Vulnerabilidades de bajo nivel y software malicioso

PID_00255335

Sergio Castillo-Pérez

Universitat Oberta de Catalunya

Índice

Introducción 5

Objetivos 7

1. Vulnerabilidades de bajo nivel 9
1.1. Conceptos previos 9
1.1.1. Organización de la memoria de los procesos 10
1.1.2. Llamadas a funciones 11
1.1.3. *Buffers* 13
1.2. *Buffers overflow* 15

2. Software malicioso 19
2.1. Taxonomía del *malware* 20
2.1.1. *Malware* de propagación automática 20
2.1.2. *Malware* oculto 21
2.1.3. *Malware* lucrativo 22
2.2. Vectores de infección 24
2.3. Mecanismos de prevención 25

3. Detección de *malware* 27
3.1. Detección sintáctica basada en firmas 28
3.1.1. Tipos de firmas 29
3.1.2. Ámbitos de búsqueda 31
3.2. Detección semántica 33
3.2.1. Análisis dinámico 35
3.2.2. Análisis estático 37

4. Mecanismos de evasión 40
4.1. Técnicas de ofuscación 40
4.1.1. *Malware* cifrado, oligomorfismo, polimorfismo y metamorfismo 41
4.1.2. Compresión de ejecutables 44
4.1.3. *Entry point obscuring* 45
4.1.4. Ofuscación por virtualización 45
4.2. Técnicas de ocultación y autoprotección 46
4.3. Mecanismos *antidebugging* 48

Resumen 49

Actividades 51

Ejercicios de autoevaluación 51

Solucionario 52

Bibliografía 53

Introducción

En la última década, la seguridad computacional se ha convertido en una necesidad omnipresente en todos los sistemas de información. La cantidad de vulnerabilidades que aparecen cada día no deja inmune a ningún sistema a no ser potencialmente comprometido.

En este contexto, existe un tipo de vulnerabilidad muy particular que se caracteriza por las graves consecuencias que puede suponer si un sistema es vulnerable. Estamos hablando de los llamados desbordamientos de *buffer*, o *buffers overflow*. Este tipo de error no es nada nuevo, y su mayor repercusión fue causada por primera vez ya en el año 1988 por el gusano de Morris.

Hoy en día, pese a los avances que se han realizado en el campo de la seguridad computacional y las tecnologías que incorporan los sistemas para luchar contra este tipo de vulnerabilidades, aún se sigue descubriendo software con este tipo de deficiencia. El peligro subyacente a este tipo de vulnerabilidad radica en el hecho de que su explotación permite, entre otras cosas, la ejecución de cualquier código arbitrario. En concreto, posibilita la inyección de cualquier código a la aplicación vulnerable, de manera que dicho código es ejecutado con los mismos privilegios que posee el programa atacado. No es de extrañar que por este motivo el software malicioso aproveche este tipo de deficiencia para tomar el control sobre aplicaciones o provocar su terminación.

Sin duda alguna, existe una estrecha relación entre las vulnerabilidades de seguridad y el software malicioso. A nuevas vulnerabilidades, nuevas vías de explotación para el software malintencionado. Más, si tenemos en cuenta que la proliferación de éste en los últimos tiempos también ha sido notoria. En este contexto, también ha habido un cambio de paradigma radical. Mientras que en los inicios la motivación por crear código malicioso era el de la fama y el reconocimiento, hoy en día la industria del *malware* se profesionaliza y sus creadores se focalizan en obtener beneficios económicos. Este hecho hace que los programadores de código malicioso estén más motivados, y como consecuencia, que la evolución de las técnicas y la invención de nuevas estrategias incorporadas en el software malicioso sean más fructíferas. Un claro ejemplo ha sido el gusano *Stuxnet*, considerado hoy en día como una de las amenazas más complejas de los últimos tiempos debido a su nivel de sofisticación. Sin duda, nuevas brechas se han abierto en los últimos tiempos que serán explotadas por el futuro código malintencionado, como son los dispositivos móviles, las redes sociales, los sistemas SCADA, o las máquinas virtuales.

A lo largo de este módulo vamos a estudiar las bases teóricas de los desbordamientos de *buffer*, al mismo tiempo que se expondrán los conceptos principales asociados al código malicioso. Analizaremos los tipos de *malware* existente, cómo es posible detectar su presencia, y qué estrategias usan éstos para evadir su detección.

Objetivos

Los objetivos que el estudiante debe haber conseguido después de estudiar los contenidos de este módulo son los siguientes:

1. Aprender las bases teóricas de las vulnerabilidades de tipo desbordamiento de *buffer*.
2. Identificar los diferentes tipos de *malware* en función de sus características.
3. Conocer los mecanismos básicos de prevención contra el software malicioso.
4. Tener una visión completa de las estrategias de detección de código malicioso.
5. Saber cuáles son los mecanismos de evasión utilizados por el *malware* para evitar su detección y erradicación.

1. Vulnerabilidades de bajo nivel

El día 2 de noviembre de 1988 una nueva amenaza apareció con el gusano de Morris. Éste, aprovechándose de la vulnerabilidad de diversos servicios extremadamente usados como eran *fingerd*, *sendmail* y *rsh*, causó grandes daños en Internet. Se estima que el gusano llegó a afectar a un 10 % de todas las máquinas conectadas a Internet por aquel entonces.

Al parecer, y de acuerdo con su creador Robert Tappan Morris, la intencionalidad del gusano no era maliciosa, sin embargo, un error de programación fue el responsable de su comportamiento que causaba una denegación de servicio en las máquinas afectadas. El error residía en el código responsable de la replicación, el cual provocaba que una misma máquina podía albergar más de una copia del gusano, con la consiguiente sobrecarga y ralentización del sistema por la existencia de múltiples procesos creados por el mismo gusano. Sin duda alguna, sus efectos fueron catastróficos para la época.

La propagación del gusano de Morris fue posible gracias a un tipo de fallo de seguridad denominado *buffer overflow* (desbordamiento de *buffer*). Este tipo de vulnerabilidad le permitía al gusano la ejecución de código en otra máquina remota y vulnerable a dicho fallo, creando así una nueva copia en el otro sistema. Desde entonces, este tipo de vulnerabilidad se ha convertido en un clásico al incrementarse su popularidad y al aparecer nuevas técnicas de explotación. A pesar de que cada día se han incorporado nuevos mecanismos eficientes de lucha contra los desbordamientos de *buffer*, como por ejemplo el *address space layout randomization* o el uso del *stack smashing protection*, es preciso conocer su funcionamiento dado que aun siguen estando vigentes en la actualidad.

En este apartado hablaremos precisamente de este tipo de vulnerabilidades, que hemos englobado bajo el concepto genérico de *vulnerabilidades de bajo nivel*. El motivo de esto reside en el hecho de que este tipo de fallos se caracterizan por producirse desde una perspectiva más cercana al funcionamiento de la máquina a bajo nivel.

1.1. Conceptos previos

Antes de abordar cuáles son las técnicas usadas por los desbordamientos de *buffer*, es necesario comprender diversos aspectos de bajo nivel que desarrollaremos a continuación. Dado que los mecanismos de los desbordamientos

de *buffer* son comprendidos mejor desde una perspectiva de bajo nivel, nos centraremos en una plataforma en concreto. En particular, las explicaciones aquí expuestas estarán basadas en la arquitectura Intel x86, sin embargo, estos mismos conceptos pueden ser extrapolados a otras arquitecturas con las particularidades que las caractericen.

Los aspectos que veremos a continuación se focalizarán en la organización de la memoria de los procesos, cómo se realizan las llamadas a las funciones, y cómo se almacenan los *buffers* en memoria.

Observación

A pesar de que las nociones aquí expuestas intentarán ser autocontenidas, es recomendable que estéis familiarizados con aspectos básicos de arquitectura de computadores y de lenguaje ensamblador. En Internet se pueden encontrar varios manuales, y os recomendamos que examinéis alguno en caso de necesidad.

1.1.1. Organización de la memoria de los procesos

Cuando un programa se ejecuta, el espacio lineal de memoria de su proceso asociado es particionado, desde un punto de vista lógico, en diversos intervalos de direcciones que llamamos **segmentos**. Cada una de estas particiones lógicas de espacios de memoria son utilizadas para diferentes fines.

Independientemente de la arquitectura y del sistema operativo de una máquina, en términos generales podemos considerar que existen cinco segmentos (ver ejemplo particular en la figura 1). En particular, éstos son:

1) *Text segment*: contiene el código del proceso. Es conocido también con el nombre de *code segment*.

2) *Data segment*: contiene los datos inicializados, esto es, las variables estáticas y globales cuyos valores iniciales están almacenadas en el archivo ejecutable, ya que el programa debe conocer el valor antes de iniciar su ejecución.

3) *BSS segment*: contiene los datos no inicializados, esto es, todas las variables globales cuyos valores iniciales no han sido guardados en el ejecutable, ya que el programa establecerá sus valores antes de referenciarlos.

4) *Heap segment*: zona donde apuntan las variables que reservan memoria de forma dinámica, y que viene gestionada con llamadas como `malloc`, `calloc`, o `free` entre otras. Asimismo, este segmento es utilizado para albergar las librerías dinámicas utilizadas por el proceso. Este segmento empieza al final del *BBS segment* y su tamaño es dinámico. En particular crece de direcciones bajas a direcciones altas.

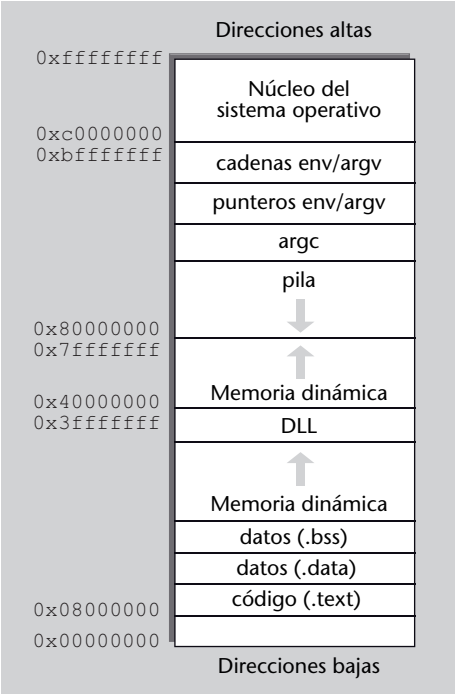
Operaciones con la pila

Recordad que en una pila utilizamos dos instrucciones, una para apilar datos y otra para desapilar. Generalmente se utilizan los nemotécnicos `PUSH` y `POP` respectivamente.

5) *Stack segment*: contiene la pila (*stack*) del programa, una estructura de tipo LIFO, que es utilizada principalmente para almacenar las direcciones de retorno cuando se hacen llamadas a funciones, a los parámetros que se les pasan a éstas, y a sus variables locales. Este segmento también es de tama-

ño dinámico, pero a diferencia del *heap segment*, crece de direcciones altas a direcciones bajas.

Figura 1. Organización de la memoria de un proceso en un sistema GNU/Linux



1.1.2. Llamadas a funciones

Una vez entendida la organización de la memoria de un proceso, es importante considerar cómo una llamada a una función es realizada desde el punto de vista de la memoria. De forma más concreta, centrémonos en el uso que se hace del segmento de pila desde una perspectiva de bajo nivel. Para esto, podemos considerar que toda llamada a una función se puede dividir en tres pasos que se ejecutan cronológicamente, y donde en cada uno de estos se realiza un conjunto de acciones necesarias para el correcto funcionamiento del proceso:

- **Llamada**: los parámetros de la función son almacenados en la pila así como el contenido del registro *instruction pointer**, lo que permitirá determinar la dirección de retorno tras la llamada. Seguidamente se ajusta el *instruction pointer* con la dirección de memoria de la función que se desea llamar, redirigiendo así el flujo de ejecución hacia ésta.

* `EIP` **en la arquitectura x86**

- **Prólogo**: el registro *frame pointer*** que contiene la dirección base de las variables locales, es almacenado en la pila y modificado para que apunte a las variables locales de la función que se está llamando. Se reserva el espacio necesario en la pila para las variables locales de la función llamada,

** `EBP` **en la arquitectura x86**

ajustando el registro *stack pointer**** que apunta a la cima de la pila. Tras el prólogo se ejecutará el código de la función en sí.

*** `ESP` en arquitectura x86

- **Retorno (o epílogo)**: se retorna al estado previo de la pila antes de la ejecución, restaurando el *frame pointer* y el *intruction pointer*. La restauración del *instruction pointer* permite redirigir el flujo de ejecución al código que realizó la llamada.

Ejemplo

Para una mayor comprensión, partamos de un ejemplo práctico y real de un programa escrito en lenguaje C, y el que analizaremos a bajo nivel tras compilarlo. El programa en cuestión es el siguiente:

```
void function(int a, int b, int c) {
    char buffer1[7];
    char buffer2[9];
}

int main(void) {
    function(1,2,3);

    return 0;
}
```

Como se puede observar, el programa está compuesto por dos funciones simples: `main` y `function`. La función principal `main` tan sólo realiza una llamada a la función `function`, y esta última se limita a declarar dos variables locales y a no hacer nada más.

Tras generar el archivo binario correspondiente mediante el compilador *gcc*, si lo cargamos con el *debugger gdb* y desensamblamos la función `main` obtendremos la siguiente salida:

```
0x080483bc <+0>:     push   ebp
0x080483bd <+1>:     mov    ebp,esp
0x080483bf <+3>:     sub    esp,0xc
0x080483c2 <+6>:     mov    DWORD PTR [esp+0x8],0x3
0x080483ca <+14>:    mov    DWORD PTR [esp+0x4],0x2
0x080483d2 <+22>:    mov    DWORD PTR [esp],0x1
0x080483d9 <+29>:    call   0x80483b4 <function>
0x080483de <+34>:    mov    eax,0x0
0x080483e3 <+39>:    leave
0x080483e4 <+40>:    ret
```

Fijémonos que las tres primeras instrucciones se corresponden al prólogo de la función `main`. En concreto, se guarda el *frame pointer* en la pila (instrucción `push ebp`), es ajustado para que apunte a las variables locales (instrucción `mov ebp,esp`), y se reserva espacio en la pila modificando el registro `ESP` al restarle el valor `0xc` (instrucción `sub esp,0xc`).

Las siguientes cuatro instrucciones forman parte de la fase de llamada a la función `function`. De forma específica, se guardan en la pila los parámetros que se le pasan a la función (valores `0x3`, `0x2` y `0x1` que aparecen en las instrucciones `mov`). A continuación se guarda en la pila el registro `EIP`, y se modifica su contenido con la dirección de la función `function` para redirigir el flujo de ejecución. Estas dos últimas acciones son realizadas de forma implícita por la sentencia `call 0x80483b4 <function>`.

