones de la máquina virtual proporciona una granularidad mucho más apropiada. De ahora en adelante, consideraremos los manejadores de instrucciones como la unidad semántica más pequeña del intérprete.

Partiendo de esa premisa básica, se puede aplicar el siguiente pseudoalgoritmo:

1. Captura el contexto actual (código de bytes de la máquina virtual, parámetros estáticos,... medio ambiente, etc.).
2. Desmontar el controlador actual.
3. Desofuscar el código, si es necesario.
4. Calcular su semántica (es decir, función de transferencia).
5. Generar salida a partir de la semántica resuelta.
6. Calcular el siguiente estado (es decir, aplicar la función de transferencia al contexto actual).
7. Si el despachador del manejador no alcanza una condición de interrupción/salida, repita desde el paso 1.

Opcionalmente, es posible regenerar código nativo a partir de la función de transferencia calculada en el paso 4. Como lo expresó Futamura21, dado un intérprete de Linterpreted escrito en un lenguaje nativo dado Lnative , es posible calcular automáticamente un compilador de Linterpreted a Lnative.

Esto es adecuado para los conceptos teóricos. Ahora supongamos que te enfrentas a un instancia de una máquina virtual. ¿Por dónde empezar? Tomemos un ejemplo práctico.

El siguiente script utiliza Metasm para compilar y luego desensamblar lo que podría ser un controlador de una máquina virtual. Para simplificar, este ejemplo solo se ocupa de la parte de la máquina virtual (por lo tanto, el código no se ofusca). El código del controlador se encuentra en 10000000h, mientras que una sección de datos que contiene el código de bytes del controlador se encuentra en 1a000000h:

```
# codificación: ASCII-8BIT #!/usr/bin/
env ruby

requiere "metasm" incluye
Metasm

$SPAWN_GUI = falso
CÓDIGO_BASE_ADDR = 0x10000000
Dirección base de tabla H = 0x18000000
DIRECCIÓN_BASE_DATOS = 0x1A000000
INYECTAR_MÁXIMO_ITER = 0x20

REGLAS NATIVAS = [.eax, .edx, .ecx, .ebx, .esp, .ebp, .esi, .edi]

definición mostrar(bd)
    bd.each{|clave,valor| pone
        "#[Expresión[clave]] => #[Expresión[valor]]"
    fin
}
```

```
# produce el código x86 del controlador
sc = Metasm::Shellcode.assemble(Metasm::Ia32.new, <<EOS)

baja calidad

movimiento ecx, eax
xor ecx, ebp
movzx eax, c
empujar eax
mover eax, [edi+eax]

movzx edx, capítulo
edición mov, [edi+edx]
xor eax, edx

movimiento
de edición pop [edi+edx], eax

lodsd xor ebp, 0x35ef6a14
xor-ax, ebp
jmp [#{DIRECCIÓN_BASE_DE_TABLE}+eax*4]
EOS

manejador = sc.encode_string

# sección de datos hexadecimales
sección_datos_hex = "\xA3\xCB\xDB\x5F\x60\xBD\x34\x6A"

# agregar una sección de código
dasm = sc.init_disassembler
dasm.add_sectionEncodedData.new(controlador), CODE_BASE_ADDR)

# agregar una sección de datos
dasm.add_sectionEncodedData.new(sección_datos_hex), DIRECCIÓN_BASE_DATOS)

# desensamblar el código del controlador
dasm.desensamblar_rápido_profundo(DIRECCIÓN_BASE_CÓDIGO)
```

Lo primero que hay que hacer es obtener automáticamente la semántica de ese controlador. Como se mencionó anteriormente, Metasm ofrece un método llamado `code_binding` que calcula la transferencia de función (la terminología de Metasm es vinculación) de un conjunto de instrucciones. De esta manera puedes escribir lo siguiente:

```
# Calcular la semántica del controlador bb =
dasm.dll_at(CODE_BASE_ADDR).block.start_addr =
bb.list.first.address end_addr = bb.list.last.address

coloca "[+]" desde 0x#{start_addr.to_s(16)}, hasta 0x#{end_addr.to_s(16)}" binding = dasm.code_binding(start_addr,
end_addr) display(binding)
```

Lo anterior produce el siguiente resultado:

```
[+] desde 0x10000000, hasta la dirección 10000021
dword ptr [esp] => (dword ptr [esi]*ebp)&0ffh dword ptr [edi+((dword ptr
[esi]*ebp)&0ffh)] =>
```

320 Capítulo 5 ■ Ofuscación

```

palabra clave [edi+((palabra clave[esi]^ebp)&0ffh)] ^
dword ptr [edi+(((dword ptr[esi]>>8)*(ebp>>8))&0ffh)]
eax => (palabra compuesta [esi+4]^(ebp^35ef6a14h))&0xffffffff ecx => (palabra compuesta
[esi]^ebp)&0xffffffff edx => (palabra compuesta [esi]^ebp)&0ffh ebp
=> (ebp^35ef6a14h)&0xffffffff esi => (esi+8)&0xffffffff

```

Tanto desde el ensamblaje como desde el enlace, se puede decir lo siguiente sobre la máquina virtual:

- Parece utilizar un almacén de claves de giro en EBP. (Observe cómo se utiliza para descifrar) los operandos de la instrucción a partir del código de bytes).
- Su contexto parece estar señalado por EDI.

Es un buen comienzo, pero aún estás lejos del objetivo. Todavía estás estancado en el nivel de ensamblaje, así que retrocedamos un paso y consideremos la inicialización de la máquina virtual, que hemos identificado de la siguiente manera:

```

Empujar
pop [edición]
pop [edi+0x4]
pop [edi+0x8]
pop [edi+0xC]
hacer estallar [edi+0x10]
hacer estallar [edi+0x14]
hacer estallar [edi+0x18]
hacer estallar [edi+0x1C]

```

Primero, los registros nativos se insertan en la pila y luego se leen desde la pila hacia un área de memoria a la que apunta EDI, que a su vez es responsable de apuntar al contexto de la máquina virtual. Esta información le permite crear una asignación entre los elementos internos simbólicos de la máquina virtual y la expresión de ensamblaje:

```

vm_simbolismo = {
    :eax => :nmanejador,
    :ebp => :vmkey,
    :esi => :bytecode_ptr,
    Indirección[[:edi], 4, nil] => :vm_edi,
    Indirección[[:edi, :+, 4], 4, nil] => :vm_esi,
    Indirección[[:edi, :+, 8], 4, nil] => :vm_ebp,
    Indirección[[:edi, :+, 0xC], 4, nil] => :vm_esp,
    Indirección[[:edi, :+, 0x10], 4, nula] => :vm_ebx,
    Indirección[[:edi, :+, 0x14], 4, nula] => :vm_edx,
    Indirección[[:edi, :+, 0x18], 4, nula] => :vm_ecx,
    Indirección[[:edi, :+, 0x1c], 4, nula] => :vm_eax,
}

```

Este simbolismo se inyecta en el enlace (cada ocurrencia de un valor izquierdo se reemplaza por su valor derecho asociado). Las expresiones tienen un método especial llamado bind que hace exactamente eso. El siguiente ejemplo primero define un valor simbólico.

expresión: la suma de dos términos, uno de ellos es una indirección; y están involucrados dos símbolos, :a y :b. A continuación, el símbolo :a se asocia (se vincula) con el valor 1000h:

```
expr = Expresión[[:a, 4], :+, :b] sym = { :a => 0x1000 }

pone expr
>> dword ptr [a]+b

pone expr.bind(sym)
>> dword ptr [1000h]+b
```

Puede generalizar esto para cada expresión del enlace. Esta asignación es la clave que permite abstraer el código de la máquina virtual desde su nivel de implementación hasta el nivel de semántica de la máquina virtual. Además, un efecto secundario positivo de este paso es a menudo una reducción significativa de la complejidad del enlace. El nuevo enlace es el siguiente:

```
[+] enlace simbólico dword
ptr [esp] => (dword ptr [bytecode_ptr]^vmkey)&0ffh dword ptr [edi+((dword ptr
[bytecode_ptr]^vmkey)&0ffh)] => dword ptr [edi+dword ptr [bytecode_ptr]^vmkey)&0ffh]]^
dword ptr [edi+(((dword ptr [bytecode_ptr]>>8)^vmkey>>8))&0ffh]

nhandler => (dword ptr [bytecode_ptr+4]^vmkey^35ef6a14h))&0xffffffff vmkey =>
(vmkey^35ef6a14h)&0xffffffff bytecode_ptr =>
(bytecode_ptr+8)&0xffffffff
```

Hemos avanzado, pero el cifrado sigue siendo problemático y no podemos avanzar si se desconoce el contexto de la máquina virtual en el momento de ejecución de este controlador: el puntero de código de bytes, la clave de giro y, opcionalmente, el número del controlador son todos valores obligatorios. Suponiendo que conoce estos valores (se encuentra en el punto de entrada de la máquina virtual o ha rastreado dinámicamente la máquina virtual hasta ese punto), puede definir un pseudocontexto:

```
contexto = {
    :nhandler => 0x84,
    :vmkey => 0x5fdbd7b7,
    :bytecode_ptr => DIRECCIÓN_BASE_DE_DATOS,
    :virt_eax => 0xffffeeee,
    :virt_ecx => 0,
    :virt_edx => 0x41414141,
    :virt_ebx => 1,
    :virt_edi => :virt_edi,
}
```

Tenga en cuenta que el contexto contiene valores simbólicos y numéricos. Por ejemplo, nhandler se define como igual a 84h, mientras que el registro virt_edi de la máquina virtual es simbólico.

Luego, se inyecta el contexto dentro del enlace, así como el simbolismo definido previamente. En la práctica, se trata de un proceso iterativo, pero no nos sobrecarguemos con detalles de implementación. Las expresiones se resuelven progresivamente y

322 Capítulo 5 ■ Ofuscación

Se reduce con respecto a todos los valores conocidos, que incluyen el contexto actual y los datos del programa (es decir, el bytecode). Al final, se obtiene un enlace resuelto, que en realidad representa el contexto de la máquina virtual después de la ejecución del controlador. A ese paso lo llamamos ejecución simbólica:

```
[+] solucionador de enlaces
[+] tecla: dword ptr [esp]
=> clave resuelta: dword ptr [esp]

[+] valor: (dword ptr [bytecode_ptr]^vmkey)&0ffh [+]
lectura de memoria resuelta
en 0x1a000000, tamaño 4 [+]
valor 5fdbcba3h

=> valor resuelto: 14h

Tecla [+]: dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0ffh)]
[+] lectura de memoria resuelta en 0x1a000000, tamaño 4 [+]
valor
5fdbcba3h
=> clave resuelta: virt_edx

[+] valor: dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0ffh)]^
dword ptr [edi+(((dword ptr [bytecode_ptr]>>8)^vmkey>>8))&0ffh]
[+] lectura de memoria resuelta en 0x1a000000, tamaño 4 [+]
valor
5fdbcba3h
[+] lectura de memoria resuelta en 0x1a000000, tamaño 4 [+]
valor
5fdbcba3h
=> valor resuelto: 0beafbeafh

[+] tecla: nhandler
=> clave resuelta: nhandler

[+] valor: (dword ptr [bytecode_ptr+4])^(vmkey^35ef6a14h)&0fffffff [+]
lectura de memoria resuelta en 0x1a000004,
tamaño 4 [+]
valor 6a34bd60h

=> valor resuelto: 0c3h

[*] tecla: vmkey => clave
resuelta: vmkey [+]
valor:
(vmkey^35ef6a14h)&0fffffff => valor resuelto: 6a34bd60h

[+] tecla: bytecode_ptr
=> clave resuelta: bytecode_ptr [+]
valor:
(bytecode_ptr+8)&0fffffff
=> valor resuelto: 1a000008h

[+] solucionado el enlace
virt_edx => 0beafbeafh
manejador => 0c3h
tecla virtual => 6a34bd60h
código_de_bytes_ptr => 1a000008h
```

El solucionador de expresiones ayuda a calcular los valores finales de la vinculación del primer controlador: el siguiente controlador que se ejecutará, así como el valor actualizado de la clave de giro. Actualizar el contexto es una operación trivial:

```
contexto_actualizado = contexto.actualizar(enlace_resuelto)

pone "\n[+] contexto actualizado"
display(updated_context)

[+] contexto actualizado
manejador => 0c3h
tecla virtual => 6a34bda3h
bytecode_ptr => 1a000008h virt_eax
=> 0ffeefeeh virt_ecx => 0
virt_edx =>
0beafbeafh virt_ebx => 1 virt_edi
=> virt_edi
```

Puede repetir este proceso y recorrer todo el gráfico de flujo de control de la máquina virtual. Este es un resultado bastante apreciable; sin embargo, los valores numéricos puros ocultan un poco la semántica del controlador. Además, uno de los objetivos es regenerar el código ensamblador nativo equivalente a la ejecución contextualizada del controlador.

