Ofuscadores de Código Intermedio. Reporte Preliminar

Daniel Dolz
daniel_jose_dolz@yahoo.com.ar

Gerardo Parra
gparra@uncoma.edu.ar

Departamento de Ciencias de la Computación
Facultad de Economía y Administración
Universidad Nacional del Comahue
Buenos Aires 1400 - 8300 Neuquén - Argentina
Tel/Fax (54) (299) 4490312/313

Resumen

La plataforma .NET de Microsoft, así como los entornos de programación en JAVA, se basan en una filosofía de *just-in-time compilation* (compilación bajo demanda al momento de la ejecución). Los programas desarrollados de esta manera se ejecutan en un entorno o framework independiente de la plataforma, basado en objetos y, en algunos casos, permitiendo que interactúen componentes desarrollados en distintos lenguajes de programación.

La clave de esta flexibilidad radica en que, tanto en JAVA como en las plataformas .NET, la compilación resulta en un código intermedio, independiente de la plataforma (bytecode y MSIL respectivamente) que brinda la flexibilidad mencionada. Sin embargo, dicha flexibilidad tiene un costo. Hoy en día, y utilizando herramientas gratuitas que pueden descargarse desde Internet, es asombrosamente fácil aplicar tecnologías de descompilación e ingeniería inversa a las dos plataformas de desarrollo más populares: JAVA y .NET.

En el presente trabajo, se analiza el riesgo de pérdida de propiedad intelectual que apareja esta arquitectura y se describe un mecanismo de protección, la ofuscación. Se analiza una transformación de ofuscación del estado del arte y se propone una mejora sobre la misma, elevando el nivel de protección y dificultando la tarea de los posibles atacantes a la propiedad intelectual.

PALABRAS CLAVES: Ofuscadores. Código Intermedio. Transformaciones de Ofuscación. Seguridad por Oscuridad.

1 Introducción

La apropiación del código fuente de un software desarrollado por una organización en manos de personal no autorizado y con intenciones evidentemente ilegales, podría tener las siguientes consecuencias:

- Pérdida a manos de la competencia del dinero invertido en I+D[1]. La competencia puede, de forma desleal, lanzar al mercado el mismo producto, con un "lavado de cara" y aprovechando la inversión de la organización original.
- Una organización competidora podría descubrir fallas en el producto y utilizarlas en su beneficio.
- En el caso puntual de los algoritmos de encriptación modernos, cuya

[1] I+D: Investigación y Desarrollo

seguridad está basada en la existencia de una clave desconocida y no de un algoritmo en particular, el acceso al algoritmo por parte de manos malintencionadas podría servir para, previa modificación de los mismos, intentar ataques de fuerza bruta contra los datos cifrados.

- El acceso al código fuente de una aplicación facilita mucho, el "crackeo" de sistemas anti piratería, como ser la registración de software mediante *keys*, *expiration dates*, *hardlocks*, etc.

Lo anterior indica problemas económicos, pero algunos gobiernos como el de los EEUU identifican al problema como de seguridad nacional [5].

En este artículo analizamos a las técnicas de ofuscación como una rama de la seguridad informática que brindan un nivel de protección superior al de las alternativas existentes. Si bien es un área relativamente nueva, existe una adecuada base teórica y varias implementaciones comerciales. Es de notar que algunas empresas como Microsoft, incluyen un paquete *lite* de ofuscación junto a su popular paquete Visual Studio 2003, reconociendo de alguna manera que el riesgo de seguridad existe y que la ofuscación es una manera de proteger el código intermedio.

La estructura del trabajo es la siguiente. A continuación, se analiza el contexto dentro de la seguridad informática donde se encuadra este trabajo. Luego, en la sección 3, se expone a la ofuscación como una alternativa de protección a la propiedad intelectual. En la sección siguiente, se describe una de las técnicas de ofuscación disponibles más modernas. Por último, se propone una mejora sobre dicha técnica y se detallan las conclusiones y líneas de trabajos futuros.