Las tres últimas instrucciones se corresponden con el epílogo de la función `main`. Notad que la instrucción `leave` equivale a:

```
mov    esp,ebp
pop    ebp
```

Por último, la instrucción `ret` restaura el valor del *instrunction pointer*, lo que permite devolver a la pila el estado previo a la ejecución de `main`, y proseguir con el flujo de ejecución antes de la llamada a `main`.

Observación

Respecto al ejemplo es necesario puntualizar que, tal y como se especifica, éste funciona de la manera descrita si la máquina que se usa es una máquina con una arquitectura de 32 bits. Sin embargo, si la máquina que se utiliza es de 64 bits, el ejemplo no implica ningún desbordamiento de pila. Esto es así puesto que en una arquitectura de 64 bits la pila se alinea a 16 bytes (es decir, cada variable como mínimo ocupa 16 bytes). Si trabajáis con una arquitectura de 64 bits, podéis compilar el programa `strbof.c` usando el siguiente comando:
`student@uoc ~$ gcc -m32 strbof.c -o strbof`
De esta forma se obliga al compilador a generar código para una arquitectura de 32 bits y el ejemplo del desbordamiento de pila funciona tal y como está descrito en los materiales.

Veamos a continuación el desensamblado de la función `function`:

```
0x080483b4 <+0>:     push   ebp
0x080483b5 <+1>:     mov    ebp,esp
0x080483b7 <+3>:     sub    esp,0x10
0x080483ba <+6>:     leave
0x080483bb <+7>:     ret
```

De nuevo, las tres instrucciones primeras se corresponden con el prólogo de función `function`. Se guarda el *frame pointer* en la pila y es ajustado para que apunte a las variables locales. A continuación, se reserva espacio en la pila para las variables locales restando al registro `ESP` el valor `0x10`, lo que permite albergar los *buffers* `buffer1` y `buffer2`.

Justo después de estas tres últimas instrucciones vendría el código propio de la función, sin embargo, dado que ésta no realiza ningún tipo de acción, seguidamente nos encontramos el epílogo. Por tanto, el epílogo está constituido por las siguientes dos instrucciones, las cuales restauran `ESP` y `EBP` (instrucción `leave`), así como el registro `EIP` (a través de la instrucción `ret`).

1.1.3. *Buffers*

Un *buffer* podemos definirlo como una cantidad de memoria reservada, limitada y continua.

Los *buffers* tienen una importancia relevante en el tipo de vulnerabilidad que nos ocupa, como veremos detenidamente más adelante. De forma más precisa, uno de los motivos principales por el que son posibles los desbordamientos de *buffer*, es la inexistencia de mecanismos que permitan detectar cuándo se ha excedido el límite del tamaño de un *buffer*. Esta circunstancia la encontramos en determinados lenguajes de programación como son el C, el C++, o derivados. Por tanto, es de vital importancia que el programador extreme las medidas oportunas, cuando utiliza uno de los citados lenguajes, para que no se produzcan estas deficiencias en el código.

Es importante remarcar que no todos los lenguajes sufren este problema, y como consecuencia no posibilitan programar código vulnerable a desbordamientos de *buffer*.

En el lenguaje C, las cadenas de texto son representadas como un apuntador al primer byte de un *buffer*, y se considera que se ha llegado al final de estas al encontrar el carácter nulo. Este carácter nulo viene representado por el valor en hexadecimal `0x00`. Esto conlleva que, *a priori*, no se sepa la longitud del las cadenas si no se recorren hasta detectar el carácter nulo. Esta forma de representación puede comportar consecuencias negativas desde el punto de vista de la seguridad, especialmente al ser usada con funciones denominadas como peligrosas.

Ejemplo

Para una mayor comprensión de esta problemática analicemos un ejemplo de un programa escrito en lenguaje C:

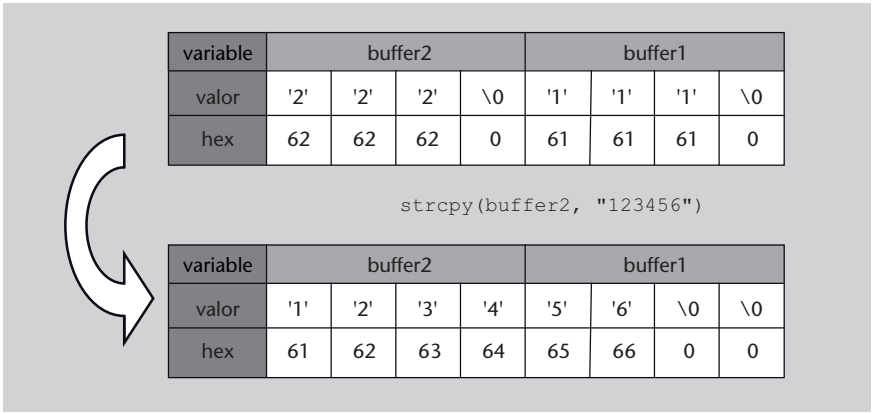
```
#include <stdio.h>
#include <string.h>

int main(void) {
    char buffer1[4] = "111";
    char buffer2[4] = "222";

    strcpy(buffer2, "123456");
    printf("%s\n", buffer1);

    return 0;
}
```

Si observamos el código entonces veremos que el programa define dos *buffers* denominados `buffer1` y `buffer2`. Ambos *buffers* tienen un tamaño de 4 bytes, cuyos contenidos son `111` y `222` respectivamente. Notad que en la declaración de las variables, el tamaño asignado a los *buffers* es de 4 bytes debido a que, junto a sus contenidos de longitud 3, se requiere un byte adicional para el carácter nulo de final de cadena. A continuación, el programa copia –mediante la función `strcpy`– la cadena de texto `123456` a `buffer2`, y seguidamente muestra el contenido de `buffer1`. Intuitivamente podríamos cometer el error de pensar que el resultado de la ejecución de este programa sería visualizar el valor `111`, sin embargo, la realidad es otra. Veamos qué ocurre:

```
student@uoc ~ $ gcc strbof.c –o strbof
student@uoc ~ $ ./strbof
56
student@uoc ~ $
```

¿Cómo se justifica este resultado? Basta con analizar la evolución de la memoria para entenderlo. En primer lugar, recordemos que las variables locales de una función se guardan en el segmento de pila. De ahí que tanto `buffer1` como `buffer2` –variables locales de la función `main`– estén albergadas en el segmento de pila y, además, la reserva de memoria para ambas variables se haga de forma contigua. De manera gráfica podríamos representar las dos variables locales tal como se muestra en la parte superior de la figura 2.

Figura 2. Desbordamiento de *buffer* con `strcpy`



Cuando se ejecuta la función `strcpy`, esta no tiene en cuenta el tamaño del *buffer* de destino `buffer2`, y copia caracteres hasta localizar el identificador nulo de la cadena '`123456`'. Como consecuencia, y dado que `buffer1` y `buffer2` son contiguos, el contenido de `buffer1` ha sido sobreescrito, llegando a copiar los bytes '`56`' junto al carácter nulo de final de cadena en el espacio reservado para `buffer1` (figura 2).

Esta problemática no es exclusiva de la función `strcpy`, sino que hay toda una lista de funciones que son susceptibles del mismo problema. Entre estas encontramos `gets`, `strcat`, `sprintf`, o `vsprintf` entre otras.

1.2. *Buffers overflow*

Una vez que se han entendido los conceptos básicos que hemos introducido en el subapartado anterior, estamos en condiciones de abordar las vulnerabilidades denominadas *buffers overflow*, o simplemente, desbordamiento de *buffers*.

Lectura recomendada

Sin duda alguna, el artículo más referenciado sobre desbordamiento de *buffers* fue el escrito por Elias Levy (conocido con el seudónimo Aleph One) en 1996, y que fue publicado en la revista *Phrack magazine* con el título "Smashing the Stack for Fun and Profit". Podéis encontrar este artículo en la siguiente dirección de Internet: http://www.phrack.org /archives/issues/49/14.txt.

Si analizamos las diferentes variantes de vulnerabilidades que aparecen bajo el epígrafe de *buffers overflow*, nos daremos cuenta de que hay una gran multitud de clases. Así, una pequeña lista de variantes que se engloban dentro de este tipo de vulnerabilidad podría ser la siguiente:

- Stack overflow
- Heap overflow
- String format
- Integer overflow
- Return into libc
- Return into environment
- Return into got

Todas las variantes mencionadas anteriormente se basan en explotar el mismo principio: desbordar un *buffer* al usar un determinado conjunto de funciones que no incluyen ningún mecanismo de control sobre las longitudes.

Lecturas complementarias

No se estudiarán en profundidad las diferentes variantes de vulnerabilidades que aparecen bajo el epígrafe de *buffers overflow*, pero sí se persigue proporcionar las bases necesarias para que podáis acudir a otras fuentes –tales como Foster y otros (2005) o Koziol y otros (2004)– y comprendáis su funcionamiento. Asimismo, estas variantes serán analizadas con detenimiento en otras asignaturas, como por ejemplo *Programación segura*.

En nuestro caso, nos centraremos en la variante *stack overflow*, que nos servirá para comprender cuál es la problemática y cómo es posible que este tipo de vulnerabilidades consiga ejecutar código arbitrario. Para ilustrar esta variante lo ejemplificaremos a partir de un programa simple y analizaremos los aspectos importantes para su comprensión. El código correspondiente al programa es el siguiente:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[256];

    if(argc > 1) {
        strcpy(buf, argv[1]);
        printf("%s", buf);
    }

    return 0;
}
```

Como se puede observar en el código, el programa se limita a copiar en un *buffer* denominado `buff` el primer parámetro (`argv[1]`) que se le pasa al programa cuando se le invoca, mostrándolo seguidamente por pantalla. Esta acción se realiza solamente si al programa se le pasa al menos un parámetro, de ahí la comparación que se hace con la variable `argc`.

Este código, aparentemente, no tiene ningún problema, sin embargo, ¿qué ocurriría si el primer argumento del programa superase la longitud de los 256 bytes que se han reservado para `buf`? Veamos su comportamiento al pasarle una cadena de 300 letras 'A':

```
student@uoc ~ $ gcc -fno-stack-protector -z execstack -o bof bof.c
student@uoc ~ $ ./bof `python -c "print 'A' * 300;" `
Violación de segmento
student@uoc ~ $
```

Parámetros *-fno-stack-protector -z execstack*

Los parámetros *-fno-stack-protector -z execstack* desactivan dos mecanismos de protección que en caso de no especificarlos no permitirían estudiar el problema de los desbordamientos de *buffer*.

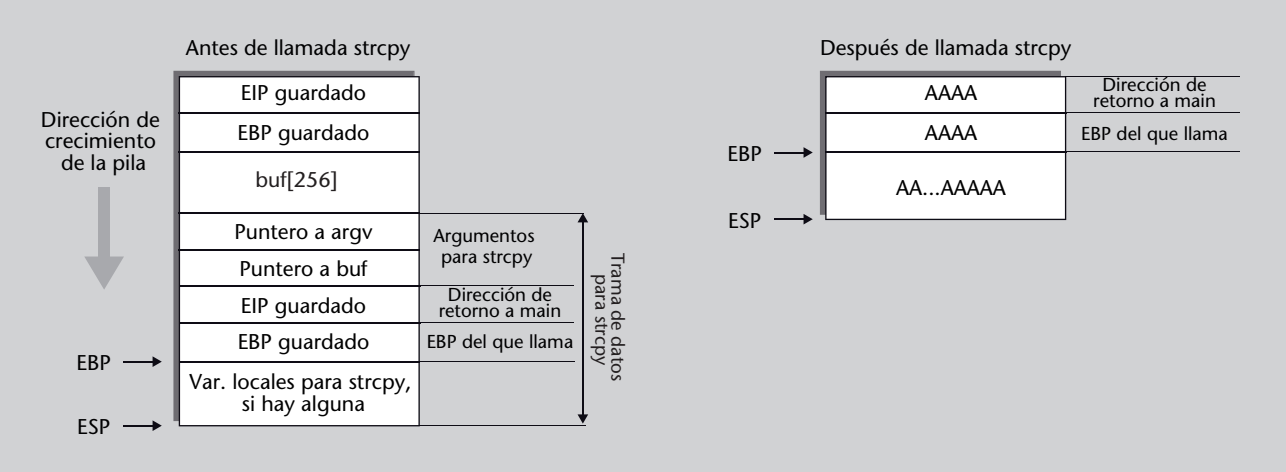
Se produce un error y la ejecución del programa se detiene sin producirse la salida esperada. Si ejecutamos el programa con el mismo parámetro pero con el *debugger gdb*, conseguiremos alguna información adicional de interés:

```
student@uoc ~ $ gdb ./bof
[...]
(gdb) run `python -c "print 'A'*300;" `
Starting program: /home/student/bof `python -c "print 'A'*300;" `