Un truco que solemos utilizar al analizar una VM con Metasm es proceder a una doble ejecución simbólica para cada manejador: una con el contexto completo (principalmente valores numéricos, utilizados para actualizar el contexto) y otra con un contexto casi puramente simbólico (utilizado para extraer la semántica de alto nivel). El siguiente ejemplo de código demuestra el uso de un contexto simbólico:

```
contexto_simbólico = { :nhandler
=> 0x84,
:vmkey => 0x5fdbd7b7,
:bytecode_ptr => DIRECCIÓN_BASE_DE_DATOS,
:virt_eax => .virt_eax,
:virt_ecx => .virt_ecx,
:virt_edx => .virt_edx,
:virt_ebx => .virt_ebx,
:virt_edi => .virt_edi,
}

enlace_simbólico_resuelto = sym_exec(contexto_simbólico, enlace_simbólico,
simbolismo_vm)

coloca "\n[+] enlace resuelto" display(solved_symbolic_binding)
```

Esta vez el resultado es el siguiente:

```
[+] solucionador de enlaces

[+] tecla: dword ptr [esp]
=> clave resuelta: dword ptr [esp]
```

324 Capítulo 5 ■ Ofuscación

```
[+] valor: (dword ptr [bytecode_ptr]^vmkey)&0fffh [+] lectura de memoria resuelta
en 0x1a000000, tamaño 4 [+] valor 5fdbcba3h

=> valor resuelto: 14h

Tecla [+]: dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0fffh)]
[+] lectura de memoria resuelta en 0x1a000000, tamaño 4 [+] valor
5fdbcba3h
=> clave resuelta: virt_edx

[+] valor: dword ptr [edi+((dword ptr [bytecode_ptr]^vmkey)&0fffh)]^ dword ptr edi+((dword ptr
[bytecode_ptr]>>8)^vmkey>>8)&0fffh) [+] lectura de memoria resuelta en 0x1a000000, tamaño 4 [+] valor
5fdbcba3h

[+] lectura de memoria resuelta en 0x1a000000, tamaño 4 [+] valor
5fdbcba3h
=> valor resuelto: virt_edx^virt_eax

[+] tecla: nhandler
=> clave resuelta: nhandler

[+] valor: (dword ptr [bytecode_ptr+4])^(vmkey^35ef6a14h)&0xffffffff
[+] lectura de memoria resuelta en 0x1a000004, tamaño 4 [+] valor
6a34bd60h
=> valor resuelto: 0c3h

Tecla [+]: vmkey
=> clave resuelta: vmkey

[+] valor: (vmkey^35ef6a14h)&0xffffffff
=> valor resuelto: 6a34bda3h

[+] tecla: bytecode_ptr => tecla
resuelta: bytecode_ptr

[+] valor: (bytecode_ptr+8)&0xffffffff
=> valor resuelto: 1a000008h

[+] Se solucionó el enlace
virt_edx => virt_edx^virt_eax
manejador => 0c3h
tecla virtual => 6a34bda3h
código_de_bytes_ptr => 1a000008h
```

Finalmente, puedes simplemente rechazar el control de VM desde el enlace resuelto, lo que te dejará con lo siguiente:

vm_edx => vm_edx^vm_eax

A partir de ese resultado final, la regeneración de código nativo es bastante sencilla. Este proceso se repite en el gráfico de flujo de control de la máquina virtual para cada controlador. Como observación general, al utilizar esta técnica, una elección delicada es cuándo mantener los valores simbólicos y cuándo reducirlos a valores numéricos. La primera opción

El primero favorece la recuperación de la semántica de alto nivel, mientras que el segundo permite una recuperación más sencilla del flujo de control de la máquina virtual. Al final, también es posible continuar con la salida. Imagine una máquina virtual que parezca un intérprete basado en pila:

```
01: enviar vm_edx  
02: agregar [esp], vm_eax  
03: pop vm_edx
```

El uso de métodos de desofuscación clásicos, como las optimizaciones del compilador, reescribiría las tres líneas anteriores de la siguiente manera:

```
01: agregar vm_eax, vm_eax
```

Tenga en cuenta que el propio código de bytes podría haberse ofuscado.

Desofuscación de aplanamiento de código

Como se mencionó anteriormente, el aplanamiento de código puede considerarse como una virtualización parcial (solo se virtualiza el flujo de control). Por lo tanto, las técnicas descritas para el análisis de VM, y especialmente la ejecución simbólica, también se pueden aplicar para el análisis de código aplanado.

Todavía existen algunas dificultades que son propias de esta técnica:

- La transformación de aplanamiento de código se aplica con mayor frecuencia a nivel de función y, a veces, puede haber más de un "nodo" aplanado en la misma función. En general, eso significa que hay múltiples instancias de las técnicas; por lo tanto, una herramienta debe ser a prueba de balas y completamente automática.
- Discernir entre el código original de una función y el código del despachador agregado es difícil cuando la implementación de aplanamiento de código es robusta (los flujos de datos del programa y del despachador están firmemente entrelazados/son interdependientes).
- La transformación inversa no es fácil de implementar.

Usando VxStripper

Para ilustrar el uso de VxStripper en un ejemplo de "juguete" simple, considere el siguiente programa (que muestra $y = 22$) después de desempaquetarlo y reconstruirlo:

```
..... !punto de entrada:  
..... ! empajar ebp  
401001 !movimiento      esp, esp  
401003 !subtítulo       esp, 10h  
401006 !movimiento       palabra clave d [ebp-4], 0  
40100d !movimiento       palabra clave d [ebp-0ch], 2  
401014 !movimiento       palabra clave d [ebp-8], 0ah  
40101b !  
..... ! loc_40101b: ..... ! cmp  
dword ptr [ebp-0ch], 6  
40101f!          JNL      ubicación_40108c  
401021 !movimiento       EAX, [ebp-0ch]  
401024 !movimiento       [ebp-10h], eax  
401027 !movimiento       ecx, [ebp-10h]  
40102a !subtítulo        ecx, 2
```

326 Capítulo 5 ■ Ofuscación

40102d ! movimiento	[ebp-10h], ecx
401030 ! 401034 !	palabra clave d [ebp-10h], 3
Si	loc_40108a
401036 ! movimiento	edx, [ebp-10h]
401039 ! saltar	palabra clave d [edx*4+data_4010a4]
401040 movimiento	palabra clave d [ebp-4], 2
401047 movimiento	palabra clave d [ebp-0ch], 3
40104e saltar	loc_40108a
401050 cmp	palabra clave d [ebp-8], 0
401054 jng	40105fh
401056 movimiento	palabra clave d [ebp-0ch], 4
40105d saltar	401066h
40105f movimiento	palabra clave d [ebp-0ch], 6
401066 saltar	loc_40108a
401068 movimiento	eax, [ebp-4]
40106b agregar	Ej., 2
40106e movimiento	[ebp-4], eax
401071 movimiento	palabra clave d [ebp-0ch], 5
401078 saltar	loc_40108a
40107a movimiento	ecx, [ebp-8]
40107d sub	ecx, 1
401080 movimiento	[ebp-8], ecx
401083 movimiento	palabra clave d [ebp-0ch], 3
40108a !	
..... ! loc_40108a: ! jmp loc_40101b	
40108c !	
..... ! loc_40108c: edx, [ebp-4]	
..... ! movimiento	
40108f ! empujar 401090 !	educación
empujar	cadena_yd_402008
401095 llamada	dword ptr [msvrt.dll.printf]
40109b ! añadir	esp, 8
40109e xor	Ea, Ea
4010a0 ! movimiento	esp, ebp
4010a2 ! estallar 4010a3 !
ret	devuelve 0;

El gráfico de flujo de control (CFG) de dicho programa es aplanado.

La ejecución del módulo de normalización produce (cuando no se aplican todos los optimizaciones) el siguiente código:

..... ! empujar	fácil
4011f1 ! movimiento	palabra clave [esp-0ch], 0ah
4011f9 ! movimiento	palabra clave d [esp-8], 4
401201 ! diciembre	palabra clave d [esp-0ch]
401205 ! añadir	palabra clave d [esp-8], 2
40120a dic	palabra clave d [esp-0ch]
40120e ! añadir	palabra clave d [esp-8], 2
401213 ! diciembre	palabra clave d [esp-0ch]
401217 ! añadir	palabra clave d [esp-8], 2
40121c diciembre	palabra clave d [esp-0ch]
401220 ! añadir	palabra clave d [esp-8], 2

401225 ! diciembre	palabra clave d [esp-0ch]
401229 ! añadir	palabra clave d [esp-8], 2
40122e ! diciembre	palabra clave d [esp-0ch]
401232 ! añadir	palabra clave d [esp-8], 2
401237 ! diciembre	palabra clave d [esp-0ch]
40123b ! añadir	palabra clave d [esp-8], 2
401240 ! diciembre	palabra clave d [esp-0ch]
401244 ! añadir	palabra clave d [esp-8], 2
401249 ! diciembre	palabra clave d [esp-0ch]
40124d ! añadir	palabra clave d [esp-8], 2
401252 ! diciembre	palabra clave d [esp-0ch]
401256 !movimiento	eax, [esp-8]
40125a !movimiento	[esp-18h], eax
40125e !movimiento	dword ptr[esp-1ch], strz_yd_402010
401266 !movimiento	ebp, esp
401268 !!ea	Eax, [esp-1ch]
40126c!movimiento	especialmente, eax
40126e !!llamada	crtdll.dll:printf_4012d8
401273 !movimiento	esp, ebp
401275 !movimiento	ebp, esp
401277 !!ea	eax, [esp+8]
40127b!movimiento	especialmente, eax
40127d!movimiento	esp, ebp
40127f ! xor	Ea, Ea
401281 !pop	educción
401282 ! retirado	

Tenga en cuenta que la generación dinámica de código realizada por VxStripper desdobra naturalmente el código desdoblado. La aplicación de transformaciones de optimización estándar da como resultado un programa sin esta ofuscación:

..... ! empujar	facil
4011f1 !movimiento	palabra clave d [esp-18h], 16h
4011f9 !movimiento	dword ptr[esp-1ch], strz_yd_402010
401201 !movimiento	ebp, esp
401203 !!ea	Eax, [esp-1ch]
401207 !movimiento	especialmente, eax
401209 !!llamada	crtdll.dll:printf_4012d8
40120a !movimiento	esp, ebp
401210 !movimiento	ebp, esp
401212 !!ea	eax, [esp+8]
401216 !movimiento	especialmente, eax
401218 !movimiento	esp, ebp
40121a ! xor	Ea, Ea
40121c ! estallar	educción
40121d ! ret	

Aunque aún queda trabajo por hacer antes de obtener un software que soporte el conjunto de herramientas de protección software utilizables por los autores de malware, estos primeros resultados nos animan a continuar con el estudio de métodos genéricos de descompresión y normalización, con el objetivo de automatizar al máximo las tareas realizadas por un analista.