2 Contexto

En la Seguridad Informática tradicional, los esfuerzos suelen centrarse en proteger a equipos "propios" (en el sentido de la administración de los mismos) contra ataques externos. Los ataques externos pueden clasificarse de varias maneras: desde la violación física de un equipo informático, hasta intrusiones por medios remotos como una LAN o la Internet, pasando por ataques como los DoS (Denial of Service) que, si bien no comprometen al equipo protegido, le impiden cumplir su función. Se habla, generalmente, de un ambiente *Malicious program/applet/script → benign hosts*, en el sentido de que se protege un equipo conocido, "benigno", de programas o scripts "malignos" destinados a comprometer de alguna manera el mismo [2].

Sin embargo, en la Seguridad Informática vista como un mecanismo para proteger la propiedad intelectual de ataques de ingeniera inversa, el paradigma es radicalmente distinto. En este caso se quiere proteger un software propio, "benigno", contra ataques de descompilación e ingeniera inversa, llevados a cabo en un equipo ajeno sobre el que no existen controles ni ninguna limitación que pueda aplicarse a priori. Se llamará a esto, ambiente *Malicious Host → Benign Program*.

Cuando se habla de ataques a la propiedad intelectual, estamos hablando de un programa desarrollado por nosotros, que es descompilado y sometido a ingeniería inversa en el equipo de computación perteneciente al ingeniero inverso[4]. El equipo, al ser del ingeniero inverso, es inaccesible a nosotros: no es posible restringir lo que el ingeniero puede hacer. No es posible impedir que cargue un software de descompilación, ni que abra nuestros archivos de código intermedio con un *debugger*. Nuestro programa, por otro lado, no intenta hacer ninguna tarea no autorizada en dicho equipo. El objetivo principal del host malicioso será obtener un código entendible, mantenible y modificable.

Si bien el problema de seguridad mencionado, es decir, la posibilidad de que personas u organizaciones no autorizadas recuperen código fuente mediante técnicas como la ingeniería inversa y la descompilación ha existido desde hace varias décadas, solo desde hace unos pocos años se ha convertido en un problema significativo.

Específicamente, el nivel de exposición del código de los entornos de programación más modernos y populares, como por ejemplo Java y el framework Microsoft .NET es mucho mayor que el de las antiguas plataformas de programación populares, como son C/C++, Delphi, Cobol, Pascal. Estos últimos lenguajes de programación

enumerados tienen la característica de ser compilados y linkeditados, lo que en la práctica significa que son convertidos directamente al lenguaje de la máquina en la que serán ejecutados.

Se podría decir que los programadores escriben “código fuente” mientras que las computadoras ejecutan “código máquina”. La conversión entre código fuente a código máquina suele ser, para los lenguajes mencionados, una conversión de un solo sentido. Este “código máquina”, si bien es visible y es accesible fácilmente, posee un formato tan tedioso y poco visible, y los esfuerzos de ingeniería inversa tan lentos y costosos que generalmente conviene programar la aplicación o el algoritmo nuevamente antes que obtener el código fuente por esa vía.

Por otra parte, Java y los lenguajes .NET (C#, Visual Basic, Jscript, etc.) utilizan un enfoque distinto. Desde el punto de vista del programador, estos lenguajes son muy poderosos, flexibles, extensibles y adaptables a prácticamente cualquier necesidad y cualquier entorno en existencia [3]. El objetivo detrás de su creación es el de proveer una ganancia notoria en la productividad del desarrollador y de las organizaciones que utilizan el software producto.

Sin embargo, la principal causa de su flexibilidad proviene del hecho que, en vez de ser convertidos a código máquina en el momento de la compilación, la compilación resulta en un código intermedio, que está a medio camino entre el código fuente que ve el programador y el código máquina que ejecuta la computadora destino. El problema es que, este código intermedio (llamado byteCode en caso de Java y MSIL en el caso del .NET) es altamente visible y es muy fácil, a partir del mismo, obtener casi sin modificaciones, el código inicial que solo debería ser patrimonio exclusivo del programador o de la organización que por derecho posee la propiedad intelectual del mismo.