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info registers eip
eip            0x41414141       0x41414141
```

Se ha producido una violación de segmento al intentarse ejecutar una instrucción en la dirección `0x41414141`*****. Recordemos cómo evoluciona la pila al llamarse a una función y veremos dónde está la problemática. En primer lugar, previo a la llamada a la función `main`, se ha guardado en la pila la dirección de retorno así como el *frame pointer* anterior. A continuación, tras la llamada a la función `main`, se ha reservado espacio para `buf` al tratarse de una variable local. Antes de la llamada a la función `strcpy`, se han apilado en orden inverso sus correspondientes parámetros, es decir, el puntero al primer argumento del programa y el puntero a `buf`. Una vez dentro de la función `strcpy` (figura 3), esta copia los 300 bytes del primer parámetro sobre `buf`, sin embargo, si nos fijamos, este solo tiene reservados 256 bytes, lo que provoca la sobreescritura de la dirección de retorno de `main`. Esto conlleva a continuar el flujo de ejecución en la dirección `0x41414141` tras el retorno de `main`. De hecho, se puede observar cómo la dirección donde se produjo la violación de segmento `0x41414141`, codificada en formato ASCII, es '`AAAA`', que justamente coincide con parte de la cadena de texto que le pasamos al programa como primer parámetro.

*** Notad la última linea que muestra el contenido del registro `EIP`.**

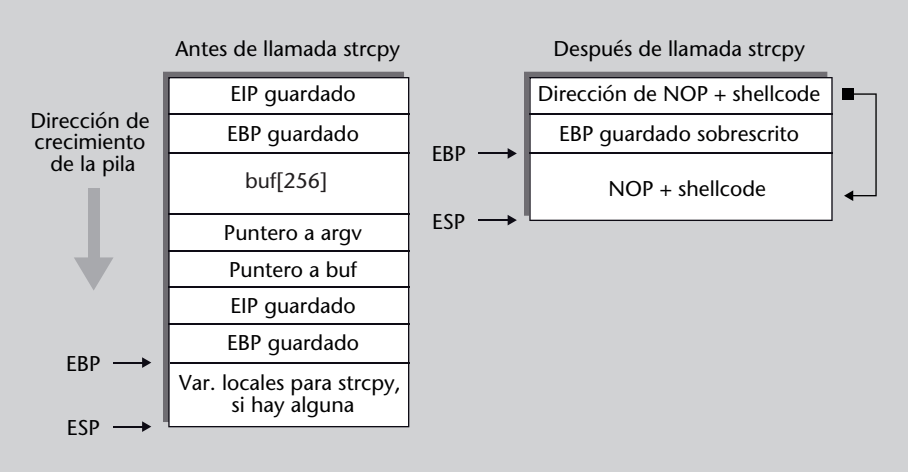
Figura 3. Evolución de la pila en un desbordamiento de *buffer*



¿De qué manera entonces se puede usar esto de forma maliciosa? Básicamente desviando el flujo del programa al modificar alguna dirección de retorno por una donde el intruso tenga un conjunto de instrucciones que le interesa ejecutar. En el caso de la variante *stack overflow*, esa dirección se corresponde con alguna del rango de direcciones ocupadas por el *buffer* destino de una función peligrosa y que se encuentra en la pila. Para conseguir dicha ejecución, se copiará sobre el *buffer* una cadena preparada con el conjunto de instrucciones a la vez que se provoca el desbordamiento. Esto supone que los *opcodes* de las instrucciones que se quieren ejecutar no podrán contener el carácter `NULL`, ya que de lo contrario `strcpy` (o alguna de las funciones consideradas como peligrosas) no copiará la cadena preparada en su totalidad y no se sobreescribirá la dirección de retorno.

En Internet se pueden localizar numerosas cadenas que se corresponden con instrucciones que realizan determinadas acciones útiles para un atacante, como puede ser añadir un usuario al sistema con una contraseña específica, u obtener un intérprete de comandos con privilegios de administrador. Estas cadenas son específicas para cada sistema operativo y arquitectura sobre la que se ejecuta y se denominan *shellcodes*. En la figura 4 se muestra cómo quedaría la pila del programa ejemplo al pasarle como parámetro una cadena preparada que permite la ejecución de código arbitrario.

Figura 4. Estado de la pila al desbordar un *buffer* con un *shellcode* preparado



Una consideración que los atacantes deben tener cuando preparan un *exploit* para este tipo de vulnerabilidades es el hecho de que no se conoce la dirección exacta de la pila donde el *shellcode* será copiado. El intruso puede analizar el programa vulnerable en su máquina para obtener la dirección de retorno exacta, sin embargo, ésta puede variar en función de la versión del núcleo del sistema operativo o de la librería del sistema *libc*. Esto puede suponer que el programa que permita aprovecharse de la vulnerabilidad no pueda ser reutilizado para otros sistemas. Para solventar esta problemática, se suelen poner delante del *shellcode* un conjunto de instrucciones (del orden de unos cientos) que no realizan ningún tipo de acción, como podría ser instrucciones `NOP` (*No-Operation*). A partir de esto, la dirección de retorno se fija para que caiga sobre algún lugar de dicho conjunto de instrucciones. De esta manera se aumentan las posibilidades de éxito independientemente de la versión del núcleo o de la librería *libc*, permitiendo que el *exploit* pueda ser usado en distintas máquinas.

2. Software malicioso

A pesar de que la idea del código malicioso pueda parecer novedosa, la realidad es que si queremos buscar sus orígenes debemos remontarnos al año 1949. En este año, el matemático John von Neumann presentó varias conferencias en la Universidad de Illinois bajo el nombre de *Theory and Organization of Complicated Automata*, en las que englobaba la teoría sobre autómatas complejos. En estas conferencias, Neumann estableció la idea de programa almacenado y teorizó por primera vez la posibilidad de que un programa pudiera replicarse por sí mismo. Sin duda alguna, esto supuso un resultado plausible para la teoría de la computación. Posteriormente, su investigación fue publicada en el año 1966 en el libro titulado *Theory of self-reproducing automata*.

Durante los años setenta aparecieron los primeros programas capaces de autorreplicarse según las teorías que ya había postulado John von Neumann tiempo atrás. Fue más tarde, en el año 1983, cuando Frederick B. Cohen acuñó el término *virus* para referirse a un programa capaz de autorreplicarse. Un año más tarde, y según las sugerencias de su mentor Leonard M. Adleman, Cohen empleó el término *virus informático*.

Virus informático según Cohen

Cohen define un virus informático (*computer virus*, N. del T.) como un "programa que puede infectar a otros incluyendo una copia posiblemente evolucionada de sí mismo".

Cohen (1984)

Cohen profundizó en este campo, investigando sobre otras propiedades de los virus informáticos, y formalizó la definición de virus basándose en el modelo de la máquina de Turing en su tesis doctoral (Cohen, 1986).

En la década de los ochenta, y en paralelo a los trabajos de Cohen, los ordenadores personales se popularizaron y supuso un nuevo nicho de oportunidad para los virus informáticos. Esta fue la época de explosión de los virus, los cuales empezaron a incluir en su código rutinas con finalidades maliciosas. Sin duda, era el nacimiento del *malware*.

Desde la concepción de Neumann de autómatas capaces de replicarse, pasando por la definición formal de virus informático introducida por Cohen, la evolución del código malicioso ha sido constante y especialmente de carácter empírico. Cada día, nuevos tipos de código malicioso más complejos aparecen, hasta el extremo de que se han tenido que acuñar nuevos términos para referirnos a cada una de las nuevas variantes. En paralelo, se ha trabajado

en establecer definiciones formales válidas para cualquier forma de *malware*, como la presentada por Kramer y Bradfield (2010). Sin lugar a dudas, estos avances han sido posibles gracias a los trabajos previos de investigadores como Neumann, Cohen o Adleman, cuyas contribuciones han permitido establecer las bases para la lucha contra el *malware* actual.

Al margen de la definiciones teóricas propuestas por la comunidad científica para describir el *malware*, nosotros utilizaremos una definición más pragmática.

El ***malware*** es un tipo de software intrusivo y hostil que tiene como objetivo infiltrarse o dañar un sistema de información sin la aprobación ni el conocimiento de su propietario.

Origen del término

El término *malware* proviene de la contracción de las palabras inglesas *malicious* y *software*. En castellano utilizamos los términos *software malicioso* o *código malicioso* como traducción del vocablo de origen inglés.

2.1. Taxonomía del *malware*

A lo largo de los años, la proliferación, la diversidad y la sofisticación del software malicioso ha crecido de forma espectacular. Hoy en día resulta complejo establecer una taxonomía completa, de manera que cada categoría que la compone permita clasificar el *malware* de forma excluyente. Es decir, dado un espécimen de *malware* particular, es posible que este tenga características híbridas que pertenezcan a más de una de las categorías en el que podemos clasificarlo. Así, es habitual, por ejemplo, encontrar código malicioso que pueda ser considerado como de propagación y oculto al mismo tiempo. Por tanto, la taxonomía del software malicioso que presentamos aquí debe ser entendida como una categorización genérica, en la que un código malicioso dado puede pertenecer a más de una clase de forma simultánea.

En particular, consideramos que hay tres grandes categorías de software malicioso. Estas son, *malware de propagación automática*, *malware oculto* y *malware lucrativo*. Seguidamente definiremos cada una de ellas.

2.1.1. *Malware* de propagación automática

El *malware* de propagación es aquel cuya principal finalidad es la de extenderse de forma automática infectando nuevos sistemas de información.

Dependiendo de la forma que emplea para propagarse distinguimos dos subcategorías:

- ***Malware* de propagación por infección vírica**. Es aquel en el que el código malicioso se replica a sí mismo al añadirse a archivos ejecutables, o con la capacidad de ejecutar algún tipo de código. Asimismo, este modifica el código del programa original para que, en algún instante, el flujo de ejecución sea redirigido a las instrucciones del *malware*. Cada vez que el código malicioso toma el control, busca nuevos archivos no contaminados con la intención de infectarlos y, tras esto, devuelve el control al programa original. A esta misma categoría, y de forma análoga al caso anterior, pertenece el *malware* que infecta el *master boot record* (MBR) de los discos duros, lo que le permite ejecutarse tras las pertinentes operaciones de la BIOS, y tomar el control antes de que se cargue el sistema operativo. Al *malware* que emplea este tipo de propagación se le conoce tradicionalmente como virus, y su nombre se debe a la analogía que existe con los virus biológicos, y a cómo estos infectan y se multiplican dentro de las células de otros organismos. En ocasiones, el término *virus* se utiliza de forma errónea para referirse a otros tipos de *malware*, tales como el *spyware* o los troyanos, que no deben ser confundidos dado que estos no infectan a otros archivos.

- ***Malware* de propagación como gusano**. Es aquel que se replica a sí mismo empleando la red a la que está conectado el sistema infectado, enviando copias de sí mismo a otros sistemas de la red que no están contaminados. Este envío se realiza sin la intervención del usuario, y puede emplear varias estrategias para propagarse, tales como la explotación remota de un desbordamiento de *buffer* conocido, el envío indiscriminado de correos electrónicos infectados con el *malware*, las redes P2P, o los clientes de mensajería instantánea entre otros. Al *malware* que emplea este tipo de propagación se le conoce tradicionalmente como gusanos.

Origen del término

El origen del término *gusano* está atribuido a John Brunner, quien en la novela de ciencia ficción *The Shockwave Rider*, publicada en 1975, describe bajo este nombre un programa capaz de replicarse a sí mismo utilizando una red de ordenadores.

2.1.2. *Malware* oculto

El *Malware* oculto es un tipo de software malicioso que se caracteriza por intentar permanecer desapercibido para el usuario dentro del sistema infectado.

Dependiendo de la forma en la que intenta pasar desapercibido, y del propósito del *malware*, distinguimos entre tres categorías:

- ***Rootkits***. Este tipo de código malicioso tiene sus orígenes en los sistemas Unix, y su denominación se utilizaba para hacer referencia al conjunto de herramientas que permitían a un atacante obtener privilegios de ad-

Ved también

En el subapartado 4.2 de este módulo se analizan con más detenimiento las técnicas *rootkits*.

ministrador (*root*). Hoy en día, el término *rootkit* se emplea de forma más generalizada para designar al conjunto de técnicas que permiten eludir la detección y la eliminación de cualquier *malware*. Estas técnicas se basan en la modificación del sistema operativo de la máquina infectada a bajo nivel, permitiendo la ocultación de procesos, archivos o conexiones de red utilizadas por el software malicioso.

- **Troyanos**. El software malicioso que pertenece a esta categoría se caracteriza por estar enmascarado detrás de un supuesto programa legítimo. La víctima, previo a la instalación de dicho software, lo percibe como un programa que proporciona unas funcionalidades de su interés. Sin embargo, tras la instalación, y sin el consentimiento del usuario y sin que este sea consciente, el *malware* actúa escondiéndose detrás de un software aparentemente lícito. La actividad oculta que lleva a cabo el código malicioso puede ser de distinta índole, aunque normalmente suele estar enfocada al control remoto de la máquina infectada, o bien al robo de información. Para forzar a la instalación del troyano, el atacante puede emplear técnicas de ingeniería social para convencer al usuario, ya sea a través de un correo o de una página web, por ejemplo.

Origen del término

El origen del término *troyano* se debe a la analogía entre este tipo de software malicioso y el caballo mitológico, según se relata en la Odisea, empleado por los griegos durante la Guerra de Troya.

- **Puertas traseras (*backdoors*)**. Esta categoría engloba a todo software malicioso que es instalado en un sistema ya comprometido, y que permite eludir los mecanismos de autenticación a la vez que permanece oculto a los administradores del sistema. De esta manera, una puerta trasera en una máquina infectada permite a un atacante garantizar el acceso al sistema en un futuro de una manera sencilla y rápida. A pesar de que la existencia de las puertas traseras no es una idea nueva, la proliferación de este tipo de código malicioso ha tenido especial relevancia desde la explosión de Internet. La red de redes ha permitido a los atacantes instalar puertas traseras que garantizan el acceso a sistemas comprometidos de forma remota. En ocasiones las puertas traseras pueden adoptar forma de troyano e incluso puede incluir técnicas de *rootkits*.

2.1.3. *Malware* lucrativo

El software malicioso que pertenece al *malware* lucrativo se caracteriza, como sugiere su nombre, por proporcionar algún tipo de beneficio al atacante.

Aunque no siempre tiene por qué ser así, en la mayoría de ocasiones el tipo de beneficio que se persigue es de carácter económico. Dependiendo de la finalidad que persigue y de cómo actúa, distinguimos entre las siguientes subcategorías:

- ***Spyware***. Es un software malicioso que registra información sensible de usuarios sin su consentimiento, violando la privacidad de estos. La información recogida por este tipo de aplicaciones puede ser de distinta índole, como por ejemplo, datos personales, números de tarjeta de crédito, hábitos de navegación web, contraseñas, pulsaciones de teclas, o incluso capturas de pantalla. Esta información se transmite a terceras partes con finalidades como son el fraude electrónico, el marketing a través de publicidad web no consentida, u otras actividades maliciosas.

keyloggers

El término *keylogger* se emplea para designar el *spyware* que registra las pulsaciones del teclado. El mismo vocablo se usa también para hacer referencia a dispositivos hardware que tienen la misma finalidad.