Esta herramienta proporciona un software autosuficiente para el análisis de malware. Sin embargo, uno de los objetivos futuros de este proyecto es permitir que la herramienta interactúe con otras herramientas de análisis. Por diseño, esta herramienta podría colaborar con cualquier herramienta de análisis de software basada en la cadena de compilación LLVM.

La cadena de compilación LLVM y las numerosas herramientas basadas en LLVM ya proporcionan una gran biblioteca de análisis de programas que pueden usarse juntos para derrotar los mecanismos de protección contra malware.

Además, Vellvm (Verifi ed LLVM) se puede utilizar para extraer formalmente implementaciones verificadas de pasos de desofuscación implementados por VxStripper. Además del análisis de amenazas de malware, también se pueden imaginar otros usos de esta herramienta, como la extracción de esquemas de detección, las protecciones de software y el análisis de la solidez del software antivirus.

Estudio de caso

El ejemplo que utilizaremos para este estudio de caso es en realidad un crackme publicado originalmente en Crackmes.de por quetz en 2007. Aunque es "solo" un crackme, presenta la mayoría de los conceptos que uno encontraría en una protección de nivel profesional.

Entre otras alegrías contiene las siguientes:

- Aplanamiento de código
- Codificación de variables
- Virtualización de código

Así es como el autor presenta su desafío:

Últimamente, la protección mediante análisis estático se vuelve cada vez más popular. Casi todos los protectores emplean algún tipo de ofuscación, máquina virtual, etc. Este keygenme es un intento de mostrar qué sucede si se abusa de la idea de ofuscación. ¿Puede un ser humano analizar eficazmente ese código? ¿Quizás con la ayuda de una herramienta?

—http://crackmes.de/users/quetz/q_keygenme_1.0/

Afortunadamente, tenemos herramientas y en esta sección las usaremos. Antes de comenzar, le recomendamos que no mire la sección de símbolos que se encuentra dentro del binario. Por cierto, las versiones anteriores de IDA Pro (quizás anteriores a la 6.2) no cargaban estos símbolos y el autor de estas líneas, felizmente, no los buscó .

Primeras impresiones

Al iniciar el ejecutable, se le ofrece la oportunidad de ingresar un nombre de usuario y una contraseña. Después de hacer clic en el botón Verificar, aparece un cuadro de mensaje que muestra la validez de sus credenciales.

Si ha leído atentamente los capítulos anteriores de este libro, probablemente ya haya activado su desensamblador favorito y haya apuntado al DialogProc de la GUI. función de devolución de llamada.

Veamos primero la arquitectura general del código protegido. El gráfico de flujo de control es demasiado desordenado para ser código generado por el compilador: tenemos un DialogProc ofuscado que llama a dos funciones con código plano (func1@0x430DB0, func2@0x431E00). Estas dos funciones llaman a lo que parece ser una máquina virtual (vm@0x401360).

Ya hemos hablado del único punto de despacho de la VM; este es bastante sencillo de detectar (busque una tabla de salto importante en ausencia de otras pistas):

```
01:.texto:00401F20  
  
02: movimiento ebp, [esp+13Ch]  
03: cmp      ebp, 3E2Dh; cambia 15918 casos  
04: si corto loc_401F36  
05: jmp ds:off_43D000[ebp*4]; cambiar salto
```

Estos son nuestros dos primeros y casi gratuitos conocimientos sobre la VM: almacena su número de controlador actual en [ESP+13Ch] y hay 15.918 entradas en el despachador (por ahora, no podemos concluir si todos son controladores diferentes).

Antes de ponernos manos a la obra, podemos intentar realizar un análisis de caja negra de las funciones func1 y func2 . Sabemos que func1 y func2 llaman a la máquina virtual; simplemente registramos cada llamada a la máquina virtual y, especialmente, el número del primer controlador que se llama (una especie de punto de entrada en el código de la máquina virtual). Esto parece bastante trivial, pero nunca se debe ignorar lo que está al alcance de la mano.

Los resultados son inmediatamente reveladores: func2 se llama antes que func1, por lo que comenzaremos con func2. En cuanto se observan los registros del punto de entrada de la máquina virtual, se destacan estos patrones:

- 546h-0BFFh-7B2h-9A2h-405h-919h-3B9h—624 veces
- 0CF5h-15Eh-184h-39Ch-5B0h-3C0h-0F75h—624 veces
- 0A06h-0xA29h-0x268h-0xCB3h—227 veces
- 736h-13Ah-1EBh-897h—396 veces
- 150h-8ABh-843h-697h-474h—200 veces

En realidad, ya es mucha información. Si miras la sección .data , Te espera una pista extra:

```
01: .datos:0043CA44 dword_43CA4402: .datos:0043CA48 dd 9908B0DFh
```

¿De dónde viene 9908B0DFh ? ¿Y 624? ¿ Y 227? Bueno, o estás muy familiarizado con los generadores de números aleatorios o buscas estos valores; identifican un algoritmo de generador pseudoaleatorio Mersenne Twister. 624 y 397 son los parámetros del período, mientras que 9908B0DFh es una constante utilizada durante la generación de números.

330 Capítulo 5 ■ Ofuscación

Hemos identificado una debilidad crítica de la protección: el código virtualizado filtra cierta información sobre la estructura del algoritmo protegido, lo que hace que sea trivial recuperar iteraciones del bucle. Sin embargo, ¿tenemos un código virtualizado bien diseñado que se anula con un punto de interrupción? Todavía no. Tenemos un candidato a algoritmo, pero necesita ser confirmado.

Nuevamente, nunca se debe aplicar ingeniería inversa a un código cuando es posible adivinar (y validar) información. En este caso, un análisis de caja negra revela mucho simplemente observando las entradas/salidas de las funciones func1 y func2. Una estrategia básica como esta o un análisis diferencial de la ejecución de la máquina virtual a veces puede ser de gran ayuda. Refinemos nuestro análisis de estas dos funciones:

- función2

- Entrada: dos argumentos: una dirección en la pila que parece ser una matriz de números enteros y un valor de 32 bits que parece depender de la longitud del nombre.
- Salida: nada destacable excepto que la matriz de números enteros ha sido actualizado.
- Ocurrencia: se llama una sola vez, al principio y antes de func1.
- Adivinación: inicialización del Mersenne Twister; la matriz es en realidad el estado del PRNG. El valor de 32 bits es la semilla de inicialización. Esto se puede validar aún más haciendo coincidir los parámetros del bucle (obtenidos de los registros) con una función de inicialización estándar.

- función1

- Entrada: dos argumentos: la dirección del estado PRNG (supuesto) y un valor de 32 bits que parece ser una letra del nombre de usuario.
- Salida: Devuelve un valor aleatorio de 32 bits.
- Ocurrencia: se llama 100 veces.
- Guess: función similar a rand32 del Twister de Mersenne.

Análisis de la semántica de los controladores

Ahora es el momento de analizar la máquina virtual. El despachador principal ya se ha encontrado en [ESP+13ch]. A menudo es una buena idea verificar manualmente algunos controladores para ver cómo acceden al contexto de la máquina virtual, cómo actualizan el contador del programa y/o o puntero de bytecode, y así sucesivamente.

Este proceso se puede aplicar a un controlador aleatorio, por ejemplo, el que comienza en 0x41836c:

```
01: .text:0041836C loc_41836C:
02:             ; DATOS XREF: .rdata:off_43D000
03:             ; tabla de salto 00401F2F casos 2815,4091
04: .text:0041836C movzx ecx, [esp+3D8h+var_2A2]
```

```

05: .text:00418374 mov esi, 97Fh
06: .text:00418379 mov ebx, [esp+3D8h+var_3C8]
07: .text:0041837D movzx edi, [esp+3D8h+var_29E]
08: .text:00418385 agregar [esp+3D8h+var_3C0], 94Eh
09: .text:0041838D imul eax, ecx, 1 canal
10: .text:00418390 sub [esp+3D8h+var_3C4], 0FF0h
11: .text:00418398 imul ecx, edi, 5DDh
12: .text:0041839E mov [esp+3D8h+var_37C], esi
13: .text:004183A2 lea edx, [eax+ebx+5C8h]
14: .text:004183A9 mov ebx, 97Fh
15: .text:004183AE mov [esp+3D8h+var_3C8], edx
16: .text:004183B2 sub ebx, ecx
17: .text:004183B4 lea edx, [ebp+ebx+29Dh+var_8BC]
18: .text:004183BB mov [esp+3D8h+var_378], ebx
19: .text:004183BF mov [esp+3D8h+var_380], ebx
20: .text:004183C3 mov [esp+3D8h+var_29D+1], edx
21: .text:004183CA jmp loc_401F20

```

Consigamos ayuda de Metasm. Como se mostró anteriormente, podemos usar el código: método de enlace para calcular la semántica de un fragmento de código:

```

dword ptr [esp+10h] => 1ch*byte ptr [esp+136h]+dword ptr [esp+10h]+5c8h
dword ptr [esp+14h] => dword ptr [esp+14h]-0ff0h dword ptr [esp+18h]
=> dword ptr [esp+18h]+94eh dword ptr [esp+58h] => -5ddh*byte ptr
[esp+13ah]+97fh dword ptr [esp+5ch] => 97fh dword ptr [esp+60h] => -5ddh*byte
ptr [esp+13ah]+97fh

esi => 97fh

edi => byte ptr [esp+13ah]&0xffffffff

```

El contexto de la máquina virtual se almacena en la pila y los registros no pasan valores entre los controladores; eso significa que se pueden descartar todas las modificaciones de los registros para obtener una visión más clara:

```

dword ptr [esp+10h] => 1ch*byte ptr [esp+136h]+dword ptr [esp+10h]+5c8h dword ptr [esp+14h] => dword
ptr [esp+14h]-0ff0h dword ptr [esp+18h] => dword ptr [esp+18h]+94eh
dword ptr [esp+58h] => -5ddh*byte ptr [esp+13ah]+97fh dword ptr
[esp+5ch] => 97fh dword ptr [esp+60h] => -5ddh*byte ptr [esp+13ah]+97fh dword
ptr [esp+13ch] => ebp-5ddh*byte ptr
[esp+13ah]+360h

```

Ya sabemos que el número del manejador se almacena en [ESP+13Ch]. El manejador lo actualiza . Su valor final depende del valor del byte ptr [ESP+13ah]. Al analizar algunos otros manejadores, podemos suponer que es un valor booleano y que hay otros valores booleanos almacenados en el contexto. Este se almacena en la segunda posición y se llamará flag2.

[ESP+58h], [ESP +5Ch] y [ESP +60h] están estrechamente vinculados con el cálculo del número de controlador. Contienen respectivamente la diferencia entre el número de controlador antiguo y el nuevo en caso de que la condición (aquí flag2) sea verdadera o falsa.

[ESP +10h], [ESP +14h] y [ESP +18h] también son de gran interés. Son actualizados por casi todos los controladores y se supone que descifran el código de bytes: acceden a la gran tabla de constantes no definidas almacenada en la sección .data). En realidad, son como una tecla en ejecución; se llamarán respectivamente key_a, key_b y key_c.