Es muy importante aclarar, a los efectos de este trabajo, la vulnerabilidad del código intermedio. Tanto en bytecode como en MSIL, es posible recuperar el código fuente original al punto que solo se pierde, con respecto al código original, los comentarios y la indentación original que le dio el programador. Los identificadores (nombres de tipos, variables, clases) y las estructuras de control como repetitivas y disyuntivas se recuperan totalmente.

Actualmente hay disponibles herramientas en Internet, de descarga libre y gratuita, que permiten que no solo hackers y crackers experimentados se hagan de códigos que no deben, sino permiten que cualquiera con mínimos conocimientos pueda hacerlo [6,7].

Algunas de estas herramientas son:

Para Java:

- Jad (http://kpdus.tripod.com/jad.html)
- Mocha (http://www.brouhaha.com/~eric/computers/mocha.html).

Para .NET

- Anakrino (http://www.saurik.com/net/exemplar/)

3 La ofuscación como alternativa de protección de propiedad intelectual

En términos generales, se entiende por ofuscar un código fuente o un código intermedio, un proceso mediante el cual se transforma mediante la aplicación de algoritmos de reescritura, un código perfectamente legible y entendible por una persona en otro de funcionalidad equivalente en un ciento por ciento, pero, en términos ideales, totalmente ilegible e incomprensible para un lector humano.

En pocas palabras, algunas de las técnicas de ofuscación más comunes consisten en la inclusión de bucles irrelevantes, cálculos innecesarios, comprobaciones fuera de contexto, nombres de funciones y de variables que no tienen nada que ver con su cometido, funciones que no sirven para nada, interacciones inverosímiles entre variables y funciones, etc. Otras técnicas, sin embargo, son mucho más potentes, en el sentido de que requieren un conocimiento superior de las características del lenguaje, e incluso pueden estar diseñadas para burlar a herramientas de ingeniería inversa específicas.

A continuación, se definirá la noción de transformación de ofuscación [2].

Definición: transformación de ofuscación

Sea $P \xrightarrow{\tau} P'$ la transformación de un código fuente o intermedio P en un código fuente o intermedio P'

$P \xrightarrow{\tau} P'$ es una transformación de ofuscación si P y P' tiene el mismo comportamiento observable, entendido desde el punto de vista de lo que percibe el usuario.

Más precisamente, para que τ sea una transformación válida se debe verificar que:

Si P falla al terminar dada una entrada, P' podría o no terminar.

De otra forma, dada una entrada, P' debe terminar y producir el mismo resultado que P.

Idealmente, P' debería tener características que dificultan la compresión del código fuente.

4 Una transformación de Ofuscación: Overload Induction

De las herramientas reales del mercado de ofuscación analizadas, la más avanzada parece ser Dotfuscator .NET, de la empresa Preemptive (http://www.preemptive.com/) [3]. La técnica más promocionada de Dotfuscator se denomina Overload Induction y se basa en la posibilidad de sobrecargar operadores de los lenguajes basados en objetos en general y de los lenguajes de la plataforma .NET en particular. Cabe aclarar que dicha técnica está protegida por patentes, de manera que ninguna otra herramienta comercial podría utilizarla.

La técnica consiste en renombrar la mayor cantidad de métodos distintos de una clase con el mismo identificador. Esto no plantea problemas de ejecución mientras los parámetros de los métodos sean diferentes, dado que por la sobrecarga de operadores, métodos distintos pueden tener el mismo nombre y la diferenciación será por sus parámetros. De esta manera, se estará induciendo la sobrecarga de operadores (overload induction) y se dejará al ingeniero inverso sin información pragmática en el sentido de que no tiene manera de conocer que, los métodos que se llaman igual, originalmente se llamaban de distinta manera.

4.1 Propuesta de mejora: Extreme Overload Induction

La técnica descripta en el punto anterior no se aplica a métodos que originalmente tenían la misma cantidad y tipo de parámetros (no hay posibilidad de overloading).