- ***Ransomware***. Es un tipo de *malware* que extorsiona a los usuarios propietarios de una máquina infectada, exigiendo algún tipo de pago tras cifrar archivos, o bien desactivar o bloquear partes del sistema. Si el usuario realiza el pago –usualmente vía transferencia bancaria o SMS con cargo adicional–, el atacante proporciona algún mecanismo para eliminar el perjuicio causado por el mismo código malicioso. En el caso particular del *malware* que hace uso de la criptografía para cifrar archivos se le conoce como *criptovirus*. Estos suelen basarse en esquemas criptográficos asimétricos o híbridos, lo que impide el acceso al contenido de los archivos cifrados al no disponer la víctima de la clave privada.

- ***Scareware***. Es un código malintencionado que, basándose en estrategias de ingeniería social, puede proporcionar beneficios económicos al atacante. En concreto, explota el engaño, la persuasión, la coacción o el miedo a través de mensajes de alarma o de amenaza para forzar a la víctima a realizar un pago. El ejemplo más común es el de programas que detrás de la apariencia de software para la detección de *malware* esconden este tipo de código malicioso. Una vez instalado, el programa reporta la existencia de una cantidad elevada de software malicioso en el sistema, cuando la realidad no es así. Entonces, el *scareware* brinda la posibilidad al usuario de eliminar las amenazas detectadas pagando por una versión diferente del software de detección. A este tipo de *scareware* se le conoce con el nombre de *rogueware*. Otras formas de *scareware* se basan en explotar el sentimiento de culpa y el miedo que pueden sentir algunos usuarios al descargar software ilegal. En particular, una vez instalado, este tipo de *scareware* muestra mensajes de advertencia indicando que se están violando las leyes del *copyright* y que se tiene identificada la IP del usuario. Seguidamente, propone evitar un juicio realizando un pago como forma de solventar la situación.

- ***Bot***. Es un código malicioso que permite a un atacante controlar de forma remota la máquina que lo ejecuta. Al conjunto de máquinas distribuidas e infectadas por *bots*, y controladas por un atacante, se le conoce con el nombre de *botnet*. La suma de recursos proporcionados por cada máquina infectada permiten realizar actividades fraudulentas, como son ataques de denegación de servicio distribuidos o el envío masivo de SPAM.

Ved también

Para saber más sobre las *botnets* podéis consultar el módulo didáctico "Botnets".

- ***Adware***. Es un tipo de *malware* que de forma automática muestra publicidad no consentida al usuario, con la finalidad de que realice algún tipo de

compra. Esta publicidad suele aparecer en los navegadores web como ventanas emergentes, siendo un comportamiento molesto e indeseable para el usuario. Algunos tipos de *adware* pueden ser clasificados también como *spyware*, ya que la publicidad mostrada va acorde con información registrada de la actividad del usuario.

- ***Dialers***. Es un tipo de software malintencionado cuyo objetivo es modificar la configuración del programa de marcado de los módems. En particular, modifican el número de teléfono a marcar por otro cuya tarifa es más elevada en comparación con la asociada al número legítimo. Esto supone un aumento en el importe de la factura telefónica del usuario, siendo el atacante el beneficiario. Este tipo de *malware* tuvo éxito cuando el acceso a Internet se realizaba con un módem conectado a la red de telefonía conmutada. Hoy en día, este tipo de software está en declive, ya que la mayoría de tecnologías actuales para el acceso a la red Internet funcionan de forma diferente.

2.2. Vectores de infección

Uno de los aspectos importantes a tener presente contra la lucha del *malware* es la identificación de los posibles vectores de infección. Conocer estas vías de infección nos permite centrar nuestros esfuerzos en diseñar e incorporar mecanismos de seguridad adecuados.

En términos generales, podemos distinguir dos estrategias posibles para la infección de un sistema: los procesos de infección iniciados por el usuario víctima, y los procesos de autoinfección iniciados a través de vulnerabilidades existentes en los sistemas.

En el primer caso, el *malware* suele venir camuflado en programas supuestamente legítimos y que, sin el consentimiento del usuario, se instala junto a este software. Así, programas de tipo P2P, complementos para los navegadores web, software descargado de forma ilegal, o *cracks*, son ejemplos de programas que pueden esconder *malware*. En otras ocasiones, la vía de infección se basa en el acceso a una determinada web con componentes ActiveX o Applets Java especialmente preparados, que, tras la autorización de su ejecución por parte del usuario, conllevan la instalación del código malintencionado. En cualquier caso, el proceso de infección requiere una aprobación de ejecución tácita por parte del usuario. La utilización de la ingeniería social suele estar presente en esta metodología, pudiéndo considerarse incluso un factor decisivo en la consecución del éxito para los atacantes. De esta manera, los atacantes utilizan estrategias para persuadir a los usuarios a descargar determi-

nado software, o realizar determinadas acciones que conlleven la instalación del *malware*.

En el segundo caso, los mecanismos de infección del código malicioso se basan en la explotación de vulnerabilidades en el software. Así, la visita a una determinada web preparada haciendo uso de un navegador vulnerable, o la apertura de un archivo especialmente preparado mediante software que incluya deficiencias de seguridad, pueden provocar la ejecución de código arbitrario que conduzca a la instalación del *malware*. En otras ocasiones, un servicio vulnerable ofrecido en una red a través de un puerto TCP/IP puede ser explotado por el *malware* para la infección. En cualquier caso, en este proceso de infección, la instalación del software malicioso suele pasar totalmente desapercibida para el usuario afectado, sin requerirse ningún tipo de acción a realizar que pueda considerarse motivo de sospecha.

2.3. Mecanismos de prevención

Podemos considerar cuatro métodos principales para prevenir nuestros sistemas de las infecciones de *malware*:

1) Fomento de buenas prácticas. En primer lugar, fomentar las buenas prácticas por parte de los usuarios, manteniendo actualizado tanto el sistema operativo como las aplicaciones, impedir las descargas de software de fuentes no fiables o ignorar los correos y contenidos adjuntos de remitentes desconocidos. Desafortunadamente, estas prácticas no se cumplen muchas veces por parte de los usuarios, ni tampoco son completamente efectivas.

2) Diseño de patrones de protección. Otras formas de prevención más técnicas se basan en el diseño de patrones de protección. El objetivo de estos patrones es impedir la infección de un sistema o bien reducir el daño de la infección. Podemos agrupar dentro de esta categoría la utilización de anillos de protección. Dichos anillos establecen una estructura de confianza por capas en el sistema operativo y se complementan con hardware específico para poder realizar una separación efectiva entre procesos de confianza y procesos sospechosos. A través de esta solución, se pueden ofrecer distintos niveles de acceso a los recursos del sistema. De hecho, los anillos se organizan de forma jerárquica, estructurando aquellos dominios más privilegiados y confiables hasta los de menores privilegios y nivel de confiabilidad. De este modo, se reduce el riesgo de que procesos de tipo *malware* ataquen al núcleo del sistema operativo (lo que les permitiría obtener el control del sistema al completo). Estos métodos han demostrado que, aunque reducen las consecuencias de una infección, no son totalmente efectivos.

3) Uso de firmas digitales. El tercer mecanismo genérico consiste en verificar la autenticidad del código que se está ejecutando mediante la utilización de firmas digitales. Estas firmas se asociarán al código y se verificarán antes de su

ejecución. Las dificultades surgen aquí para aquel código que no haya sido firmado. En este caso, los usuarios deberán decidir entre no usar sus funcionalidades, o bien exponerse a ciertos niveles de riesgo si dicho programa es lanzado. La práctica ha demostrado que, ante situaciones de este tipo, un porcentaje elevado de usuarios prefieren ejecutar el software antes que verificar su legítima procedencia o validez.

4) Uso de aplicaciones automáticas. Por último, y ante la ineficiencia de los métodos anteriores, las tendencias actuales se centran en la investigación de aplicaciones automáticas capaces de detectar y aislar código de tipo malicioso.

Ved también

Dada la especial relevancia que en la actualidad tienen las aplicaciones automáticas, nos dedicaremos a analizar su funcionamiento en el apartado 3 de este módulo.

3. Detección de *malware*

Sin duda alguna, la detección del software malicioso ha sido una de las medidas más extendidas como mecanismo de prevención. En este apartado se analizan las técnicas más comunes empleadas por software especializado en la detección de *malware*. Como veremos, la complejidad de estas estrategias varía entre ellas. Esta divergencia de complejidades obedece a la evolución que ha experimentado el código malicioso a lo largo del tiempo, cuya finalidad siempre ha sido evadir los sistemas de detección. Así, software malicioso que ha incorporado mecanismos para dificultar su detección ha requerido de nuevas formas más sofisticadas de análisis.

A pesar de los progresos que se han hecho en el campo del software para la detección de *malware* en los últimos años, es importante remarcar que, tal y como ya postuló Cohen (1987) en sus trabajos sobre los virus, el problema de la **detección perfecta** de software malicioso es un problema indecidible.

Detección de *malware*

La detección perfecta de *malware* es un problema indecidible y que, por tanto, no existe ningún programa capaz de detectar todo código malicioso posible. Asimismo, la erradicación de un código malicioso en un sistema infectado no siempre está garantizada, ya que ésta depende de la detección.

No existe un programa capaz de detectar la totalidad de variantes de código malicioso que pueden llegar a existir. Por lo tanto, no siempre será posible realizar una detección proactiva, lo que conduce, en alguna ocasión, a una inevitable infección de los sistemas.

Sin embargo, esto no significa que no podamos diseñar mecanismos que nos permitan detectar y luchar contra un subconjunto de la totalidad del espectro del *malware*.

En la actualidad, múltiples estrategias y modelos han sido propuestos para la detección del *malware* a través de software especializado. Todos lo modelos de detección de *malware* pueden ser clasificados en dos categorías en función del tipo de detección que realizan. En particular, tenemos los modelos *imprecisos* y los *exactos*. En el caso de los imprecisos, el motor de detección sólo es capaz de determinar si un determinado objeto se trata de *malware* o no, sin llegar a precisar los detalles de la versión o variante de código malicioso de que se trata. En contraposición, los modelos exactos son capaces de realizar una detección concisa, proporcionando información particular acerca de la versión de software malicioso detectado. Evidentemente, el uso de un modelo u otro tiene consecuencias en el procedimiento de eliminación del código malicioso en un sistema infectado, siendo en el *exacto* un proceso más directo al conocerse los detalles de la infección de antemano.

Entre las diferentes tecnologías para la detección nos centraremos en el estudio de dos concretas por su amplio uso en la actualidad. En primer lugar, analizaremos la detección sintáctica basada en firmas, un modelo exacto que, por su naturaleza, nos impide luchar contra mecanismos de ofuscación que pueden incorporar el *malware*. En segundo lugar, veremos cómo la detección semántica se presenta como una manera de superar las deficiencias de la detección basada en firmas, sin embargo, éste se basa en un modelo impreciso con las consecuencias que esto supone.

3.1. Detección sintáctica basada en firmas

Desde un punto de vista histórico, la detección sintáctica basada en firmas fue la primera técnica empleada para la identificación de software malicioso. Actualmente, esta tecnología continúa siendo parte del núcleo de los motores de detección de *malware*, y se caracteriza por su ratio baja de falsos positivos.

La detección sintáctica basada en firmas es una estrategia que se basa en localizar dentro de objetos potencialmente maliciosos (comúnmente ficheros o procesos) algún patrón que identifique a un determinado *malware* conocido.

Estos patrones –también conocidos como *firmas sintácticas*– vienen expresados como una secuencia de bytes, y representan cadenas de texto o instrucciones de bajo nivel en forma de *opcodes*. Como veremos más adelante, existen varios tipos de firmas con características diferentes.

Los motores de detección sintáctica disponen de una base de datos de estas firmas que definen el conjunto de software malicioso reconocible. Si el motor encuentra alguna de las firmas de la base de datos en un objeto, este se identifica de forma fehaciente como un *malware* específico. Para conseguir localizar firmas dentro de un objeto, diversos algoritmos de búsqueda y concordancia pueden ser empleados. Estos algoritmos están optimizados y tienen una complejidad acotada en función del tipo de firma que se utiliza. Asimismo, estos motores pueden mejorar el rendimiento al limitar la búsqueda a partes estratégicas de los objetos, y no realizar una búsqueda exhaustiva en todo su contenido.

La tecnología de detección de software malicioso basado en firmas sintácticas presenta diversas deficiencias, lo que lo convierte en un método ineficaz bajo ciertas circunstancias:

1) Las firmas de una base de datos siempre están asociadas a software malintencionado conocido, no permitiendo la detección de nuevas formas de

código malicioso. Así, si un nuevo *malware* es liberado y su firma no está presente en la base de datos de un sistema, este podría ser infectado al pasar desapercibido para el motor de detección. De hecho, algo tan simple como la modificación de un código malicioso conocido a nivel de la firma que lo detecta permite evadir su detección. En este sentido, el software malicioso ha ido incorporando a lo largo del tiempo estrategias de evasión que se benefician de esta debilidad. Dichas estrategias se basan en crear, de forma automática, mutaciones del código malicioso en cada nueva infección. De esta manera no es posible establecer patrones de detección únicos, ya que una firma válida para un *malware* en concreto no será útil para la identificación de su versión mutada. De hecho, se ha demostrado cómo una detección basada en firmas sintácticas contra mutaciones de tipo polimórfico o metamórfico conduce a un problema de tipo NP-Completo (Spinellis, 2003) o incluso indecidible (Filiol, 2007) respectivamente.