El siguiente es un ejemplo de uso de clave tomado del controlador 0xa0a en la dirección 0x427b17:

```
nHandler => dword ptr [4*tecla_c+436010h]^ dword ptr
[4*tecla_b+436010h]^ dword ptr
[4*tecla_a+436010h]
```

Los nombres se pueden inyectar dentro del enlace del controlador, lo que lo hace más comprensible (incluso si ese no es nuestro objetivo principal aquí) y fácil de manipular:

```
clave_a => 1ch*bandera6+clave_a+5c8h
clave_b => clave_b-0ff0h
clave_c => clave_c+94eh
delta_verdadero => -5ddh*bandera2+97fh
delta_falso => 97fh delta =>
-5ddh*bandera2+97fh
nHandler => ebp-5ddh*bandera2+360h
```

Este manejador tiene casi la semántica de un salto condicional. Analizando algunos otros manejadores, es posible recuperar y validar un mapeo de las variables simbólicas de la máquina virtual, que serán representadas por un objeto hash:

```
VM SIMBÓLICA = {
    Indirección[Expresión[.esp, +, 0x10], 4, nil] => :key_a,
    Indirección[Expresión[.esp, +, 0x14], 4, nil] => :key_b,
    Indirección[Expresión[.esp, +, 0x18], 4, nil] => :key_c,
    Indirección[Expresión[.esp, +, 0x58], 4, nil] => :delta,
    Indirección[Expresión[.esp, +, 0x5c], 4, nil] => :delta_false,
    Indirección[Expresión[.esp, +, 0x60], 4, nil] => :delta_true,
    Indirección[Expresión[.esp, +, 0x134], 1, nulo] => :flag8,
    Indirección[Expresión[.esp, +, 0x135], 1, nulo] => :flag7,
    Indirección[Expresión[.esp, +, 0x136], 1, nil] => :flag6,
    Indirección[Expresión[.esp, +, 0x137], 1, nulo] => :flag5,
    Indirección[Expresión[.esp, +, 0x138], 1, nil] => :flag4,
    Indirección[Expresión[.esp, +, 0x139], 1, nil] => :flag3,
    Indirección[Expresión[.esp, +, 0x13a], 1, nil] => :flag2,
    Indirección[Expresión[.esp, +, 0x13b], 1, nil] => :flag1,
    Indirección[Expresión[.esp, +, 0x13c], 4, nil] => :nHandler
}
```

Otras ubicaciones de memoria no parecen tener un propósito específico; pueden considerarse/mapear como registros de propósito general. Con este mapeo, podemos

tenemos todo lo que necesitamos para procesar una ejecución simbólica de la VM (es decir, ejecución paso a paso de la semántica del controlador).

Ejecución simbólica

Para procesar una ejecución simbólica, es necesario tener algunas pistas sobre el contexto de inicialización de la máquina virtual; recordar el valor inicial de la llave de giro o el valor del contador del programa (número de controlador). En nuestro caso, las llamadas a la máquina virtual están ofuscadas (recuerde las funciones aplazadas en gráficos que se analizaron anteriormente), lo que hace que el contexto de inicialización sea bastante difícil de recuperar de forma estática. En esa situación, uno puede simplemente tomar lo mejor de los dos mundos y utilizar un compromiso entre el análisis estático y dinámico, a veces denominado ejecución concólica.

Básicamente, eso significa que depuras el objetivo y capturas (interrumpes) cada llamada a la VM; dentro de la devolución de llamada, cambias del análisis dinámico al estático y procedes a las siguientes acciones:

1. Volcar la memoria del objetivo.
2. Inicialice el contexto de análisis simbólico con el volcado de memoria. En realidad, se puede utilizar un tipo de carga diferida. Todos los accesos al contexto no inicializado se resolverán y almacenarán en caché mediante el volcado de memoria.
3. Calcule la ejecución simbólica de la VM.

Con la ejecución concólic, resulta muy fácil seguir el flujo de ejecución de la máquina virtual (es decir, una sucesión de controladores). El proceso de análisis y seguimiento de controladores está completamente automatizado.

A continuación se muestra un ejemplo de salida de la herramienta para un controlador. Se aprecia claramente el uso extensivo de la clave de giro (que consta de key_a, key_b y key_c); en esta situación, la clave se utiliza para ofuscar el acceso al contexto de la máquina virtual:

```
[+] controlador de desmontaje 2be en 42c2cdh
[+] analizando el controlador en 0x42c2cd
[+] considerando el código de 0x42c2cd a 0x42c3b3
[+] enlace del controlador en caché

dword ptr [dword ptr [esp+4*(dword ptr [4*key_b+436000h]^
    (dword ptr [4*tecla_c+436000h]*dword ptr [4*tecla_a+436000h]))+140h]] =>
    dword ptr [dword ptr [esp+4*(dword ptr [4*key_b+436004h]^
        (dword ptr [4*tecla_c+436004h]*dword ptr [4*tecla_a+436004h]))+140h]]

tecla_c => tecla_c+5
tecla_b => tecla_b+5
tecla_a => tecla_a+5

nHandler => (dword ptr [4*key_b+43600ch]^
    (dword ptr [4*tecla_c+43600ch]*dword ptr [4*tecla_a+43600ch]))+
    (((dword ptr [4*tecla_c+436010h]^(dword ptr [4*tecla_b+436010h])^
        dword ptr [4*clave_a+436010h]))*(byte ptr [dword ptr [esp+4*
```

334 Capítulo 5 ■ Ofuscación

```
(dword ptr [4*tecla_b+436008h]^(dword ptr [4*tecla_c+436008h]^
dword ptr [4*tecla_a+436008h]))+140h]]&0ffff)&0fffffff)
```

[+] enlace simbólico

```
dword ptr [esp+0a8h] => dword ptr [esp+0ach] tecla_c => 0d6h
tecla_b => 1f6h
tecla_a => 126ah
nHandler =>
((21eh*(flag4&0ffh))&0xffffffff)+0ac6h
```

[+] enlace resuelto

```
dword ptr [esp+0a8h] => 4f3de0b9h
tecla_c => 0d6h
tecla_b => 1f6h
tecla_a => 126ah n
Manejador => 0ac6h
```

Resolviendo el desafío

Lo que hemos diseñado hasta ahora es equivalente a una herramienta de seguimiento de niveles de una máquina virtual. Para obtener una herramienta orientada al desensamblado, sería necesario manejar instrucciones de ramificación (saltos o llamadas no condicionales). Podríamos crear una herramienta más compleja, una especie de compilador, basada en un desensamblador de bytecode y capaz de regenerar código nativo. Usando un ejemplo anterior, la herramienta procesaría la entrada:

palabra clave d [ESP+0a8h] => palabra clave d [ESP+0ach]

a una fuente tipo C:

vm_ctx.reg_2ah = vm_ctx.reg_2bh;

Para este ejemplo, nos basaremos únicamente en la función de rastreo. La estrategia es sencilla. Tenemos una buena idea del algoritmo implementado por la máquina virtual, por lo que utilizaremos un análisis diferencial/de caja negra para identificar la divergencia entre un algoritmo Mersenne Twister (MT) estándar y el de la máquina virtual. Cuando se identifique una divergencia, comprobaremos la salida del rastreo.

Tomemos de nuevo un ejemplo para ilustrar esto: la inicialización del estado del algoritmo MT. La inicialización se implementa mediante la función func2. Solo analizaremos sus entradas/salidas. El estado es una matriz de 624 dwors. Utilizando implementaciones estándar y la misma semilla utilizada por el programa para un nombre de seis caracteres (3961821h), obtenemos lo siguiente:

- Implementación estándar: estado[1] = 0x968bfff6d
- Implementación de la máquina virtual: estado[1] = 0x968e4c84

Buscamos estos valores en la traza:

[+] controlador de desmontaje 2c2 a las 41f056h
[+] analizando el controlador en 0x41f056
[+] considerando el código de 0x41f056 a 0x41f165

```
[+] enlace del controlador en caché

byte ptr [esp+0dh] => byte ptr [dword ptr [esp+4*(dword ptr
[4*tecla_b+43600ch]^dpalabra ptr [4*tecla_c+43600ch]^dpalabra ptr
[4*tecla_a+43600ch])]+140h]]&0ffh
ptr de palabra d [ptr de palabra d [esp+4*(ptr de palabra d [4*key_b+436000h]^ptr de palabra d
[4*tecla_c+436000h]^dpalabra ptr [4*tecla_a+436000h])+140h]] => dpalabra ptr [dpalabra

Primer paso:
[esp+4*(dword ptr [4*tecla_b+436004h]^dword ptr [4*tecla_c+436004h]^dword ptr
[4*tecla_a+436004h])+140h]]^dword ptr [dword ptr [esp+4*(dword ptr
[4*tecla_b+436008h]^dpalabra ptr [4*tecla_c+436008h]^dpalabra ptr
[4*tecla_a+436008h])+140h]]
tecla_c => tecla_c+6
tecla_a => tecla_a+6
tecla_b => tecla_b+6
nHandler => (dword ptr [4*key_b+436010h]^dword ptr [4*key_c+436010h]^dword ptr

[4*tecla_a+436010h])+(((tecla dword [4*tecla_c+436014h]^tecla dword
[4*tecla_b+436014h]^dpalabra ptr [4*tecla_a+436014h]))*(byte ptr [dpalabra ptr
[esp+4*(dword ptr [4*tecla_b+43600ch]^dword ptr [4*tecla_c+43600ch]^dword ptr
[4*tecla_a+43600ch])+140h]]&0ffh))&0xffffffff)

[+] enlace simbólico

byte ptr [esp+0dh] => bandera2&0ffh dword ptr
[esp+100h] => dword ptr [esp+9ch]^dword ptr [esp+10ch] clave_c => 10e2h clave_a => 12b3h clave_b =>
0c8ah nHandler =>

((164h*(bandera2&0ffh))&0xffffffff)+0a53h

[+] enlace resuelto

byte ptr [esp+0dh] => 1 dword ptr
[esp+100h] => 968e4c84h clave_c => 10e2h clave_a
=> 12b3h clave_b =>
0c8ah
```

Tenemos una operación XOR entre dword ptr [ESP+9ch] y dword ptr [ESP+10ch]. Podemos comprobar a partir del contexto sus valores:

```
[+] volcado de contexto
[...]
palabra clave [esp+9ch] => 968bff6dh
palabra clave [esp+10ch] => 5b3e9h
[...]
```

Este manejador tiene una semántica similar a XOR y está incluido dentro de uno de los bucles identificados anteriormente (uno con 624 iteraciones, el tamaño del estado MT). Hay algunos pasos más para recuperar la transformación completa, pero este enfoque es suficiente. Su pseudocódigo sería el siguiente:

```
confusión = 0x5b3e9h
para i en (N-1)
    estado[i+1] ^= desorden
```

desorden = lcg_rand(desorden)

donde N es el tamaño del estado: 624. lcg_rand es un generador congruencial lineal $x_{n+1} \equiv (ax_n + c) \pmod{m}$, con a, c y m respectivamente iguales a 0x159b, 0x13e8b, y 0xffffffff.

El resto del algoritmo Mersenne Twister también ha sido ligeramente modificado; cada uno de estos ajustes implica la primera letra del nombre de usuario y una operación simple ADD/SUB/XOR. No diremos más sobre estos ajustes; consulte la siguiente sección de "Ejercicios".

- Nombre de usuario: "¡Claro que sí, tenemos herramientas!"
- Número de serie : "117538a51905ddf6"

Reflexiones finales

Ese ejemplo es un gran campo de juego, muy bien diseñado por su autor. Hemos utilizado una combinación interesante de análisis dinámico y estático para trabajar en él. La protección implementa el aplanamiento de código, la virtualización de código y la codificación de datos, conceptos que se pueden encontrar en la mayoría de los sistemas de protección de nivel profesional, y aún así sigue siendo accesible. Proporciona una plantilla útil para afinar las herramientas y experimentar con nuevas ideas y/o algoritmos. La simplicidad del esquema de protección y del algoritmo nos permitió tomar muchos atajos para esta sección.