Nuestra propuesta de mejora consiste en dejar, a una clase, totalmente uniforme en cuando al nombre de sus identificadores. En el caso de que existan métodos con los mismos parámetros, se propone el agregado de parámetros opacos para uniformizarlos.

Idealmente, todos los métodos de la clase, luego de la ofuscación, tendrán el mismo nombre, tal vez "a" de manera que, aplicado en toda una aplicación A, todas las llamadas a todos los métodos de todas las clases de la aplicación serán extremadamente parecidos entre sí.

Para el caso de conflicto de parámetros, se agregarán parámetros "dummies" o irrelevantes de manera que al momento de la ejecución efectivamente pueda realizarse la identificación del método correcto mediante la sobrecarga de operadores.

Consideramos que esta técnica es superior debido a que el ingeniero inverso se encontrará con el siguiente panorama:

- **Todos** los métodos de **todas** las clases se llaman igual. El identificador de dicho método podría ser simple (caso "a") o extremadamente complejo, como por ejemplo un identificador de 100 caracteres de largo con caracteres no imprimibles intercalados.
- La única manera de particularizar un método es a través de la cantidad y tipo de los parámetros. Sin embargo, al estar las llamadas a los métodos ofuscadas por alguna otra técnica, la complejidad total aumenta.
- La incorporación de parámetros dummys por el ofuscador agregará complejidad al programa. Si además,

mediante el uso de otras técnicas de ofuscación, dichos parámetros falsos son incorporados al cuerpo del subprograma, se agrega una variable más que el hacker tiene que tener en cuenta.

5 Conclusiones y Trabajos Futuros

Se ha descripto brevemente el riesgo de pérdida de propiedad intelectual que acarrean las plataformas más populares de programación y desarrollo de software como son Java y los lenguajes del .NET framework.

Postulamos que la ofuscación es un área de la seguridad informática que podría brindar un nivel de protección superior al resto de las alternativas de protección. Dentro de las técnicas de ofuscación conocidas, se ha descripto una de las más modernas y se ha propuesto una mejora sobre la misma. Esta mejora permite elevar el umbral de ofuscamiento, volviendo más segura la distribución de código intermedio y elevando considerablemente los esfuerzos que debería invertir un ingeniero inverso para tener una versión funcional y útil del código.

Como trabajos futuros, existe un campo sumamente interesante en el desarrollo y evaluación de nuevas técnicas de ofuscación, así también en la investigación sobre las existentes, de manera de obtener las transformaciones más potentes pero al menor costo (relación óptima). Otra línea de trabajo futuro consiste en cuantificar y analizar la aplicación de transformaciones de ofuscación de manera acumulativa y ver si el resultado de la aplicación sucesiva de transformaciones distintas mejora el resultado más allá de la suma individual de resultados o si existe una ganancia extra. También resulta sumamente interesante la relación con las técnicas de optimización de código intermedio, de manera de generar transformaciones que además de oscurecer el código, lo optimicen.

6 Referencias

[1] Paul Tyma. Encryption, hashing, and obfuscation, Special to ZDNet, Abril 2000

[2] Christian S Collberg, Clark Thompson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection, http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson2000a/index.html, February 2000

[3] Preemptive, Dotfuscator, Producing More Efficient, More Secure .NET Applications. Technical White Paper. http://www.preemptive.com, 2004.

[4] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. Software – Practice & Experience, 25(7):811-829, Julio 1995.

[5] Mr. Jeff Hughes, Dr. Martin R. Stytz, Ph.D. Advancing Software Security– The Software Protection Initiative, 2001

[6] Willy Alexánder Marroquín. Ofuscadores (De la protección relativa del código intermedio), http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art146.asp

[7] Gary McGraw, John Viega, Make your software behave: Security by obscurity. Think your encrypted code is safe? Think again. http://www-106.ibm.com/developerworks/java/library/s-obs.html, Octubre 2000.