2) La generación de las firmas puede conllevar un proceso largo y tedioso de análisis, lo que supondría dejar expuestos a una infección a los sistemas durante un tiempo elevado. En particular, cuando se localiza un objeto que se presume como malicioso, es habitual aplicar ingeniería inversa a su código para determinar si se trata o no de *malware*. Tras este análisis, y una vez que el objeto puede ser catalogado como código malicioso, se buscan características que lo identifiquen de forma única. A partir de estas características de unicidad se genera la firma, que posteriormente se distribuye e incorpora a las bases de datos. Este proceso, que requiere de intervención humana, no siempre es trivial, y el tiempo necesario se ve influenciado por las técnicas incluidas en el *malware* para dificultar su análisis. Por tanto, desde que el software malintencionado se libera, hasta que las firmas son creadas e incorporadas a las bases de datos, existe un periodo de tiempo crítico durante el cual no es posible la detección por parte de los motores sintácticos. Nótese la estrecha relación que existe entre este problema y el tratado en el punto anterior.

3) Y por último, a pesar de que la distribución de las firmas en los mejores casos se realiza de forma automática, existen motores en los que se requiere la intervención del usuario para lanzar el proceso de actualización de la base de datos. Si este proceso manual no se efectúa a tiempo, un sistema puede verse comprometido por el software malintencionado al no poderse detectar.

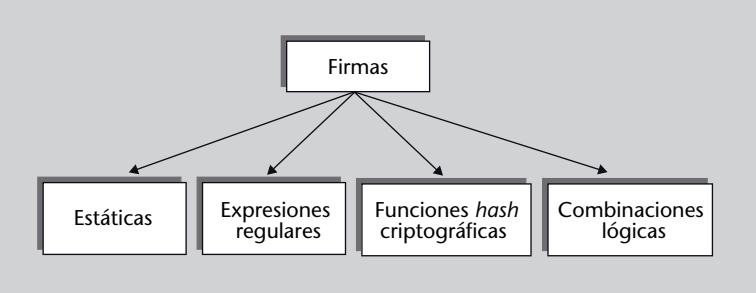
3.1.1. Tipos de firmas

En el subapartado 3.1 hemos podido ver cómo las firmas sintácticas son el pilar de la metodología de detección que estamos tratando. Con el objetivo de mejorar la precisión en la identificación de *malware*, cuatro tipos de firmas sintácticas son utilizadas. De forma más concisa, éstas son (figura 5):

1) Firmas como cadenas estáticas. Las firmas como cadenas estáticas son la forma más básica entre las distintas posibles. Estas vienen definidas como una secuencia de bytes consecutivos de longitud arbitraria. Si esta secuencia

se localiza en un objeto analizado, entonces es identificado como código malicioso.

Figura 5. Tipos de firmas sintácticas



2) Firmas con expresiones regulares. Este tipo de firmas soporta expresiones regulares en su definición. De esta manera es posible localizar concordancias de bytes según repeticiones, rangos, combinaciones, etc. Debido a esto, el algoritmo de búsqueda y concordancia tiene una mayor complejidad que el de las cadenas estáticas y, por tanto, menor velocidad en el proceso de identificación de *malware*. Sin embargo, el soporte de expresiones regulares le dota de mayor flexibilidad para detectar software malicioso más complejo, o variantes respecto a una versión de *malware* ya conocida. Así, a menudo, una única firma que utiliza expresiones regulares permite detectar toda una familia de variantes de un determinado código malicioso. Como es lógico pensar, estas firmas genéricas son útiles si, al liberarse una variante, esta comparte el mismo patrón identificativo con versiones anteriores.

3) Firmas basadas en funciones *hash* criptográficas. Esta variante de firmas sintácticas se sustenta en el uso de funciones *hash* criptográficas. Recordemos que estas funciones se caracterizan por las siguientes propiedades:

a) Compresión: el valor de aplicar una función *hash* a un mensaje tiene un tamaño constante, y generalmente inferior al tamaño del mensaje.

b) Unidireccionalidad: dado el resumen de aplicar una función *hash* a un mensaje, es imposible reconstruir el mensaje original.

c) Facilidad de cálculo: dado un mensaje de longitud arbitraria, es fácil y rápido (para un ordenador) calcular su función *hash*.

d) Difusión: dado un mensaje m, si modificamos tan sólo un bit de m y lo denominamos m', entonces el valor $hash(m')$ debería modificar aproximadamente la mitad de los bits respecto a $hash(m)$.

e) Resistencia débil a colisiones: dado un mensaje m, no es computacionalmente factible encontrar otro mensaje m' tal que $m \neq m'$, pero que $hash(m) = hash(m')$.

f) Resistencia fuerte a colisiones: no es computacionalmente factible encontrar una pareja de mensajes m, m' tal que $m \neq m'$, pero que $hash(m) = hash(m')$.

Estas propiedades permiten guardar las firmas como el cálculo de una función *hash* sobre un objeto (o parte de este) correspondiente a código malicioso. Las ventajas del uso de este tipo de firmas es doble. En primer lugar, pueden reducir el espacio necesario para almacenar una firma, siempre y cuando el tamaño resultante de calcular la función *hash* sea inferior al tamaño de otro tipo de firma alternativa. En segundo lugar, una firma sintáctica basada en funciones criptográficas puede ser computacionalmente menos costosa de localizar que los algoritmos de búsqueda de otros tipos de firmas.

4) Combinaciones lógicas de firmas. Este tipo de firmas combina múltiples subfirmas sintácticas relacionándolas mediante operadores lógicos, permitiendo definir patrones más flexibles y precisos. Por tanto, la búsqueda de la nueva firma para la identificación de un determinado *malware* pasará por verificar que se cumple la expresión lógica compuesta por todas las subfirmas. Esta estrategia puede ser utilizada para, por ejemplo, detectar una familia de variantes de un código malicioso concreto. Así, se podría definir una firma sintáctica genérica para todas las variantes, y luego una específica para cada variante particular. Para este escenario, combinaríamos con el operador lógico `AND` la firma genérica con la particular para la detección de una variante concreta.

Cada una de estas cuatro firmas tiene asociado un algoritmo de búsqueda y concordancia para la identificación de código malintencionado, y cuya complejidad algorítmica es diferente para cada uno.

La utilización de un tipo u otro de firma varía en función del software malicioso que debe identificar y, por tanto, no se puede afirmar que ninguna de ellas sea mejor que otra. Dependiendo del caso particular de software malintencionado, la elección de una u otra mejorará la precisión y el tiempo necesario en la identificación. Es habitual encontrar motores de detección sintáctica que soportan varios tipos de firmas de forma simultánea.

3.1.2. Ámbitos de búsqueda

Con el objetivo de mejorar el rendimiento en el proceso de detectar software malicioso, cada firma sintáctica tiene asociado un ámbito de búsqueda.

Entendemos por *ámbito de búsqueda* a una restricción que limita la localización de la firma a un tipo de objeto o a una parte de su contenido. Esto permite restringir el proceso de búsqueda, con el consiguiente ahorro de cómputo en comparación a una búsqueda exhaustiva en todos los tipos de objeto y/o en la totalidad de sus contenidos.

En algunas ocasiones, estos ámbitos de búsqueda pueden ser combinados para acotar aún más la localización de las firmas. Distinguimos cinco maneras diferentes de especificar ámbitos de búsqueda. Estos son:

1) Totalidad. Este ámbito de búsqueda se refiere a la totalidad del objeto a analizar. Es decir, la firma se intentará localizar en cualquier posición dentro de todo el objeto. Por lo tanto, desde un punto de vista de cómputo, se corresponde con el caso más costoso dado que requiere una búsqueda exhaustiva.

2) Desplazamiento. Este tipo de ámbito especifica el punto inicial de la búsqueda dentro de un objeto a partir de un desplazamiento. Este desplazamiento, indicado en número de bytes, puede hacerse respecto al inicio del objeto (desplazamiento positivo) o respecto al final (desplazamiento negativo). Asimismo, este desplazamiento también puede realizarse con relación a secciones específicas y no sobre la totalidad del objeto. Algunas firmas también permiten definir en su ámbito de búsqueda un tamaño de bytes respecto el desplazamiento, lo que define una porción acotada del objeto donde se realizará la localización.

3) Filtros de objetos. Este ámbito de búsqueda restringe la localización de las firmas a objetos que cumplen una serie de características, como puede ser tipo de archivo, tamaño del objeto, número de secciones contenidas, etc.

4) Secciones específicas. El contenido de los objetos en un sistema suele estar estructurado de una manera específica que depende de su tipo. Así, por ejemplo, la estructura de los ejecutables en la plataforma Windows conocida como PE (*Portable Executable*) es diferente de la de los documentos de tipo PDF (*Portable Document Format*), o de las imágenes JPG. La organización de estas estructuras suele hacerse en secciones o cabeceras, y cada una de ellas contiene un tipo de información particular. El ámbito de búsqueda basado en secciones específicas se centra precisamente en restringir la búsqueda a alguna de estas secciones. Por tanto, un ámbito de este tipo deberá especificar la sección particular del objeto donde debería localizarse la firma. Como es lógico pensar, este tipo de ámbito se combina con el visto en el punto anterior, dada la dependencia que existe entre la estructura interna de los objetos y su tipo.

5) Combinaciones de ámbitos. Las combinaciones de ámbitos son una forma de refinar aún más la porción del objeto donde localizar la firma. Para conseguir esto se emplean de forma simultánea algún tipo de combinación de los ámbitos descritos anteriormente.

Ejemplo

Así, una firma basada en *opcodes* podría especificar que solo se realizase la búsqueda en archivos de tipo ejecutable PE (ámbito de tipo *filtro de objetos*), en la sección de código (ámbito de tipo *secciones específicas*), y con un desplazamiento de 1.012 bytes (ámbito de tipo *desplazamiento*).

3.2. Detección semántica

La detección semántica difiere de la detección sintáctica al fundamentarse en la identificación de acciones llevadas a cabo por el *malware*, y no por la localización de patrones de bytes en objetos.

Terminología

La detección semántica es también conocida como *detección basada en comportamiento*.

Identificar estas acciones e interpretar su significado es una tarea compleja, pero que aporta ciertas ventajas respecto a la detección sintáctica. En particular, la detección semántica es más resistente ante técnicas de mutación como son el polimorfismo o el metamorfismo. Por tanto, esta estrategia puede ser vista como una forma de superar las deficiencias de la detección sintáctica, dado que una mutación no modifica el comportamiento final del *malware*. Esto puede ser entendido si consideramos también que las firmas sintácticas no tienen en cuenta la semántica de las instrucciones.

Ved también

El polimorfismo y el metamorfismo se estudian en el subapartado 4.1 de este módulo.

Las bases de la detección basada en comportamiento ya fueron establecidas por Cohen (1986, 1987) en sus primeros trabajos formales relacionados con los virus. A pesar de que su investigación se centró en el campo de los virus, estas mismas bases pueden ser extrapoladas hoy en día a cualquier otro tipo de software malicioso. En concreto, postuló que un virus, al igual que cualquier otro programa, utiliza los servicios proporcionados por el sistema. Por tanto, determinar si un programa específico se trata de software malintencionado pasa por establecer lo que es un uso ilegítimo de los servicios del sistema, y contrastarlo con las acciones realizadas por todo proceso. De acuerdo con lo expuesto, Cohen definió dos aproximaciones para la detección basada en comportamiento:

- La primera de ellas modela el comportamiento legítimo sobre la base de aplicaciones conocidas y no maliciosas. A partir de aquí, cualquier desviación con respecto a esta referencia se considera como acciones ejecutadas por *malware*. Esta primera aproximación tiene la capacidad de detectar cualquier tipo de software malicioso no conocido. Sin embargo, la complejidad de esta propuesta radica en definir dicho modelo de comportamiento legítimo. Esto cobra especial sentido si tenemos en cuenta la multitud de aplicaciones que existen y sus diferentes naturalezas, lo que conduce a que no sea posible extraer un perfil legítimo y único común a todas ellas. Por este motivo, esta estrategia siempre se basa en modelos estadísticos y es proclive a falsos positivos.

- La segunda aproximación, en contraposición con la anterior, se basa en modelar el comportamiento sospechoso de las aplicaciones maliciosas. En este caso, cualquier tipo de acciones detectadas que se aproximen a este modelo de referencia inducirán a la identificación del código malicioso. Esta segunda opción, a pesar de no poder detectar nuevo *malware* con tanta facilidad, se utiliza de forma más habitual al no ser tan sensible a los falsos positivos.

El modelado del comportamiento en la detección semántica pasa por definir un conjunto de firmas que empleará el motor de detección. A diferencia de las firmas sintácticas, las firmas semánticas requieren de estructuras más complejas al tener que reflejar aspectos dinámicos y de comportamiento. Estas firmas, cuando son definidas de forma correcta, permiten detectar una amplia gama de código malicioso, inclusive nuevas generaciones creadas a partir de un *malware* particular. Esto es posible ya que programas sintácticamente diferentes pero con el mismo comportamiento pueden ser detectados por una única firma. Sin embargo, dado que no es posible una identificación precisa con las firmas semánticas, se puede incurrir en un problema en el proceso de erradicación del código malicioso, especialmente cuando éste es de carácter infeccioso como los virus. Adicionalmente, una de las ventajas de las firmas semánticas en comparación con las sintácticas es el menor nivel de crecimiento de sus bases de datos. Recordemos que, en el caso de la detección sintáctica, necesitamos firmas adicionales por cada nueva forma o variante de *malware*, mientras que en la detección semántica no tiene por qué ser así. Como consecuencia, la distribución y la actualización de las bases de datos no tiene por qué ser tan frecuente como en la detección sintáctica.

La recolección de información semántica para la detección de software malicioso puede realizarse según tres contextos diferentes:

1) El primero, y el más elemental, pasa por una captación de la información en tiempo real en un sistema. Bajo esta opción, si además se registran las acciones y los estados intermedios del entorno de ejecución, es posible devolver el sistema a un estado "saludable" tan pronto como la amenaza es detectada.

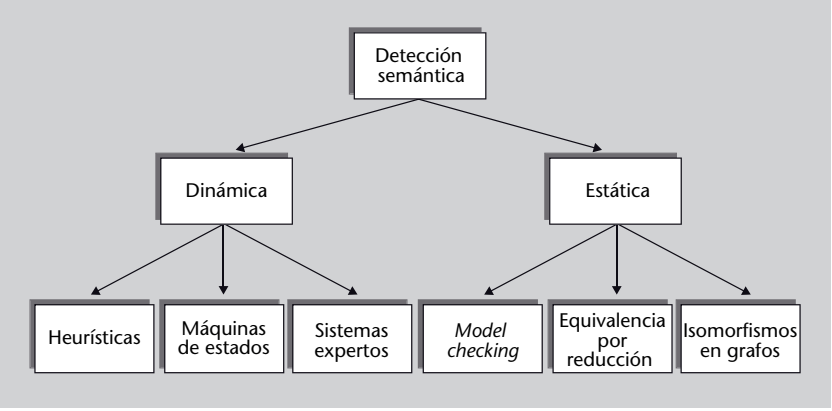
2) Una segunda opción pasa por la emulación de cualquier código antes de su ejecución real en un entorno controlado denominado *sandbox*. Las limitaciones de esta estrategia aparecen si un código malintencionado es capaz de detectar el *sandbox*, lo que le permitiría adaptarse y comportarse como software benigno.

3) Por último, el uso de máquinas virtuales puede ser utilizado para la captación de la información, al ser éstas capaces de virtualizar todo un sistema. De nuevo, si existe la posibilidad de detectar la máquina virtual, el *malware* podría modificar su comportamiento y evadir la detección.

Lectura recomendada

Existen trabajos que remarcan el peligro de las máquinas virtuales y la posibilidad de infección de la máquina anfitrión. Es lo que defienden Embleton y otros (2008); King y otros (2006).

Figura 6. Taxonomía de la detección semántica



Independientemente del contexto, la detección basada en comportamiento puede dividirse en dos categorías principales. La diferencia entre estos dos tipos de detección radica en la forma en la que se capta la información. Por un lado, el *análisis dinámico* considera las acciones llevadas a cabo en tiempo real, mientras que el *estático* obtiene las acciones sin la ejecución del código. Seguidamente analizaremos en mayor profundidad cada una de estas dos categorías, y veremos qué alternativas existen en cada una de ellas de acuerdo a la taxonomía presentada por Jacob, Debar y Filiol (2008).