Ceremonias

El primer ejercicio que te proponemos es generar el binario del caso de estudio de este capítulo. Este es un excelente punto de partida:

- El sistema binario es único, relativamente pequeño y fácil de analizar, desensamblar e instrumentar, por lo que es un desafío accesible incluso para principiantes.
- La mayoría de las técnicas importantes implementadas se han descrito en el estudio de caso. Búscalas y asegúrese de comprender sus funciones internas.

Después de leer este capítulo, será una experiencia invaluable poder afrontar el desafío por tu cuenta. Tu tarea es la siguiente:

1. Basándose en la metodología propuesta (o una que se le ocurra), cree su propia herramienta para analizar el bytecode de la máquina virtual.
2. Póngase en contacto con su división de demostración favorita y empaquete un impresionante keygen para Este buen crackme.

Para familiarizarte con Metasm, encontrarás dos guiones de ejercicios con el material que viene con el libro: symbolized-execution-lvl1.rb y symbolized-execution-lvl2.rb. Responder las preguntas te llevará a un viaje en

Elementos internos de Metasm. Puede encontrar los scripts en www.wiley.com/go/practical-ingenieria-inversa.com.

Notas

1. Bansal, Sorav y Aiken, Alex. “Generación automática de superoptimizadores de mirillas”, 2006, <http://theory.stanford.edu/~sbansal/pubs/asplos06.pdf>.
2. Barak, Boaz et al., “Sobre la (im)posibilidad de ofuscar programas”. Informe técnico, Coloquio electrónico sobre complejidad computacional, 2001. <http://www.eccc.uni-trier.de/eccc>.
3. Beckman, Lennart et al., “Un evaluador parcial y su uso como herramienta de programación”, Inteligencia Artificial, Volumen 7, Número 4, págs. 319–357, 1976.
4. Bellard, Fabrice. “QEMU, un traductor dinámico rápido y portátil”. Documento presentado en las Actas de la Conferencia técnica anual de USENIX, FREENIX Track, 41–46, 2005.
5. Billet, Olivier, Gilbert, Henri y Ech-Chatbi, Charaf. “Criptoanálisis de una implementación de AES de caja blanca”. En *Selected Areas in Cryptography*, editado por Helena Handschuh y M. Anwar Hasan, 227–240. Springer, 2004.
6. Boyer, RS, Elspas, B. y Levitt, KN SELECT: un sistema formal para probar y depurar programas mediante ejecución simbólica. *SIGPLAN Not.*, 10:234–245, 1975.
7. Chipounov, V. y Candeal, G. “Enabling Sophisticated Analyses of x86 Binaries with RevGen.” Documento presentado en los talleres Dependable Systems and Networks (DSN-W), 2011 IEEE/IFIP 41st International Conference, 211–216, 2011.
8. Chipounov, V., Kuznetsov, V. y Candeal, G. “S2e: una plataforma para el análisis de rutas múltiples in vivo de sistemas de software”, ACM SIGARCH Computer Architecture News, vol. 39, núm. 1, 265–278, 2011.
9. Chow, S., Eisen, PA, Johnson, H. y van Oorschot, PC A White-Box DES Implementation for DRM Applications. En *Security and Privacy in Digital Rights Management*, ACM CCS-9 Workshop, DRM 2002, Washington, DC, EE. UU., 18 de noviembre de 2002, Documentos revisados, volumen 2696 de Lecture Notes in Computer Science, 1–15. Springer, 2002.
10. Chow, S., Eisen, PA, Johnson, H. y van Oorschot, Criptografía de caja blanca para PC y una implementación de AES. En *Selected Areas in Cryptography*, volumen 2595 de Lecture Notes in Computer Science, 250–270. Springer, 2002.

11. Chow, Stanley T., Johnson, Harold J. y Gu, Yuan. Resistente a la manipulación Codificación de software, 2003.
12. Collberg, Christian, Thomborson, Clark y Low, Douglas. Una taxonomía de las transformaciones de ofuscación. Informe técnico, 1997.
13. Collberg, Christian S., Thomborson, Clark D. y Low, Douglas. "Fabricación de estructuras opacas baratas, resistentes y discretas". En POPL, 184–196, 1998.
14. Cousot, P. y Cousot, R. "Interpretación abstracta: un modelo reticular unificado para el análisis estático de programas mediante la construcción o aproximación de puntos fijos". En Acta de la conferencia del 4º Simposio ACM sobre principios de lenguajes de programación (POPL '77), 238–252. ACM Press, Nueva York, 1977.
15. Cousot, P. y Cousot, R. "Diseño sistemático de marcos de transformación de programas mediante interpretación abstracta". En Acta de la conferencia del vigésimo noveno simposio anual ACM SIGPLAN-SIGACT sobre principios de lenguajes de programación, 178-190. Nueva York, 2002. ACM Press.
16. Cousot, P. "Diseño constructivo de una jerarquía de semánticas de una transición Sistema por interpretación abstracta. ENTCS, 6, 1997.
17. Dalla Preda, Mila. "Ofuscación de código y detección de malware mediante interpretación abstracta" (tesis doctoral), http://profs.sci.univr.it/~dallapre/MilaDallaPreda_PhD.pdf.
18. Dalla Preda, Mila. y Giacobazzi, Roberto. "Ofuscación del código de control mediante interpretación abstracta". En la Tercera Conferencia Internacional IEEE sobre Ingeniería de Software y Métodos Formales, 301–310, 2005.
19. Deroko. Nanomitas.w32. <http://deroko.phearless.org/nanomites.zip>.
20. Ernst, Michael D. Análisis estático y dinámico: sinergia y dualidad". En Actas de WODA 2003: Taller sobre análisis dinámico, Portland, Oregón, 24-27, mayo de 2003.
21. Futamura, Yoshihiko. "Evaluación parcial del proceso computacional: un enfoque hacia un compilador-compilador", 1999. <http://cs.au.dk/~hosc/local/> Documento sin título.
22. Gazet, Alexandre y Guillot, Yoann. "Cómo vencer la protección del software con Metasm". En HITB Malaysia, Kuala Lumpur, 2009. <http://metasm.cr0.org/docs/2009-guillot-gazet-hitb-desproteccion.pdf>.
23. Godefroid, P., Klarlund, N. y Sen, K. "DART: Sistema automatizado dirigido Pruebas aleatorias". En PLDI '05, junio de 2005.

24. Goubin, L., Masereel, JM y Quisquater, M. "Criptoanálisis de implementaciones de DES de caja blanca". Archivo de Cryptology ePrint, Informe 2007/035, 2007.
25. Jacob, Matthias et al. "El superdiversificador: individualización de mirillas para la protección del software". 2008. <http://research.microsoft.com/apps/pubs/default.aspx?id=77265>.
26. Jakubowski, Marius H. et al. "Transformaciones iteradas y métricas cuantitativas para la protección del software", 2009. <http://research.microsoft.com/aplicaciones/pubs/default.aspx?id=81560>.
27. Josse, S. "Desempaquetado seguro y avanzado mediante emulación de computadora". En Actas de la Conferencia AVAR 2006, Auckland, Nueva Zelanda, 3 al 5 de diciembre de 2006, 174–190.
28. Josse, S. "Detección de rootkits desde fuera de la matriz". Journal in Computer Virología, vol. 3, 113–123. Springer, 2007.
29. Josse, S. "Recompilación dinámica de malware". En Actas del IEEE de la 47.^a Conferencia HICSS, 2014.
30. Kinder, Johannes, Zuleger, Florian y Veith, Helmut. "Un marco de trabajo basado en interpretación abstracta para la reconstrucción del flujo de control a partir de binarios". 2009. http://pure.rhul.ac.uk/portal/files/17558147/archivo_vmcai09.pdf.
31. Lattner, C. y Adve, V. "LLVM: Un marco de compilación para el análisis y la transformación de programas de por vida". En Simposio internacional sobre generación y optimización de código, 75–86, 2004.
32. Maximus. "Revertir una máquina virtual simple". 2006. <http://tuts4you.com/download.php?view.210>.
33. Maximus. "Reconstrucción de máquinas virtuales". 2006. <http://tuts4you.com/descargar.php?view.1229>.
34. Rolles, Rolf. "Cómo encontrar errores en máquinas virtuales con un demostrador de teoremas, primera ronda". http://www.openrce.org/blog/view/1963/Encontrar_errores_en_máquinas_virtuales_con_un_demostrador_de_teoremas,_ronda_1.
35. Rolles, Rolf. "Cómo derrotar a HyperUnpackMe2 con un módulo de procesador IDA", 2006. http://www.openrce.org/articles/full_view/28.
36. Scheller, T. "Llvm-qemu, backend para QEMU usando componentes LLVM", Google Summer of Code 2007. <http://code.google.com/p/llvm-qemu/>.
37. Sen, K., Marinov, D. y Agha, G. "CUTE: un motor de pruebas unitarias Concolic para C". En ESEC/FSE '05, septiembre de 2005.

38. Smaragdakis, Yannis y Csallner, Christoph. "Combinación de razonamiento estático y dinámico para la detección de errores". En Actas de la Conferencia Internacional sobre Pruebas y Demostraciones (TAP), LNCS vol. 4454, 1–16, Springer, 2007.
39. Smithson, Matt et al. "Un sistema de análisis y reescritura binarios basado en representación intermedia a nivel de compilador ". En MALWARE, 47–54, 2010.
40. Song, Dawn et al. "BitBlaze: un nuevo enfoque para la seguridad informática a través del análisis binario". En Actas de la 4.^a Conferencia internacional sobre seguridad de los sistemas de información, Hyderabad, India, 2008.
41. Thakur, A. et al. "Generación de pruebas dirigidas para código de máquina". 2010. <http://research.cs.wisc.edu/wpis/papers/cav10-mcveto.pdf>.
42. Weiser, M., "Program Slices: Investigaciones formales, psicológicas y prácticas de un método automático de abstracción de programas" (tesis doctoral, Universidad de Michigan, Ann Arbor, 1979).
43. Zhao, J. et al, "Formalización de la representación intermedia LLVM para transformaciones de programas verificados". En Actas del 39.^º Simposio anual ACM SIGPLAN-SIGACT sobre principios de lenguajes de programación, 427–440, 2012.
44. Zhu, William y Thomborson, Clark. "Un esquema demostrable de ofuscación homomórfica en seguridad de software". En CNIS, 208–212. ACTA Press, 2005.

Nombres de muestra y Hashes SHA1 correspondientes

A continuación se muestran ejemplos de malware reales utilizados en los tutoriales y ejercicios del libro. Son malware activos y pueden causar daños a su computadora si no se manejan adecuadamente. Tenga cuidado al almacenarlos y analizarlos.

NOMBRE DE REFERENCIA	SHA1
Muestra A	092e149933584f3e81619454cbd2f404595b9f42
Muestra B	abeja8225c48b07f35774cb80e6ce2cdfa4cf7e5fb
Muestra C	d6e45e5b4bd2c963cf16b40e17cd7676d886a8a
Muestra D	2542ba0e808267f3c35372954ef552fd54859063
Muestra E	0e67827e591c77da08b6207f550e476c8c166c98
Muestra F	086b05814b9539a6a31622ea1c9f626ba323ef6d
Muestra G	531971827c3b8e7b0463170352e677d69f19e649
Muestra H	cb3b2403e1d777c250210d4ed4567cb527cab0f4
Muestra I	5991d8f4de7127fcf34840f1dbca2d4a8a6f6edf
Muestra J	70cb0b4b8e60dfed949a319a9375fac44168ccbb
Muestra K	23ff fc74cf7737a24a5150fab4768f0d59ca2a5c
Muestra L	7679d1aa1f957e4afab97bd0c24c6ae81e23597e

Índice

SÍMBOLOS \$

en comandos de ejecución de script, 242 \$\$
comando (comentarios), 226–227 \$<./\$>< comandos
(archivos de script), 240–241

* (asterisco) para crear comentarios, 227
comandos ??/?? (expresiones), 190–191 @@ prefijo
x (expresiones), 191 [] (corchetes)
para indicar acceso a memoria (x86), 5
{ } (llaves) en comandos de
bloque, 228 en
sentencias condicionales, 229 |
comando (depuración), 196 0x4038F0
rutina, 173–175 0xE* patrón, 71 ()
paréntesis en sentencias
condicionales, 229
registro EFLAGS de 32 bits, 3 registros
de propósito general
(GPR) de 32 bits, 2 conmutador /
3GB , 89 GPR de 64 bits, 3 registros de 64
bits, 36

Una ABI (Interfaz binaria de aplicación)
(ARM), 72
interpretación abstracta, 290, 294–295 semántica
abstracta, 290–291 Acorn RISC
Machine, 39 ADD instrucción, 14
Dirección parámetro
(memoria), 203–204 traducción de
direcciones,
26–27 ejecución ad-hoc. Véase
ejecución asíncrona y ad-hoc instrucción ADR ,
52 AL (siempre ejecutar)
condición, 70–71 alias
(DbgEng) @call alias de archivo de script, 244–
249 automático, 219,
225–226 nombre fijo, 225 con nombre de
usuario, 219–224 Manual del
programador de la
arquitectura AMD64, 28–34
operación AND , 75 APC (llamadas a
procedimientos
asíncronos)

NIVEL APC (1) IRQL, 105
conceptos básicos, 131–135

- Implementando la suspensión de hilo con, 134
modo usuario, 131–132
API (interfaces de programación de aplicaciones)
Accediendo a DbgEng, 258–261
Módulo de conexión de API (VxStripper), 309
- Control de dispositivos Io , 151
KeInitializeApc , 132–133
Obtener dirección física, 181
Hilo de creación del sistema Ps, 128
Funciones de la API de Win32, 32
Extensión WinDbg (SDK), 261–262
Interfaz binaria de aplicación (ABI), 15
registro de estado del programa de aplicación (APSR), 61
Aproximación de ordenación parcial, 290–291
contexto arbitrario (núcleos), 109–110
argumentos, pasar a archivos de script, 242–244
operaciones aritméticas
 sustitución aritmética mediante identidades
 (ofuscación), 275
BRAZO, 60–61
Conjunto de instrucciones x86, 11–13
Arquitectura ARM
 operaciones aritméticas, 60–61
 Manual de referencia de la arquitectura ARM:
 Edición ARMv7-A y ARMv7-R, 40
- Estado ARM, 41–42
Fundamentos, 40–42
ramificación y ejecución condicional, 61–66
- tipos de datos y registros, 43–44
Tutorial de descompilación de funciones
 desconocidas, 71–77
Funciones e invocación de funciones, 57–60
- incrementando valores en la memoria, 4
instrucciones, 46–47, 70–71
Código JIT (justo a tiempo), 67
- Cargar/almacenar datos. Ver carga/
 almacenamiento de datos (ARM)
visión general, 39–40
SMC (código automodificable), 67
primitivas de sincronización, 67–68
Servicios y mecanismos del sistema, 68–70
- controles y configuraciones a nivel de sistema, 45
Prueba de conocimientos sobre ARM, 77–78
Comando aS (alias), 220–222
Ejecución asincrónica y ad-hoc
 llamadas a procedimientos asincrónicos
 (APC), 131–134
 rutinas de finalización, 143–144
 Llamadas a procedimientos diferidos (DPC),
 135–139
 devoluciones de llamadas de procesos y subprocessos, 142
 subprocesos del sistema, 128–129
 temporizadores, 140–141
 elementos de trabajo, 129–131
 llamadas a procedimientos asincrónicos (APC),
 131–134
Sintaxis de AT&T para código ensamblador x86,
 4–5
alias automáticos (DbgEng), 219, 225–226
- B**
Instrucción B (Sucursal), 58
comando ba (puntos de interrupción de hardware),
 210
método backtrace_binding , 298
retroceso/corte (Metasm), 297–302
- Función de cambio de barril (ARM), 42–43, 60
puntero del marco base, 16
BeaEngine de BeatriX, 36
Tablero Beagle, 77–78
Instrucción BKPT , 70
Instrucción BL (Rama con enlace), 58–59
- Puntero parpadeante , 113–114

- bloques (scripting), 228–229
Instrucción BLX (Rama con enlace e intercambio), 59
- comando bp (puntos de interrupción de software), 210
Ramificación y ejecución condicional (ARM), 61–66
- Comando .break (scripting), 233
puntos de interrupción (depuración), 208–211
- Comando bu (puntos de interrupción no resueltos), 209
métodos de almacenamiento en búfer (controladores), 151–152
- Instrucción BX (Sucursal y Bolsa), 58
- codificación/cifrado de bytecode, 316
formatos bytes/word/dword/qword (registros), 202
- do
- C (Bandera de porte), 61–62
Sintaxis del evaluador de C++ (expresiones), 191
- Instrucción CALL , 15, 278, 281
@call alias del archivo de script, 244–249
devoluciones de llamadas, desensamblador (Metasm), 296
Convenciones de llamadas, 15–16
Puntero de instrucción de guardado similar a CALL/rama, 279
ventana de llamadas, 190
direcciones canónicas (x64), 37
Token de comando .catch (scripting), 232
- Instrucciones de comparación CBZ/CBNZ , 63
Caracteres y cadenas (scripting), 227–228
- listas circulares doblemente enlazadas, 112
Instrucciones de comparación CMN/TEQ , 64
- Instrucción CMOV (Intel), 42
Instrucción CMP , 18, 62–63
código
Enlace de código (Metasm), 302
código que recorre arrays, 8
- Ofuscación de código, 316–317
virtualización de código, 285–286
- Método de enlace de código (Metasm), 316–317
- aplanamiento de código (desofuscación), 316, 325
- ventana de comando/salida, 189–190
comentarios (\$\$ comando) (scripting), 226–227
- instrucciones de comparación, 62–63
integridad/solidez de los algoritmos de análisis, 290, 293–295
rutas de finalización, 143–144
equivalencia computacional, 268
ordenamiento parcial computacional, 290
ejecuciones concólicas, 292, 331
puntos de interrupción condicionales, 210–211
código condicional (cc), 17–18, 62
ejecución condicional (ARM)
conceptos básicos, 42
ramificación y, 61–66
declaraciones condicionales (scripting), 229–231
- Depurador de consola (CDB), 188–189
Ofuscación por plegado constante, 273
algoritmo de propagación constante, 291
optimización del compilador de despliegue constante, 271
optimización del compilador de despliegue constante, 273
solucionadores de restricciones (SMT/SAT), 292
Macro CONTAINING_RECORD , 118–119
Comando .continue (scripting), 233
continuo de análisis dinámico/estático, 291–293
- Flujo de control (conjunto de instrucciones x86), 17–25
control indirecto, 280–283
Ofuscaciones basadas en control
Fundamentos, 278–283
interacción con ofuscaciones basadas en datos, 284–288
- gráficos de flujo de control (CFG), 283, 285–286

- coprocesadores en ARM, 45
CPSR (registro de estado actual del programa), 41, 44
Función CreateToolhelp32Snapshot , 32
nivel de privilegio actual (CPL), 2
- D**
- Comando d (volcado de memoria), 205
Prevención de ejecución de datos (DEP), 182
movimiento de datos
 Arquitectura x64, 36–37
 Conjunto de instrucciones x86, 5–11
tipos de datos
 Arquitectura ARM, 43–44
 Arquitectura x64, 36
 arquitectura x86, 3
Ofuscaciones basadas en datos
 Fundamentos, 273–277
 interacción con ofuscaciones basadas en el control, 284–288
esquemas de codificación de datos (ofuscación), 273–274
DbgEng
 Comandos, 195
 Marco de extensión DbgEng (SDK), 257
 Interfaces del depurador, 258–261
 visión general, 188
 caché d (núcleo ARM), 67
 Eliminación de código muerto (optimización del compilador), 274–275
 Eliminación de la declaración muerta, 271
 Extensión de depuración (ARM), 40
 herramientas de depuración (Windows)
 Automatización con SDK. Consulte SDK para ampliar el depurador
 puntos de interrupción, 208–211
 Interfaces del depurador (DbgEng), 258–261
 Lenguaje de marcado del depurador (DML), 215
- ventanas del depurador, 189–190
Evaluación de expresiones, 190–194
extensiones/herramientas/recursos, 264–265
Inspección de procesos y módulos, 211–214
- Comandos relacionados con la memoria, 203–208
Comandos de depuración varios, 214–216
- visión general, 187–189
Control de procesos y subprocesos, 194–198
Gestión de registros, 198–203
Creación de scripts con. Consulte Creación de scripts con herramientas de depuración
 configuración de rutas de símbolos, 189
 símbolos, 208
 operadores útiles, 192–194
 extensiones de escritura, 262–264
aproximaciones decidibles del hormigón semántica, 290–291
- Llamadas a procedimientos diferidos (DPC), 135–139
- técnicas de desofuscación
 aplanamiento de código, 325
Continuum de análisis dinámico/estático, 291–293
- Aproximaciones decidibles de la semántica concreta, 290–291
- visión general, 289–290
basado en patrones, 312–313
Basado en análisis de programas, 313–315
- Conceptos de solidez/completitud, 293–295
- Usando Metasm, 317–325
utilizando VxStripper, 325–328
Implementaciones de VM (virtualización de código), 315–317
- herramientas de desofuscación
 AIF, 295–296
 Metasm, marco de código abierto. Ver Marco de código abierto Metasm
Marco de ingeniería inversa de Miasm, 302–304

- Resumen de la herramienta de reescritura binaria VxStripper 312. Véase la herramienta de reescritura binaria VxStripper Desclaux, Fabrice, 302 rutina de descifrado (scripts), 255–256 objetos de dispositivo (controladores), 149–150 API de DeviceControl , 151 Gestión de derechos digitales (DRM), 268 Método de almacenamiento en búfer de E/S directa, 151 devoluciones de llamadas del desensamblador (Metasm), 296 ventana de desensamblado, 190 NIVEL DE DESPACHO (2) IRQL, 105 declaración de cambio del despachador, 285 comando de selector de visualización (registros), 202–203 Instrucciones DIV/IDIV , 13 Tutorial de la rutina DIIMain (x85), 28–34 Instrucción DMB (ARM), 68 Cadena DosDevices , 156 bucles do-while (scripts), 235–236 DPC_WATCHDOG_VIOLACIÓN (0x133), 139 controladores, análisis del núcleo de controladores de la vida real, 184–185 conceptos básicos, 146–147 objetos de controlador y dispositivo, 149–150 Función DriverEntry , 155 Rutina DriverUnload , 149 puntos de entrada, 147–149 Manejo de IRP , 150 Tabla KeServiceDescriptor, 153–155 mecanismos para la comunicación entre el usuario y el núcleo, 150–152 secciones, 155 registros de control del sistema, 153 Rutina DriverUnload (rootkit x86), 159–160 Instrucción DSB (ARM), 68 Comando .dvfree (memoria), 238 Traductor binario dinámico (DBT), QUEMU, 305 IRP dinámicos, 145 criterios de segmentación dinámicos, 291 análisis dinámico/estático (desofuscación), 290–293 E e command (edición de memoria), 206–207 Eagle, Chris, 295 EFLAGS registro, banderas comunes en, 17 EIP registro, 2 .else/.elsif tokens de comando, 229 programas cifrados sobre datos cifrados, 274 Endianness bit (E), 44 EngExtCpp extensiones (SDK), 257 ENODE estructuras, 130 puntos de entrada (controladores), 147–149 errores, script (depuración), 231–232 ETHREAD estructura de datos del núcleo, 107–108 Objetos ETHREAD , 132 eventos monitoreo (depuración), 197–198 señalizado/no señalizado, 110 excepciones conceptos básicos, 95 vectores de excepción, 68 manejo de excepciones/interrupciones, 25, 27–28 monitoreo (depuración), 197–198 Nunca ejecutar (XN), 182 contexto de ejecución (Windows), 109–110 Expresiones, evaluación (depuración), 190–194 ExpWorkerThread, 131 extensiones (herramientas de depuración)

bucles `foreach` proporcionados por extensiones , 239–240
 recursos para, 264–265
 escritura (SDK), 262–264