3.2.1. Análisis dinámico

El análisis dinámico considera las acciones llevadas a cabo en el sistema por parte de todo programa en ejecución. A partir de la información de estas acciones y una base de conocimiento en forma de firmas, el motor de detección será capaz de catalogar un proceso como malicioso.

Las acciones pueden ser analizadas gracias a la interceptación de las llamadas al núcleo del sistema operativo, también conocidas como *syscalls*. Para esto, el motor de detección se interpone entre la interfaz de llamadas y el código de las *syscalls*. De esta manera, cada vez que un programa realiza una llamada al núcleo, el motor de detección toma el control antes de ejecutar el código de la *syscall* correspondiente. Para la detección, no sólo se tiene en cuenta la llamada realizada, sino también sus parámetros, el identificador de proceso que la realizó, su nivel de privilegios, así como cualquier otra información de contexto que pueda ser de utilidad.

Es importante remarcar que esta misma técnica puede ser utilizada por *malware* que emplea técnicas de *rootkits*, lo que podría suponer inhabilitar los mecanismos de detección. Asimismo, dado que el motor de detección se ejecuta antes que las llamadas al núcleo, es de vital importancia que la penalización que introduzca en el sistema sea mínima. En caso contrario, la percepción que se podría tener del sistema es el de una ejecución lenta.

Ved también

Las técnicas *rootkits* se estudian en el subapartado 4.2.

A partir de la información de las llamadas, disponemos de diversas alternativas para tratar estos datos y poder detectar el software malintencionado. En particular, a continuación presentamos tres técnicas que difieren considerablemente en los métodos que emplean. En concreto, estas son: utilización de **heurísticas**, **máquinas de estados** y **sistemas expertos**.

Técnicas heurísticas

Históricamente, las utilización de heurísticas fue la primera forma de detectar comportamientos maliciosos. Esta técnica se basa en interpretar un conjunto

de acciones ejecutadas en secuencia. Dichas acciones pueden ser capturadas a través de la interceptación de las llamadas al sistema, usualmente mediante un *sandbox*. Los motores de detección heurísticos pueden seguir dos estrategias:

- La primera estrategia se basa en otorgar a cada acción atómica un valor cuantitativo en forma de peso. Este peso, obtenido a partir de la experimentación previa, expresa la gravedad de la acción de manera numérica. En este caso, desde la perspectiva del motor de detección, las firmas se corresponden precisamente a la asociación entre acciones y pesos. Cuando la suma acumulativa de estos pesos supera un determinado umbral para un programa específico, se considera que éste es código malicioso.

- La segunda técnica se basa en identificar cada acción atómica con una etiqueta en función de un conocimiento previo. Cada una de estas etiquetas estará asociada a un tipo de acción. El motor de detección tendrá como firmas secuencias de etiquetas que identifican diferentes tipos de *malware*. Cuando una secuencia de acciones llevadas a cabo por un programa originen un conjunto de etiquetas ordenadas y correspondientes a una firma, se identificará al código que las originó como malicioso. Por tanto, con cada nueva acción será necesario guardar su etiqueta asociada junto a las anteriores. Usualmente, el conjunto de todas la firmas pueden ser representadas en forma de árbol, donde cada nodo se corresponde a una etiqueta, y los nodos hoja identifican un *malware* particular. De esta manera, la identificación de software malicioso pasa por recorrer el árbol en función de la secuencia de etiquetas que se van generando, hasta llegar a un nodo hoja.

Máquinas de estados

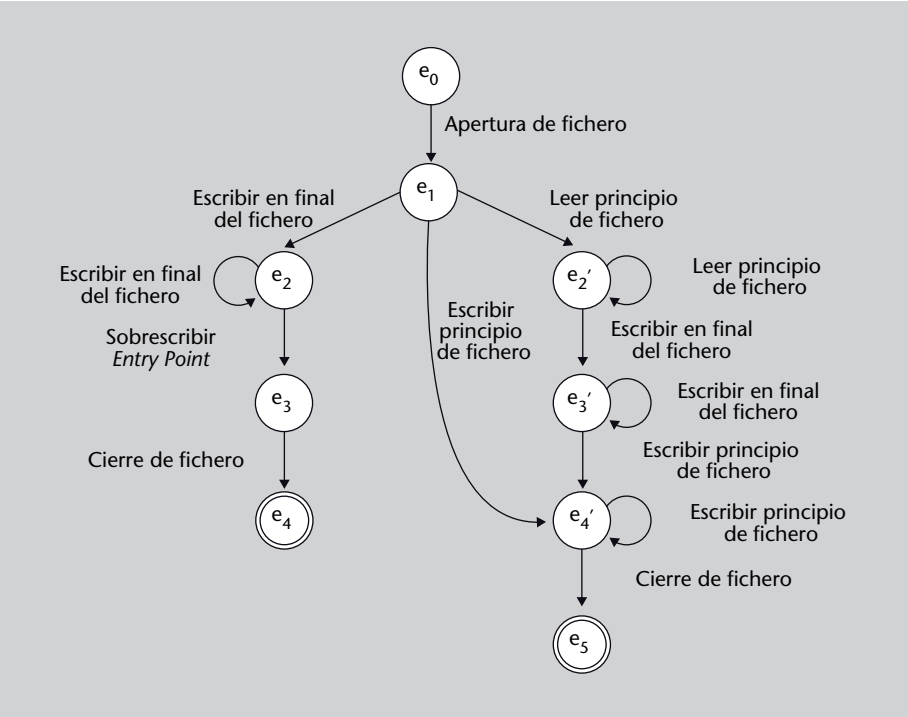
El uso de máquinas de estados también puede permitir la detección de *malware* en base a una secuencia de acciones. En particular, los comportamientos malintencionados son modelados como un autómata finito determinista $A = (Q,\Sigma,\delta,q_0,F)$ con las siguientes características:

- Los estados Q del autómata se corresponden a los estados internos del comportamiento malicioso.
- El estado inicial $q_0 \in Q$ se corresponde al inicio del análisis.
- El alfabeto Σ está compuesto por símbolos que consideraremos como llamadas al sistema.
- La función de transición $\delta : Q \times \Sigma \rightarrow Q$ describe los cambios de estado en base a llamadas sospechosas.
- Los estados de finales $F \subseteq Q$ implican la detección del comportamiento malicioso que define el autómata.

A partir de una instancia de un autómata determinado que define un comportamiento malicioso y dado un código en ejecución, el autómata progresará desde el estado inicial q_0 sobre la base de las llamadas que se realizan al siste-

ma. Si el autómata acaba en un estado final, el código asociado se le considera como malintencionado. En caso contrario, el código es catalogado como benigno. En la figura 7 se puede observar uno de estos autómatas.

Figura 7. Autómata finito determinista como firma para la detección de un comportamiento de infección vírica (Jacob y otros, 2008)



Sistemas expertos

Los sistemas expertos se basan en un conjunto de reglas modeladas por un analista para situaciones particulares. En concreto, estas reglas se definen para cada acción sospechosa en términos de los servicios del sistema que se utilizan. Asociada a cada una de estas reglas hay una decisión sobre la acción, aceptándola o denegándola. Cada vez que se realiza una acción de forma individual por parte de un proceso, esta se compara con el conjunto de reglas y, en caso de encontrar una regla que concuerde, se ejecuta la decisión vinculada a la regla. En cada acción se tiene en cuenta también el nivel de privilegios de quien lanzó la acción, ya que puede suponer la diferencia entre una acción legítima de otra que no.

3.2.2. Análisis estático

La detección basada en comportamiento utilizando análisis estático se sustenta en la extracción de las acciones realizadas por un potencial código malicioso sin su ejecución.

A diferencia del dinámico, el análisis estático proporciona una mayor cantidad de información y más completa al no realizar el análisis exclusivamente sobre acciones observadas. Para conseguir esto es necesario extraer la información semántica a partir del código binario, lo que requiere realizar diversos procesos para obtener una representación intermedia del programa. A partir de esta representación y de las firmas semánticas se podrá estimar la legitimidad del código que se está analizando.

Para llegar a dicha representación intermedia se emplean técnicas automatizadas de ingeniería inversa y desensamblado, lo que permite construir seguidamente los grafos de control de flujo* y los grafos de flujo de datos**. A partir de ambos grafos se pueden obtener posteriormente otro tipos de representaciones semánticas acordes al tipo de análisis estático que se esté realizando. Estos procesos no siempre son triviales, dado que el código malicioso suele incorporar mecanismos que dificultan el proceso de desensamblado, tales como técnicas de ofuscación, *antidebugging*, etc.

*** En inglés, *Control Flow Graph* (CFG).**
**** En inglés, *Data Flow Graph* (DFG).**

Ved también

Las técnicas de evasión se estudian en el apartado 4.

Dentro de la categoría del análisis estático podemos identificar tres tipos de tecnologías diferentes para la detección del *malware*:

1) La verificación de modelos*. Este método se basa en una aproximación algebraica, empleando un algoritmo que, utilizando lógica deductiva, simplifica una representación abstracta del código del *malware* usando reglas de reescritura. Para conseguir esto, inicialmente esta estrategia debe realizar una transformación del código a dicha representación abstracta. Es importante destacar que las reglas de reescritura preservan la semántica del código malicioso, permitiendo luchar contra técnicas de mutación como el polimorfismo y el metamorfismo. Una vez que se ha obtenido la forma reducida, se verifica utilizando un intérprete si la ejecución exhibe algún comportamiento malintencionado de acuerdo a especificaciones de *malware* conocidos, y expresados también en la misma representación algebraica.

*** En inglés, *model cheking*.**

2) Equivalencia por reducción. Esta técnica se sustenta en utilizar como firmas semánticas fórmulas lógicas de primera clase, y que semánticamente se corresponden a acciones maliciosas. El motor de detección toma como entrada el grafo de control de flujo y las firmas como fórmulas lógicas. Seguidamente, verifica cuáles de estas fórmulas se corresponden a estados intermedios en todos los posibles caminos de ejecución del grafo. En el caso de detectar la correspondencia entre las fórmulas y los estados intermedios, el objeto es catalogado como malicioso. Dado que existen infinitos posibles caminos de ejecución a explorar, el algoritmo es altamente recursivo y costoso en términos de recursos. De la misma manera que el *model cheking*, esta estrategia permite la detección independientemente de si el *malware* utiliza técnicas de evasión basadas en ofuscación de código.

3) Isomorfismos en grafos. El isomorfismo en grafos emplea el grafo de control de flujo para realizar la detección de software malicioso. En particular, a cada nodo del grafo se le asigna una etiqueta semántica que describe el tipo

de acción que desempeña el bloque básico asociado. Dada esta representación semántica del potencial código malicioso, esta se coteja con las firmas semánticas de la base de datos del motor de detección. En este caso, las firmas semánticas expresan comportamientos maliciosos, y vienen representadas como un grafo cuyos nodos están etiquetados con acciones determinadas. El proceso de detección pasará por localizar el grafo de la firma dentro del extraído del código analizado. Este proceso es conocido comúnmente como el problema del isomorfismo en grafos, y que consiste en localizar un subgrafo dentro de un grafo mayor.

4. Mecanismos de evasión

Sin duda alguna, el software especializado en la detección de código malicioso ha sido una de las vías más empleadas contra la lucha del *malware*. Si se es capaz de detectar un código malicioso a tiempo, es posible evitar la infección. Cuando en los inicios los autores de *malware* vieron que la proliferación de sus creaciones era contrarrestada mediante este tipo de software, tuvieron que idear nuevas formas de supervivencia. Estas estrategias se centran en conseguir precisamente que el software malicioso permanezca oculto a los motores de detección. Cuanto más tiempo un código malintencionado pase desapercibido, más tiempo dispondrá para propagarse, y la infección afectará a más sistemas. Adicionalmente, estos mecanismos de evasión pretenden hacer más complejo su análisis, lo que requiere de un mayor esfuerzo por parte de los expertos y, por consiguiente, más tiempo para la generación de las firmas.

Debido al amplio uso de las estrategias de evasión por parte del *malware*, en la actualidad nos vemos en la necesidad de entender y analizar cuáles son los métodos empleados. Por tanto, presentamos en este apartado las técnicas más comunes utilizadas por el software malicioso con la finalidad de no ser detectado.

Enlace de interés

En la web *VX Heavens* (http://vxheaven.org/) se puede encontrar una gran cantidad de documentación sobre *malware* y de cómo este es programado. También incluye software malicioso que puede resultar de interés para su análisis y comprender mejor los mecanismos de evasión.

Podemos distinguir básicamente tres tipos de tecnologías de evasión:

1) las técnicas de *ofuscación*,

2) los métodos de *ocultación y autoprotección* y

3) los mecanismos *antidebugging*.

A pesar de la categorización que aquí exponemos, es importante remarcar que estas no son excluyentes. Es decir, podemos –y de hecho es muy habitual– encontrar software malicioso que combine varias técnicas de las tres categorías de forma simultánea. A continuación se describen cada una de las citadas categorías y los diversos métodos que podemos encontrar en cada una de estas.

4.1. Técnicas de ofuscación

Las técnicas de ofuscación pueden verse como una transformación del código de un programa con el objetivo de hacer su comprensión más difícil, mientras que al mismo tiempo se preserva su funcionalidad.

Estas técnicas han sido aplicadas tanto a la protección del software en general como a la del *malware*. Desde la perspectiva de la protección del software, la ofuscación del código permite proteger los programas de ataques que atenten contra la propiedad intelectual. En concreto, estas técnicas dificultan aplicar ingeniería inversa, lo que dificulta dichos ataques. Por otro lado, desde el punto de vista del *malware*, la finalidad de los mecanismos de ofuscación es doble. En primer lugar, incrementar la dificultad en el proceso de análisis así como el tiempo necesario para realizar este y, como consecuencia, retrasar la generación de las firmas. En segundo lugar, las estrategias de ofuscación tienen como finalidad la de mutar el código y, por tanto, dificultar o incluso hacer inviable el uso de técnicas de detección basadas en firmas sintácticas.

A continuación presentamos las técnicas de ofuscación más comunes que podemos encontrar en el software malicioso, es decir, el *malware* cifrado, el oligomorfismo, el polimorfismo, el metamorfismo, la compresión de ejecutables, el *entry point obscuring* y la ofuscación por virtualización.

4.1.1. *Malware* cifrado, oligomorfismo, polimorfismo y metamorfismo

Estas técnicas de ofuscación se basan en mutar el código del *malware* en cada nueva infección desde un punto de vista sintáctico, pero manteniendo la misma funcionalidad que la versión previa. A partir de esto, se desprende de forma natural que estas técnicas están centradas sobre todo en evadir los motores de detección sintácticos. Sin embargo, como se verá, estas técnicas también dificultan el proceso de análisis manual que pueda realizar cualquier experto en seguridad.

Dentro de esta categoría encontramos distintas posibilidades, en particular el *malware* cifrado, el oligomorfismo, el polimorfismo y el metamorfismo. Cada una de estas técnicas es una evolución respecto a la anterior, siendo el *malware* cifrado la primera en aparecer cronológicamente y, por ende, la más simple, mientras que el metamorfismo se corresponde a la evolución más sofisticada de los cuatro métodos de ofuscación aquí tratados. Vamos a verlas con más detalle:

1) *Malware* cifrado. Este primer sistema evade los métodos de detección sintácticos mediante el uso de una función de cifrado. Concretamente, el código malicioso está compuesto por una rutina de descifrado, una clave, y el cuerpo principal del *malware* cifrado. Cuando se ejecuta el código malintencionado, la rutina de descifrado es quien primero toma el control, descifrando el resto del cuerpo con la clave contenida en el mismo código. Tras el descifrado, el cuerpo principal es ejecutado. En cada nueva infección, el código malicioso determina una clave aleatoria que usa para construir una nueva variante. Esta nueva versión estará compuesta por la misma rutina de descifrado, la clave