F

mutexes rápidos, 111
 excepciones de fallas, 28, 95
 fallas, página, 95
 monitor de archivos, escritura (script de depuración), 253–
 255 archivos controladores de minifiltro del sistema de archivos, 147 pasar argumentos a archivos
 de script, 242–244 tokenizar
 desde, 238 alias de nombre fijo (DbgEng), 219, 225
 Flink pointer, 113–114 floating-point format (registros), 201 vaciado de caché, 67 bucles `for` (scripting), 233–
 234 rutinas forzadas en línea, 162 bucles `foreach` (scripting), 236–240 análisis de rootkit/módulo forense
 (VxStripper), 309
 omisiones de puntero de marco, 16
 asignaciones totalmente homomórficas (ofuscación), 274
 invocaciones de funciones
 Arquitectura ARM, 57–60
 Arquitectura x64, 37
 Conjunto de instrucciones x86, 13–17
 Funciones Función epílogo, 17 Punteros/desplazamientos de función, 92–93 Función prólogo, 16 Entrada/salida, 279
 Uso de scripts como, 244–249

G

registros de propósito general (GPR), 2–3
 construcciones `goto` , 23
 aplanamiento de grafos, flujo de control, 285–286

mutex protegidos, 111
 Guilfanov, Ilfak, 296
 Guillot, Yoann, 296

Análisis semántico de manejadores H

(estudio de caso de ofuscación), 330–333
 puntos de interrupción de hardware (DbgEng), 209 interrupciones de hardware, 28, 95 propiedad de homomorfismo, 274

I -cache (núcleo ARM), 67
 Herramienta IDA (desofuscación), 295–296
 intérprete idaocaml , 312
 identidades, sustitución aritmética mediante

(ofuscación), 275
 instrucciones IDIV/DIV , 13 registro
 IDT, 31–32 token de comando `.if` , 229 construcciones `if-else`, 18–20, 23 base de imagen del módulo especificado, obtención (script de depuración), 249 tabla de direcciones de importación (IAT), 310 instrucción IMUL , 12 función InitializeListHead , 112–113 funciones en línea, 279 inspección de procesos/módulos (depuración), 211–214 semántica de instrucciones (Metasm), 297–298

instrucciones (ARM)
 conceptos básicos, 46–47 ejecución condicional de, 70–71 para invocaciones de funciones, 58
 instrucciones (x86) operaciones aritméticas, 11–13 flujo de control, 17–25 movimiento de datos, 5–11 invocación de función, 13–17 descripción general, 3–4 pila operaciones, 13–17

- Sintaxis, 4–5
Instrucción INT 3 , 184
Software de arquitecturas Intel 64 e IA-32
 Manual del desarrollador, 2
Manual de desarrollo de software de Intel, 1
Sintaxis Intel para código ensamblador x86,
 4–5
Manual de referencia de Intel/AMD, 31
propiedad intelectual, protección, 267 interfaces,
depurador (SDK), 258–261 interrupciones,
conceptos
 básicos (llamadas al sistema), 95–
 98 excepciones y, 27–28
 tablas de descriptores de interrupciones
 (IDT), 95–96
 nivel de solicitud de interrupción (IRQL),
 104–106
Paquetes de solicitud de E/S (IRP), 144–146
Estructuras IO_STACK_LOCATION , 175–
 176
 Estructura IO_STATUS_BLOCK , 161
 Función IoAllocateWorkItem , 129
 Solicitud completa de Io, 143–144
IoCreateDevice (rootkit x86), 156–157
- Operaciones/códigos IOCTL ,
 151–152
Manipuladores de IRP, análisis (conductores),
 160–161
Manejo de IRP (conductores), 150
Instrucción ISB (ARM), 68
Bandera de IT (bits si-entonces), 61–62
Instrucción TI (si-entonces), 64
- Comando J j (DbgEng), 230
Extensión Jazelle (ARM), 40
Instrucciones del JCC , 17–18
Código JIT (justo a tiempo), 67
Instrucciones JMP , 22
Rama similar a JMP , 279
Josse, Sébastien, 304
tablas de salto, 21–22, 64–65
inserción de código basura, 271, 284
- K
Estructura del KAPC , 132
Estructura del KDPC , 135–137
Bloque de versión Kd, 121, 123
API KeInitializeApc , 132–133
Función del kernel KeInsertQueueDpc ,
 115
kernel
 Kernel Debugger (KD), 188–189 kernel
 drivers. Ver drivers, kernel kernel memory
 space, 88 Kernel Patch Protection
 feature, 153 kernel synchronization primitives
 (Windows), 110–111 kernel-mode APCs,
 131–132 kernel-mode
 driver framework (KMDF), 147
KeServiceDescriptorTable (drivers), 153–
 155
KeStackAttachProcess, 110
- KLDR_DATA_TABLE_ENTRY
structure,
 121
Estructuras KNODE , 130
Estructura del KPRCB , 137
Conductor KSECDD , 183
Estructura KSPIN_LOCK , 111
Estructura de datos del núcleo KTHREAD ,
 107–108
Estructura de KTHREAD , 134–135
Estructura de KTIMER , 140–141
- yo
Estructura LARGE_INTEGER , 172
Pseudoinstrucción LDMFD , 55
Instrucciones LDM/STM (ARM), 52–55
Pseudoinstrucciones LDR (ARM), 51–52
- Instrucción LDREX (ARM), 67–68
Instrucciones LDR/STR (ARM), 47–51
Instrucción LEA , 5, 9
instrucciones de desplazamiento a la izquierda/
derecha, 12 controladores de filtro
heredados, 146 controladores de software
heredados, 146 registro de enlace (LR) (ARM), 43

- Listas (núcleo de Windows),
 detalles de implementación, 112–119
 funciones de manipulación de listas en modo
 núcleo, guía paso a paso, 119–123
 descripción general, 111–112
- Cadena de compilación LLVM (Low Level Virtual Machine), 305, 307, 311, 325
 comando
- !m (depuración en modo kernel), 211
 carga/almacenamiento
 de datos (ARM)
 Instrucciones LDM/STM , 52–55
 Pseudoinstrucciones LDR , 51–52
 Instrucciones LDR/STR , 47–51
 descripción general, 47
 Instrucciones PUSH/POP , 56–57
- Privilegio del sistema local , 183
- Instrucción LODS , 11
 bucles
 básicos, 22–25
 desenrollado, 316
- METRO
- Macroensamblador (MASM), 190–193
- Matriz de funciones principales , 158
- malware, 267, 341
- Fabricación barata, resistente y
 Construcciones opacas sigilosas, 283
 máscaras, registro, 199–201
 notaciones
 de direcciones y rangos de memoria, 203–204
 volcado de contenidos de, 205–206
 edición de contenidos de, 206–207
 puntos de interrupción de memoria (DR0–DR3), 3 listas de descriptores de memoria (MDL) (Windows), 106–107
 ventana de volcado de memoria, 190
 diseño de memoria (Windows), 88–89
 comandos relacionados con la
 memoria (depuración), 203–
 208 métodos para especificar el acceso, 5–6
 comandos varios, 207–208
- métodos de movimiento entre registros
 y, 3–4 Metasm marco
 de código abierto backtracking/slicing, 298–
 302 enlace de código, 302 devoluciones
 de llamadas de
 desensamblador , 296 semántica de
 instrucciones, 297–298 descripción
 general, 296 uso,
 317–325, 331 Miasm
- marco de ingeniería inversa, 302–
 304 MmGetPhysicalAddress
- API, 181 bits de modo (M), 44 registros
 específicos del modelo
 (MSR). Ver MSR (registros específicos del
 modelo) (x86) módulos, inspección
 (depuración), 211–214
- Instrucción MOV/MOVS , 3, 60
 Instrucciones MOVS/MOVSW/MOVSD , 8–9
 Instrucciones MOVW/MOV/T , 60
 Instrucciones MRC (lectura)/MCR (escritura), 45
 MSR (registros específicos del modelo) (x86),
 3
- Instrucción MUL , 12
 Instrucción MUL (ARM), 61 mutex,
 111
- norite
- N (bandera negativa), 61–62
 característica namomitas (Armadillo), 283
 Ningún método de almacenamiento en búfer, 151–152
 Nunca ejecutar (NX), 182
 memorias de grupo no paginadas (Windows),
 106
 eventos no señalizados, 110
 módulo de normalización (VxStripper), 310–311
- Operador NOT , 275
 Depurador simbólico del NT (NTSD),
 188–189
- Macro NT_SUCCESS() , 157–158

- Oh
- estudio de caso de ofuscación
- Análisis de la semántica de los controladores, 330–333
- Primer análisis, 328–330
- visión general, 328
- Resolviendo el desafío, 334–336
- ejecución simbólica, 333–334
- técnicas de ofuscación
- Basado en el control, 278–283
- Basado en datos, 273–277
- técnicas de desofuscación. Ver
técnicas de desofuscación
- Ejemplo, 269–273
- visión general, 267–269
- Seguridad por oscuridad, 288–289
- flujo de control/flujo de datos simultáneos, 284–288
- Una taxonomía de las
transformaciones ofuscadoras, 283
- modo de compensación, 50–51
- Guión de OllyDbg , 312–313
- predicados opacos, 283
- códigos de operación, coprocesador, 45
- Indirección del control del sistema operativo, 282–283
- Aplicaciones que preservan operaciones
(estructuras algebraicas), 274
- operadores (expresiones), 192–194
- Complemento Optimice, 296
- Operador OR (|), 200
- funciones de esquema, 279
- salida de comandos, tokenización a partir de, 237–238
- análisis de programas de
sobreaproximación, 290
- PAG
- errores de página, 95
- memoria de grupo paginado (Windows), 106
- PandaBoard, 77–78
- Campo ParentNode , 130
- Técnicas de evaluación parcial
(optimización), 291
- NIVEL PASIVO (0) IRQL, 105
- Parche protector, 153, 183
- desofuscación basada en patrones, 312–313
- Ofuscación basada en patrones, 271, 275–277
- Direccionamiento relativo a PC, 51
- optimización del compilador de mirilla, 271, 277
- memoria de grupo (Windows), 106
- Operaciones POP/PUSH (pilas), 13
- modo de dirección postindexada, 50–51
- pseudo-registros predefinidos, 216–218
- .prefer_dml 1 comando, 215
- modo de dirección preindexada, 50–51
- Comando .printf (depurador), 214–216
- función printf , 23
- privilegios, definición de modos (ARM), 40–41
- Bloque de entorno de proceso (PEB), 109
- ¡ Comando de extensión de proceso, 212!
- procesos y subprocesos
- Fundamentos, 107–109
- devoluciones de llamadas, 142
- control (depuración), 194–198
- procesadores
- Estructuras de bloques de control del procesador
(Windows XP), 120–121
- región de control del procesador (PCR), 89
- inicialización del procesador (Windows), 89–91
- bloque de control de la región del procesador
(PRCB), 89–90
- Indirección del control basado en procesador
(ofuscación), 280–282
- contador de programa (PC) (ARM), 43–44
- Transformaciones de programas
(ofuscación), 268
- Desofuscación basada en análisis
de programas, 313–315
- modo protegido (x86), 1

PsCreateSystemThread API, 128
 pseudoinstrucciones, 51, 55
 pseudoregistros (scripting), 216–219
Operaciones
 PUSH/POP (pilas), 13
 instrucciones (ARM), 56–57
Instrucción PUSH-RET, 279, 282

Q

QEMU (Quick EMULATOR) Dinámico
 Traductor binario (DBT), 305–307

R

Comando r (registros), 198–199
 Marco de radar, 312
 Parámetro de rango (memoria), 204
 Registro RBP, 36
 Instrucciones RDMSR/WRMSR, 3
 Instrucción RDTSC, 3
 modos reales (x86), 1
 referencia (núcleo de Windows), 179
 registros
 Arquitectura ARM, 43–44
 administración (depuración), 198–203
 ventana, 190
 arquitectura x64, 36
 arquitectura x86, 2–4
 Ruta de registro (controladores), 149
 Prefijo REP, 9–10
 estructuras de repetición (scripts) bucles
 do-while, 235–236 bucles
 foreach, 236–240 bucles for, 233–234 descripción general, 232–233
 bucle while, 234–235
 Controlador de excepciones RESET, 68 recursos, depuración, 264–265
 Instrucción RET, 15, 278, 298
 direcciones de retorno, 55
 Inversión de una máquina virtual simple, 316
 instrucciones de desplazamiento a la derecha/izquierda, 12 niveles de anillo (x86), 1–2

Direccionamiento relativo a RIP (x64), 36–37, 51
 Rolles, Rolf, 312
 análisis
 de rootkits (VxStripper), 309 análisis de rootkits x64, 172–178
Tutorial de análisis de rootkits x86, 156–171

Criptografía

básica C, basada en 25 rondas, 288–289

S

Instrucciones SCAS, 10–11
 secuencias de comandos con herramientas de depuración alias. Véase alias (DbgEng) bloques, 228–229 caracteres y cadenas, 227–228 comentarios (\$\$ comando), 226–227 declaraciones condicionales, 229–231 ejemplos de secuencias de comandos de depuración, 249–256 pseudo-registros, 216–219 estructuras de repetición. Consulte estructuras de repetición (scripting) errores de script, 231–232 comandos de archivo de script, 240–244 uso de scripts como funciones, 244–249 Manual de referencia de comandos SCSI, 185 SDK para ampliar el depurador interfaces del depurador, 258–261 recursos de extensión, 264–265 descripción general, 257–258 API de extensión WinDbg, 261–262 escritura de extensiones, 262–264 secciones (controladores), 155 seguridad, logro por oscuridad, 288–289 manejador SEH, 282 semántica concreta, 290–291 instrucción, 297–298 equivalencia semántica, 268 listas enlazadas simples secuenciadas, 112 localidad secuencial, 279–280

- Descriptores de la tabla de servicios, 92–93
Instrucción SETNE , 42
Hashes SHA1 (malware), 341
Instrucción SIDT , 30 eventos
señalizados, 110 divisiones
con signo/sin signo, 13 datos múltiples
de instrucción única
 Conjunto de instrucciones (SIMD),
45 listas enlazadas
individualmente, 112 cortes/retrocesos
 (Metasm), 297–302
- SMC (código automodificable), 67
Instrucción SMULL (ARM), 61 puntos
de interrupción de software (DbgEng),
208–210
- interrupciones de software (ARM), 69
solidez/integridad de los algoritmos de análisis,
290, 293–295 código espagueti,
279
- Spasojevic, Branko, 296 bloques
de giro, 111
operaciones de pila (x86), 13–17
puntero de pila (SP) (ARM), 43
ofuscación basada en pila, 271 IRP
estáticos, 145
criterios de corte estático, 291
técnicas de análisis estático/dinámico, 290, 291–
293
- Convención de llamada STDCALL , 32
Pseudoinstrucción STMFD , 55
Pseudoinstrucciones STMIA/STMEA , 47
Instrucciones STOS , 10–11
reducción de fuerza (operaciones
 aritméticas), 12
Instrucción STREX (ARM), 67–68 cadenas
(scripts) y caracteres,
227–228 tokenización a partir
de, 236–237 escritura de script
básico de descifrador, 255–256 función
 strlen()
(C), 10 nombres de campos de
estructura, 74
Instrucción SUB , 14
sub_10460 rutina de hoja, 158, 166
- rutina sub_10550 , 166 funciones
sub_11553/sub_115DA , 121
Instrucción SVC , 102
Modo SVC , 70
Instrucción SWI/SVC , 69
bloques de casos de conmutación,
19–20 declaraciones de casos de conmutación
(ARM), 65–66 comandos sxe/sxd (depuración),
198 ejecuciones simbólicas, 292–294, 322,
333–334
- comandos
 de símbolos para inspección, 208 rutas
 de símbolos, configuración (depuración),
 189
- primitivas de sincronización
 Arquitectura ARM, núcleo 67–68,
 sintaxis 110–111
- Ensamblaje ARM,
evaluación de 46 expresiones (depurador), 190
notaciones para código de ensamblaje x86/
 Intel/AT&T, 4–5
- instrucción SYSENTER , 28, 100–101
- llamadas al sistema (Windows)
 conceptos
 básicos, 92–94 fallas/trampas/interrupciones
 descripción general,
 94–95
 interrupciones, 95–98 trampas, 98–104
 contexto del sistema (núcleos), 109–110
 coprocesador de control del sistema (CP15), 45
 registros de control del sistema (controladores), 153 servicios y mecanismos
 (ARM), 68–70
 subprocesos del sistema, 128–
 129 controles/configuraciones a nivel del sistema (ARM),
 45
 mecanismos a nivel de sistema (x85), 25–28
- yo
- Una taxonomía de la ofuscación
 Transformaciones, 283
- Instrucciones TBB/TBH (ARM), 65–66
- TCG (generador de códigos pequeños), 305–307

Estructura TEB (Thread Environment Block),
203 localidad
temporal, 279–280 prueba de

conocimiento ARM, 77–78 dinámica,
292 instrucción
TEST , 18 subprocessos
procesos
y. Véase procesos e hilos devoluciones de
llamadas
de subprocessos y procesos, 142 contexto de
subproceso (núcleos), 109–110 bloque de
entorno de subprocessos (TEB), 109 suspensión de
subprocessos con APC, 134 Thumb bit (T), 44
extensión Thumb (ARM),
40 estado Thumb, 41–42 instrucciones
específicas de Thumb-2
(ARM), 64–65 temporizadores, 110–111, 140–141

tokenización a partir de archivos,
238 a partir de
la salida de
comandos, 237–238 a partir de cadenas, 236–237

Semántica de trazas, 290
inversión de transformación. Véase
técnicas de desofuscación, bloques
de traducción (instrucciones TCG),
305
excepciones de trampa,
28 trampas (llamadas al sistema), 95, 98–104

U
U-Boot bootloader, 69 UDPRs
(user-defi ned pseudo-registers). Ver
usuarios UMULL instruction
(ARM), 61 análisis de programas de sub-
aproximación, 290 estructura
UNICODE_STRING ,
156 módulo de desempaquetado (VxStripper),
309–310

puntos de interrupción no resueltos (depuración),
209–210

división sin signo/con signo (DIV/IDIV),
13
Descompresor UPX, escritura (script de depuración),
250–253
usuarios
pseudo-registros definidos por el usuario
(UDPR), 218–219, 242, 244–249
rangos de direcciones de usuario/núcleo, 89
comunicación usuario-núcleo (controladores), 150–
152 APC
de modo usuario, 131–132 marco
de controlador de modo usuario
(UMDF), 147 alias
nombrados por el usuario (DbgEng), 219–224

V

V (bandera de desbordamiento), 61–62
Vellvm (LLVM verificado), 305 traducción
de direcciones virtuales, 25–27 propiedad de
caja negra virtual, 287–288
Reconstrucción de máquinas virtuales, 316
máquinas virtuales (VM), 286
Implementaciones de VM (virtualización
de código), 315–317
Herramienta de reescritura binaria VxStripper
Módulo de enganche de API, 309
arquitectura de, 308
conceptos básicos,
304–305 módulo forense/análisis de rootkit,
309
módulo de normalización, 310–311
Extensión DBT de QEMU, 306–307
desempaquetado del módulo, 309–310
uso, 325–328

W

caminando hacia atrás controla los flujos, 299–300
Marco de extensión WdbgExts
(SDK), 257–261
sitios web para descargar el intérprete
idaocaml , 312
Marco de código abierto Metasm, 296

- Marco de ingeniería inversa de Miasm, 302
- Complemento Optimice, 296
- Marco Radare, 312 sitios web para obtener más información
- BeaEngine de BeatriX, 36
- desofuscaciones de flujo de control, 284
- depuración, 264–265 ABI
- x64 en Windows, 37 Código de operación e instrucción x86 Referencia, 296
- bucles while (scripting), 234–235 contexto de ataque de caja blanca (WBAC), 268, 286–288
- Funciones de la API de Win32, 32
- API de extensión WinDbg (SDK), 261–262
- Interfaz gráfica de WinDbg , 188–189 ventanas, depurador, 189–190
- Modelo de controlador de Windows (WDM), 147
- Fundamentos de Windows
- contexto de ejecución, 109–110
- nivel de solicitud de interrupción (IRQL), 104–106
- primitivas de sincronización del núcleo, 110–111
- listas de descriptores de memoria (MDL), 106–107
- diseño de memoria, 88–89
- memoria de grupo, 106
- procesos e hilos, 107–109 inicialización del procesador, 89–91 llamadas del sistema. Ver llamadas del sistema (Windows)
- Núcleo de Windows
- ejecución asíncrona y ad-hoc.
- Ver ejecución asíncrona y ad-hoc
- Paquetes de solicitud de E/S (IRP), 144–146
- controladores del núcleo. Ver controladores, listas del núcleo. Ver listas (núcleo de Windows) descripción general, 87–88 referencias/consejos/ejercicios, 179–184 tutorial de rootkit x64, 172–178 tutorial de rootkit x86, 156–171
- Administrador de objetos de Windows, 156 elementos de trabajo, 129–131
- Arquitectura x64
- direcciones canónicas, 37
- movimiento de datos, 36–37
- invocación de funciones, 37
- conjuntos de registros y tipos de datos, 36
- recorrido por rootkit, 172–178 arquitectura x86
- Tutorial de la rutina DllMain , conjunto de instrucciones 28 a 34. Ver instrucciones (x86)
- Referencia de instrucciones y códigos de operación 296
- descripción general, 1–2 conjuntos de registros y tipos de datos, 2–3 tutorial de rootkit, 156–171 mecanismos a nivel de sistema, 25–28 rangos de instrucciones de longitud variable, 4
- Truco de intercambio XOR , 270
-
- Z (bandera cero), 61–62 –
- z, modificador de línea de comandos, 188–189

ACUERDO DE LICENCIA DE USUARIO FINAL DE WILEY

Vaya a www.wiley.com/go/eula para acceder al EULA del libro electrónico de Wiley.