Instrucciones para el cifrado/descifrado

El *malware* que emplea rutinas de descifrado como forma de ofuscación puede utilizar diversos métodos, como son el uso de las instrucciones invertibles `ADD`, `SUB`, `INC`, `DEC`, `XOR` o `NOT`, o combinaciones de las mismas.

obtenida aleatoriamente, y el cuerpo del *malware* cifrado con la nueva clave. De esta manera, el cuerpo del código malicioso es diferente cada vez, impidiendo determinar firmas sintácticas basadas en dicho cuerpo. Sin embargo, dado que la rutina de descifrado permanece constante a lo largo de todas las generaciones, es posible determinar firmas sintácticas asociadas a la porción de código responsable del descifrado.

2) Oligomorfismo. Con el objetivo de superar las deficiencias del *malware* cifrado, los creadores de software malicioso idearon una evolución de este que denominamos oligomorfismo. A diferencia del *malware* cifrado en el que la función de descifrado permanece constante, el oligomorfismo se basa en modificar la rutina de descifrado generación tras generación. Para esto, el código malintencionado crea dinámicamente una nueva rutina basada en porciones de código seleccionadas aleatoriamente entre un conjunto de alternativas. Como es lógico pensar, el cifrado del cuerpo se hace acorde a esta rutina construida. En este sentido, el oligomorfismo se considera el precursor de lo que se conocería posteriormente como polimorfismo. A pesar de la evolución que supone esta idea respecto al *malware* cifrado, el número de posibilidades en cuanto a rutinas de descifrado es limitado y, por tanto, utilizar una detección mediante firmas sintácticas sobre el código de descifrado aun sigue siendo una estrategia factible. Así, una posibilidad es crear firmas sintácticas para cada porción de código entre las diversas alternativas para, posteriormente, intentar detectar varias de estas firmas encadenadas y que compondrán la rutina de descifrado. Otra alternativa para la detección de software malicioso que incorpora este mecanismo es la emulación del código. Ésto permite el descifrado dinámico del cuerpo del *malware* para, *a posteriori*, localizar una firma sintáctica definida.

3) Polimorfismo. Como mejora al oligomorfismo desde la perspectiva de la evasión, el *malware* polimórfico apareció posteriormente. Este incorpora la capacidad de generar una gran cantidad –del orden de millones– de rutinas de descifrado diferentes. Para conseguir esto, el software malicioso polimórfico se apoya en el uso de diversos métodos de ofuscación, tales como la reordenación de rutinas o la inserción de código innecesario. Estos métodos, comunes también al metamorfismo, serán tratados más adelante. Como consecuencia de la gran cantidad de rutinas de descifrado que un software malicioso puede crear en cada nueva generación, no existe un patrón sintáctico que buscar. La detección de este tipo de código malicioso requiere de mecanismos más sofisticados como son el uso de emuladores, o la detección basada en comportamiento.

Lectura complementaria

La estrategia del cifrado/descifrado como mecanismo de ofuscación puede ser usada de diversas formas sofisticadas. Así, por ejemplo, es posible encontrar *malware* con varios niveles de cifrado anidados con sendas rutinas de descifrado. Os animamos a leer a Szor (2005) para ampliar esta información.

4) Metamorfismo. El *malware* metamórfico apareció como un método de ofuscación que iba más allá de las propuestas anteriores en cuanto a nivel de sofisticación. Al igual que el polimorfismo, el metamorfismo emplea diversas técnicas de ofuscación, sin embargo, en el caso que nos ocupa se aplican a la totalidad del cuerpo del *malware*, no a una rutina de descifrado. De esta manera, cada nueva versión creada para una infección conduce a una varian-

te totalmente diferente a la anterior sintácticamente hablando. Esto requiere que el software malicioso sea capaz de reconocer su propio cuerpo principal, analizarlo y mutarlo cuando se propague. En cierto modo, podemos decir que el *malware* metamórfico es pseudo-consciente de sí mismo.

Técnicas de ofuscación del polimorfismo y del metamorfismo

Como hemos comentado anteriormente, tanto el polimorfismo como el metamorfismo emplean técnicas de ofuscación específicas. En el caso del polimorfismo, estas estrategias se usan para mutar la rutina de descifrado, mientras que en el metamorfismo se usan para mutar la totalidad del cuerpo del código malintencionado. En ambos casos, las técnicas suelen combinarse de forma simultánea, consiguiendo así que su detección sintáctica sea más difícil. Con la finalidad de comprender mejor cómo actúan estos tipos de códigos maliciosos en el proceso de crear nuevas generaciones mutadas, vamos a introducir las técnicas de ofuscación principales que utilizan:

1) Inserción de código innecesario. Esta técnica inserta instrucciones (código basura) en el cuerpo principal del *malware* que no tienen ningún tipo de efecto en el comportamiento final del código. Esto permite cambiar la apariencia sintáctica del software malicioso a la vez que mantiene su funcionalidad.

2) Reasignación de registros. Este método de ofuscación se basa en que el mismo *malware* analiza su cuerpo principal y crea una nueva versión en la que se reasignan nuevos registros a las instrucciones que los emplean. Este proceso se realiza respetando la lógica del programa y manteniendo la consistencia con relación al uso de los registros por parte de las instrucciones. De esta manera, los *opcodes* de la nueva versión resultante difieren respecto a la anterior.

3) Sustitución de código por instrucciones equivalentes. Esta estrategia se fundamenta en crear una nueva generación reemplazando un conjunto de instrucciones de la versión anterior por otras totalmente equivalentes. Así, mientras que la lógica del programa se mantiene, su aspecto se ve modificado en comparación a la versión previa.

4) Permutación de subrutinas. Esta táctica ofusca el código original cambiando el orden en el que aparecen las subrutinas dentro del cuerpo del *malware*. De forma numérica, si un software malicioso está compuesto por n subrutinas diferentes, este será capaz de generar un total de $n!$ variantes.

5) Transposición de código. Este mecanismo muta el código al transponer bloques de instrucciones mientras se preserva el comportamiento original del *malware*. Existen dos formas de conseguir esto. La primera, se basa en mezclar

los bloques de instrucciones de forma aleatoria e introducir saltos incondicionales al final de cada bloque para mantener el orden de ejecución. La segunda forma consigue mutar el código al seleccionar y transponer instrucciones independientes cuyo orden de ejecución no afecta al comportamiento final.

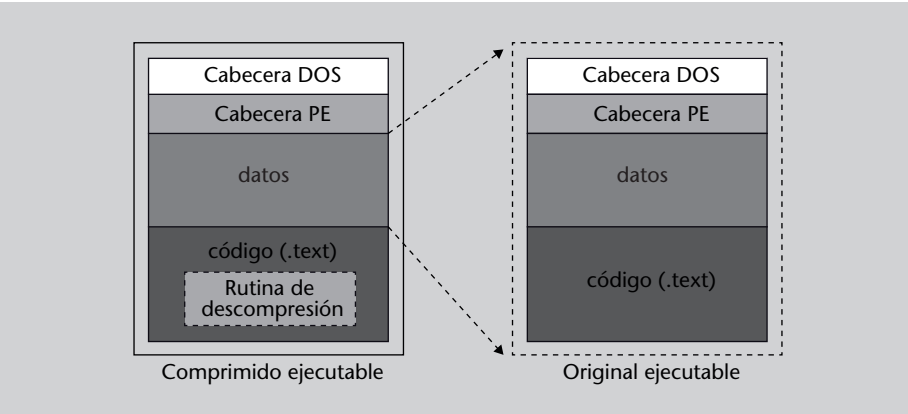
6) Integración de código entrelazado. Este modo de evasión es empleado por *malware* de propagación por infección vírica, el cual se inserta a sí mismo en el código del archivo a infectar de forma entrelazada. Para esto, es necesario que el software malicioso desensamble el ejecutable a infectar y lo represente en un formato manejable, se inserte entre el código del ejecutable, modifique el código original para mantener la consistencia a nivel de instrucciones y de referencia a datos y, por último, reconstruya el nuevo ejecutable infectado.

4.1.2. Compresión de ejecutables

El origen de la compresión de los ejecutables se remonta a los años ochenta, cuando la capacidad de los sistemas de almacenamiento de información era menor que en la actualidad y su coste más elevado. Por aquel entonces, y con el objetivo de obtener un mejor aprovechamiento del espacio, se utilizaron algoritmos de compresión sobre los ejecutables.

Si bien es cierto que esta idea puede continuar siendo aplicada en la actualidad de forma legítima, esta estrategia también puede ser utilizada por software malicioso como mecanismo de evasión. El motivo de esto radica en la siguiente idea. Si un código malicioso se modifica ligeramente y se libera como una nueva versión, es probable que comparta porciones de código respecto a su predecesor y que, por lo tanto, la existencia de una firma sintáctica que detectaba la generación previa probablemente también detecte la nueva. Sin embargo, si se utiliza un algoritmo de compresión sobre la nueva versión, aunque esta incorpore un simple cambio, resultará en un ejecutable radicalmente diferente. De esta manera, la firma sintáctica que detectaba la versión anterior no es útil para variantes futuras.

Figura 8. Representación conceptual de un *malware* empaquetado en Windows



Un ejecutable comprimido está estructurado en dos grandes partes. Por un lado, la rutina de descompresión y, por otro, los datos comprimidos que se corresponden con el código original del ejecutable. Cuando se lanza la ejecución de un programa comprimido, la rutina responsable de la descompresión se ejecuta en primera instancia. Entonces, esta, a partir de los datos comprimidos, regenerará el código del ejecutable original y le cede el control.

A pesar de que cada creador de *malware* podría implementar sus propios algoritmos de compresión, la realidad muestra que en la actualidad existen múltiples herramientas que automatizan dicho proceso, y que son empleadas asiduamente por los creadores de *malware*. Estas son conocidas comúnmente con el nombre de empaquetadores (o *packers* en inglés). Su forma de trabajar es la siguiente: dado un ejecutable indicado por el usuario, generan una nueva versión compuesta por el código original comprimido, así como la rutina necesaria para su descompresión. Algunas de estas herramientas incluso añaden técnicas *antidebugging* al ejecutable resultante, lo que dificulta su posterior análisis.

Packers

En Internet podéis localizar información sobre los *packers* más conocidos. Algunos de estos son UPX, Armadillo, ASPack, PE Compact o ASProtect entre otros.

4.1.3. *Entry point obscuring*

El *Entry Point Obscuring* (EPO) es un método de ofuscación propio del *malware* de infección. Tradicionalmente, la infección de un ejecutable siempre se ha realizado modificando su punto inicial de ejecución (o *entry point*), de manera que este apunte primero al código del *malware*. Una vez que el código malicioso toma el control y realiza las acciones de su interés, cede el control al código del ejecutable original. A diferencia de esto, la técnica de *entry point obscuring* modifica el ejecutable original en cualquier punto de su código, insertando instrucciones de tipo `CALL` o `JMP` para redirigir el flujo de ejecución hacia el código del *malware*. Haciendo esto, se consigue evadir motores de detección sintácticos que intenten localizar *malware* analizando el *entry point* como ámbito de búsqueda.

4.1.4. Ofuscación por virtualización

La ofuscación por virtualización es un mecanismo de ofuscación que implementa en el mismo código del *malware* un entorno de ejecución junto a un intérprete, el cual es capaz de ejecutar programas escritos en un lenguaje específico en forma de *bytecodes*. El intérprete –altamente ofuscado– acepta un lenguaje que se elige aleatoriamente en cada infección. Dicho de otro modo, podemos considerar que el software malicioso que incluye esta estrategia implementa un procesador virtual que acepta un repertorio de instrucciones particular.

Terminología

En la literatura, en referencia a la estrategia de ofuscación por virtualización también encontraremos la denominación de *ofuscación mediante máquinas virtuales* (Rolles, 2009).

El modo de operar de este tipo de *malware* se basa en seleccionar aleatoriamente, previa una nueva infección, un lenguaje aceptado por el intérprete, y recodificar el cuerpo del *malware* sobre la base de ese nuevo lenguaje. Evidentemente, el intérprete que contendrá la nueva variante se genera de acuerdo al nuevo lenguaje elegido. De esta manera, tanto el intérprete como el conjunto de *bytecodes* se modificarán sintácticamente, mientras que se preserva el comportamiento del código malicioso.

Realizar un análisis de este tipo de código malicioso para entender su funcionamiento pasa por comprender la arquitectura y el procesador virtual que implementa. Esto no siempre es trivial y puede llegar a convertirse en un trabajo tedioso para el analista. El uso de ingeniería inversa en esta situación se convierte en algo complejo al tener que abordar aspectos de alto nivel (por ejemplo, la propia arquitectura y el procesador virtual) que quedan ofuscados por los detalles de bajo nivel. Incluso después del esfuerzo en comprender el lenguaje y el intérprete, una nueva versión puede aparecer con una implementación diferente de un procesador virtual nuevo.

4.2. Técnicas de ocultación y autoprotección

Las técnicas de ocultación y autoprotección son mecanismos que implementa el *malware* con el objetivo de pasar desapercibido tanto a administradores como a software de detección, o bien para protegerse y dificultar su erradicación en un sistema infectado. Comúnmente, al conjunto de técnicas que permiten conseguir esto se las denomina *mecanismos rootkit*.

Lectura recomendada

Una buena referencia para entender cómo funcionan las técnicas de *rootkit* en las plataformas Windows es el siguiente libro:
G. Hoglund; J. Butler (2005). *Rootkits: Subverting the Windows Kernel*. Addison-Wesley

Para conseguir ocultarse o protegerse, el código malicioso modifica partes del sistema operativo. A pesar de que cada sistema operativo tiene sus particularidades y dichas modificaciones pueden realizarse de diversas maneras, de forma general podemos considerar que hay tres modos de conseguir la ocultación y/o autoprotección. Estos tres modos son los siguientes:

1) Mecanismos *rootkit* en espacio de usuario. Los mecanismos *rootkit* a nivel de aplicación atacan a las API* utilizadas por los programas. Estas API están implementadas como librerías dinámicas proporcionadas por el mismo sistema operativo, lo que permite compartir su código entre varias aplicaciones de forma simultánea. De esta manera, las aplicaciones tienen acceso a un conjunto de funciones que no deben implementar, ya que vienen proporcionadas por el mismo sistema. Evidentemente, este principio parte de la premisa de que las aplicaciones confían plenamente en las funciones de las librerías. En el contexto que nos ocupa, esto tiene especial sentido si tenemos en cuenta que algunas de estas funciones proporcionan métodos para la apertura de archivos, o la obtención del listado de procesos del sistema. Así, si un *malware* es capaz de interceptar la llamada de una función para obtener la lista de procesos activos, por ejemplo, podría asegurarse de que el resultado proporcionado

*** API es la abreviatura de *application program interface.***

no incorporase el proceso asociado a él mismo, lo que le permitiría pasar desapercibido a las aplicaciones que utilizasen dicha función.

Para entender cómo el *malware* puede conseguir esto, debemos comprender de qué manera funcionan las librerías dinámicas. Cuando se genera un ejecutable de un programa que utiliza librerías dinámicas, el compilador incluye en la imagen binaria resultante ciertas estructuras de datos. Estas estructuras de datos incorporan el nombre de las librerías junto a la lista de funciones que usa el programa. Asimismo, para cada elemento de la lista existe un apuntador no inicializado con la dirección de memoria de la función asociada. Cuando un programa se carga en memoria para su ejecución, el sistema operativo lee las estructuras de la imagen binaria. Seguidamente, con la información pertinente de las estructuras, carga las librerías en memoria, estima las direcciones de cada función por cada librería, y rellena los apuntadores de la lista con las direcciones de las funciones. De esta manera, cuando en el proceso correspondiente se realiza una llamada a una función de una librería, se sabe la dirección de memoria donde ésta se encuentra y, por tanto, hacia dónde se debe redirigir el flujo de ejecución. Con la modificación de uno de estos apuntadores, el software malicioso puede desviar el flujo de ejecución a su propio código, lo que le permite tener pleno control sobre la llamada y falsear los datos de respuesta. La modificación del apuntador de una función de una librería se puede realizar de diversos modos, y las técnicas dependen ya de cada sistema operativo.

2) Mecanismos *rootkit* en espacio de núcleo. Conceptualmente, los mecanismos de *rootkits* en espacio de núcleo no se alejan sustancialmente de los de espacio de usuario. En este sentido, este tipo de mecanismos también desvía la ejecución hacia código malicioso de su interés, el cual actúa en función de sus necesidades. La diferencia principal entre las dos categorías se basa en que en esta segunda, el proceso de desviar el flujo de ejecución se realiza con un mayor nivel de privilegios. Concretamente, el código malintencionado se carga y se ejecuta en el espacio de direcciones del núcleo del sistema operativo. Precisamente, esta característica de ejecución con privilegios de sistema les proporciona una mayor resistencia a ser detectados o eliminados, ya que las herramientas de detección o eliminación suelen ejecutarse en espacio de usuario. Esto implica una mayor complejidad en su código, lo que los hace menos comunes.

La forma más habitual de desviar el flujo de ejecución por este tipo de mecanismos se basa en modificar las tablas que contienen los apuntadores a las llamadas del núcleo (*syscall table*). Con el cambio de los apuntadores de las *syscalls*, el código malicioso tendrá pleno control sobre las llamadas que pueden suponer un riesgo para él. De este modo, el *malware* actuará en consecuencia cada vez que desde el espacio de usuario se realice una llamada interceptada.

Ejemplo

Así, si se intenta ejecutar la llamada al núcleo para matar el proceso del *malware*, y esta llamada está interceptada, el código malicioso puede actuar para que esto no suceda. Sin

embargo, si la llamada para matar un proceso se hace sobre uno diferente, el código malintencionado llamará a la *syscall* original. De esta manera, el sistema se comportará de forma normal para los casos que no atenten contra el software malicioso.

A pesar de ser la técnica de la modificación de la tabla de las *syscalls* la más común, existen otras alternativas, como es el uso de controladores para la ocultación de información o la autoprotección.

3) Mecanismos *rootkit* híbridos. Los mecanismos híbridos, tal y como indica su nombre, pueden ser aplicados tanto en espacio de usuario como en espacio de núcleo. Esta técnica, a diferencia de las dos anteriores en las que se reemplaza algún tipo de apuntador, ataca directamente a la función implicada. Para esto, modifica los primeros bytes de la función, reemplazándolos por un salto incondicional hacia el código del *malware*. De esta manera, se modifica el flujo de ejecución tomando el control el software malicioso cuando se produce una llamada a una función. Para preservar el comportamiento original de la llamada, y previo a la sobreescritura de los primeros bytes con el salto incondicional, el código malintencionado debe guardar los bytes reemplazados. Este mecanismo resulta aún más complejo de detectar si el *malware* introduce el salto incondicional en cualquier otro punto de la función diferente de sus primeros bytes.

4.3. Mecanismos *antidebugging*

Los mecanismos *antidebugging* son un conjunto de estrategias que el *malware* puede incorporar en su propio código, cuya misión es la de dificultar cualquier proceso de ingeniería inversa que se intente aplicar sobre él. Su objetivo principal es el de detectar si un *debugger* está supervisando la ejecución del propio código malicioso. Si este es el caso, el mismo *malware* modificará su comportamiento.

Lectura recomendada

Una buena recopilación de técnicas *antidebugging* para la plataforma Windows puede encontrarse en el artículo "Anti-Debugging. A Developers View" de Tyler Shields, y que está disponible en la siguiente dirección de Internet: http://www.veracode.com/images/pdf/whitepaper_antidebugging.pdf

En este sentido, puede adoptar varias posturas:

- ejecutar código complejo sin ninguna finalidad para desalentar al analista,
- exhibir un comportamiento legítimo en vez de malicioso, o
- finalizar su ejecución.

Las técnicas más recientes incluso emplean algún método para detectar si la ejecución se está realizando en una máquina virtual, dado que su uso es muy habitual en los procesos de análisis de software malintencionado. Es preciso destacar que la mayoría de los mecanismos empleados son altamente dependientes de los sistemas operativos y de las arquitecturas de los sistemas.

Resumen

En los últimos años, la industria del *malware* ha focalizado sus esfuerzos en explotar las deficiencias de seguridad como una posible vía de infección de los sistemas. De entre todas las posibles deficiencias, los desbordamientos de *buffers* son una clara amenaza al permitir a los atacantes la ejecución de código arbitrario. La causa de esto reside en los errores cometidos por los programadores al no controlar los límites de los tamaños de los *buffers*, lo que permite sobreescribir las direcciones de retorno almacenadas en la pila, y redirigir el flujo de ejecución hacía una secuencia de instrucciones especialmente preparada.

En el ámbito del *malware*, establecer una taxonomía genérica que permita clasificar el código malicioso sobre la base de ciertas características se hace complejo. Cada día se hace más patente que las nuevas formas de *malware* exhiben comportamientos que dan lugar a que puedan clasificarse según diversas categorías. Esto pone de manifiesto la constante evolución que está sufriendo el código malicioso, y la necesidad de tener que extremar las medidas de seguridad para proteger a los sistemas de información. En este sentido, el software de detección de *malware* es una herramienta proactiva que debe ser combinada junto a otras estrategias, como son las buenas prácticas llevadas a cabo por los usuarios, los diseños de software basados en patrones de protección, o el uso de las firmas digitales.

El software de detección de código malicioso se sustenta en dos grandes pilares para la localización de objetos dañinos. La forma más básica es la detección basada en la sintaxis del código, siendo esta estrategia un método fácilmente eludible por el código malicioso a través de técnicas de ofuscación. Por otro lado, la detección semántica se presenta como una forma de superar las deficiencias de la detección sintáctica, siendo incluso capaz de detectar *malware* desconocido. A pesar de la indiscutible utilidad del software de detección, es importante remarcar el hecho de que la detección perfecta no existe, tal y como ya postuló Cohen en sus trabajos en el campo de los virus informáticos. Como consecuencia, nuestros sistemas siempre deben ser considerados como potencialmente vulnerables a cualquier forma de código malicioso.

Prueba de que la detección perfecta no existe es la existencia de una gran cantidad de estrategias implementadas por el *malware*. Así, las técnicas de ofuscación, los mecanismos *rootkit*, o los métodos *antidebugging*, son un claro ejemplo de estas tácticas. Sin duda alguna, todas estas técnicas están pensadas para eludir los motores de detección, o como medida de protección para evi-

tar su erradicación. De esta manera, el código malicioso es capaz de perdurar sigilosamente en los sistemas. Este hecho debería hacernos reflexionar acerca de la importancia que tiene el *malware* como amenaza, siempre latente, en el campo de la seguridad informática.

Actividades

1. Descargad el software de compresión para ejecutables UPX (*Ultimate Packer for eXecutables*) que encontraréis en la siguiente dirección web: http://upx.sf.net/. Copiad un ejecutable cualquiera en un directorio y comprimidlo con dicho software. Observad la diferencia de tamaño. Utilizad un editor hexadecimal (por ejemplo, HT Editor*) para ver las diferencias entre los dos ejecutables de nivel de contenido binario, y de las cabeceras PE.

* **http://hte.sf.net/**

2. Analizad algún antivirus y buscad si implementa algún tipo de heurística ¿Viene activada por defecto? ¿Qué tipo de relación existe entre las heurísticas y los falsos positivos?

3. Leed el documento *Creating signatures for ClamAV*, que podréis localizar en la siguiente dirección web: http://www.clamav.net/doc/latest/signatures.pdf. En este se describe la forma de especificar firmas sintácticas para la detección de *malware* en el antivirus *ClamAV* ¿Qué tipo de firmas permite? ¿Tiene la posibilidad de indicar ámbitos de búsqueda asociados a las firmas? ¿Qué tipos de ámbitos soporta?

4. Buscad a través de Internet la descripción de algún *malware* específico que haya explotado algún desbordamiento de *buffer*. Clasificad el *malware* localizado según la taxonomía que hemos presentado.

5. Razonad una lista de buenas prácticas que todo usuario debería seguir para reducir el riesgo de infección de los sistemas con software malicioso ¿Creéis que la gente de vuestro entorno sigue estas buenas prácticas en la totalidad?

6. Localizad en el portal http://www.exploit-db.com/ diversos *shellcodes* identificando qué acción permiten hacer y para qué plataforma son válidos. Verificad que ninguno de ellos contiene un *opcode* con el valor `0x00`.

7. En la misma web de la actividad anterior, buscad un *exploit* que permita la ejecución remota de código. Identificad cuál es el software que se ve afectado por el *exploit*, e intentad comprender dónde se produce el desbordamiento de *buffer*.

Ejercicios de autoevaluación

1. La ejecución de código arbitrario en una explotación de un desbordamiento de *buffer* se produce en la fase de...
a) prólogo.
b) llamada.
c) retorno.
d) Todas las anteriores.

2. Un *shellcode* utilizado en un desbordamiento de un *buffer*...
a) suele venir precedido de instrucciones `NOP` para maximizar el éxito del ataque.
b) tiene un tamaño limitado cuando se trata de un *stack overflow*.
c) tiene una fuerte dependencia con la arquitectura y el sistema operativo de la plataforma atacada.
d) Todas las anteriores.

3. La ejecución de código gracias a un desbordamiento de un *buffer* es posible ya que...
a) se sobreescribe la dirección de retorno de alguna función.
b) existen lenguajes de programación que operan con *buffers* sin controlar sus tamaños.
c) se pueden crear *shellcodes* que no contengan ningún *opcode* con valor `0x00`.
d) Todas las anteriores.

4. ¿Cuál de las siguientes afirmaciones es falsa respecto a la detección de *malware* empleando firmas sintácticas?
a) Las firmas pueden incluir expresiones regulares.
b) Las firmas pueden ser definidas en base a funciones *hash* criptográficas.
c) Las firmas son completamente inmunes al polimorfismo.
d) Ninguna de las anteriores.

5. La detección perfecta de software malicioso...
a) es un problema indecidible.
b) siempre es posible con un motor de detección basado en análisis semántico.
c) siempre es posible con un motor de detección basado en firmas sintácticas.
d) Ninguna de las anteriores.

6. El empaquetado empleado por el *malware*...
a) es una técnica *antidebugging*.
b) es una forma de evasión contra la detección semántica.
c) Las firmas sintácticas son inmunes al *malware* empaquetado.
d) Ninguna de las anteriores.

7. ¿Qué afirmación es cierta respecto al software malicioso de propagación como gusano?
a) Puede utilizar la ingeniería social en un correo para propagarse.
b) Puede utilizar un desbordamiento de *buffer* para propagarse.
c) Puede implementar técnicas *rootkit*.
d) Todas las anteriores.

8. Los ámbitos de búsqueda utilizados en las firmas sintácticas...
a) aceleran el proceso de detección en términos generales.
b) penalizan en rendimiento en la identificación de código malicioso.
c) en ocasiones pueden ser evadidos mediante *entry point obscuring*.
d) *a* y *c*

9. La reasignación de registros es una técnica de ofuscación propia de...
a) el *malware* cifrado.
b) el *malware* oligomórfico.
c) el *malware* polimórfico.
d) Ninguna de las anteriores.

10. ¿Qué afirmación es cierta con relación a la detección basada en comportamiento?
a) No requiere firmas.
b) La base de datos tiene un nivel de crecimiento inferior en comparación a la de las firmas sintácticas.
c) Son muy eficientes en la erradicación del *malware* dado su nivel de identificación precisa.
d) Ninguna de las anteriores.

Solucionario

Ejercicios de autoevaluación

1. c; **2.** d; **3.** d; **4.** c; **5.** a; **6.** d; **7.** d; **8.** d; **9.** c; **10.** b;

Bibliografía

Cohen, F. B. (1984). *Computer viruses: Theory and experiments*. Los Angeles (EE. UU.): University of Southern California.

Cohen, F. B. (1986). *Computer viruses*. Tesis Doctoral, University of Southern California, Los Angeles (EE. UU.).

Cohen, F. B. (1987). «Computer viruses: Theory and experiments». *Computers & Security*, volumen 6 (n.º 1, págs. 22–35).

Embleton, S.; Sparks, S.; Zou, C. (2008). «SMM Rootkits: a New Nreed of OS Independent Malware». En: «Proceedings of the 4th international conference on Security and privacy in communication netowrks», SecureComm '08, (págs. 11:1–11:12). New York (EE. UU.): ACM.

Filiol, E. (2007). «Metamorphism, Formal Grammars and Undecidable Code Mutation». *International Journal of Computer Science*, (n.º 2, págs. 70–75).

Foster, J. C.; Osipov, V.; Bhalla, N. (2005). *Buffer Overflow Attacks: Detect, Exploit, Prevent*. (1.ª ed.). Syngress.

Jacob, G.; Debar, H.; Filiol, E. (2008). «Behavioral detection of malware: from a survey towards an established taxonomy». *Journal in Computer Virology*, (n.º 4, págs. 251–266).

King, S. T.; Chen, P. M.; Wang, Y. M.; Verbowski, C.; Wang, H. J.; Lorch, J. R. (2006). «SubVirt: Implementing malware with virtual machines». En: «SP'06: Proceedings of the 2006 IEEE Symposium on Security and Privacy», (págs. 314–327). Washington (EE. UU): IEEE Computer Society.

Koziol, J.; Litchfield, D.; Aitel, D.; Anley, C.; Eren, S.; Mehta, N.; Hassell, R. (2004). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley.

Kramer, S.; Bradfield, J. (2010). «A General Definition of Malware». *Journal in Computer Virology*, (n.º 6, págs. 105–114). Disponible online: http://dx.doi.org/10.1007/s11416-009-0137-1.

Rolles, R. (2009). «Unpacking Virtualization Obfuscators». En: «Proceedings of the 3rd USENIX conference on Offensive technologies», WOOT'09. Berkeley (EE. UU.): USENIX Association.

Spinellis, D. (2003). «Reliable Identification of Bounded-length Viruses is NP-complete». *IEEE Transactions on Information Theory*, volumen 49 (n.º 1, págs. 280–284).

Szor, P. (2005). *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.