[Puertas de llamada y VXDCalls]

"Puertas de llamada y VXDCalls , o como poner el sistema a tus pies"

Por Mr. Silver / WKT!

"Antes de seguir debo decir que la información que he encontrado en la red sobre estos temas a sido bastante pobre en lo que a explicaciones se refiere, tras un trabajo de investigación y de atar unas cosas con otras presento el resultado de mi investigación que puede no ser correcta del todo y espero que si alguien encuentra algún error me lo comunique. La información que aquí se brinda puede resultar extremamente peligrosa si se hace un uso indebido de ella. Las siguientes técnicas pueden aplicarse para la construcción de virus y programas que pueden destruir el sistema sin que el usuario ni tan siquiera reciba ningún mensaje de error. Pero tambien pueden ser utilizadas para crear herramientas que nos sean de gran utilidad en el noble arte del cracking, como volcadores de memoria, API hookers, Monitoreadores, y parcheadores de procesos..."

Puertas de llamada, para conseguir el nivel de privilegio máximo sobre el S.O. Un momento que estas diciendo? y que es eso de privilegios y puertas?. Pues los niveles de privilegio de una aplicación son como una especie de autorización que el S.O concede a cada tarea para que trabaje. En principio un PC (hablo de un 386 o superior) puede tener 4 niveles de privilegio y a mayor privilegio menos nivel, es decir, el nivel máximo de privilegio que una aplicación puede tener es el nivel 0 (o Ring 0 en inglés), y el mínimo el nivel 3 (Ring 3). Si no me equivoco Windows no utiliza los 4 niveles de privilegio sino que solo usa el 0 y el 3, siendo estos el de supervisor y el de usuario respectivamente. En nivel 0 (Ring 0) se ejecuta el código del S.O y normalmente el código que utilizan los controladores de dispositivos (drivers), conocidos estos tambien por tener la extension VXD, por último el resto de aplicaciones se ejecutan en nivel 3.

Una aplicación en nivel 3 (Ring 3) esta muy limitada en cuanto a acceso directo al hardware se refiere, las instucciones de E/S (IN OUT y otras como CLI STI) son exclusivas del nivel 0 (Ring 0) y una aplicación en nivel 3 no puede (teoricamente jejejejee....) acceder directamente a la memoria usada por otra aplicación, o por poner un ejemplo no puede acceder a una posición de memoria que este fuera de la memoría asignada a esa tarea. Para nosotros esto significa la imposibilidad (teórica) de poder alterar el código del programa en caliente, interceptar las llamadas al API y monitorizarlas con una función propia, y más imposible es aún alterar el propio funcionamiento del S.O. Todo esto antiguamente no existia ya que el DOS, no distinguia entre privilegios ni mariconadas de protección de memoria, si uno queria machar la memória donde se encontraba el código del COMMAND.COM era libre de hacerlo (recuerdo cuantas veces se me ha colgado el DOS por esto). Algunas de las cosas que he explicado anteriormente si son posibles de realizar mediante el uso de algunas funciones del API de Windows, pero seguimos estando limitados en muchos aspectos.

La solución a nuestro problema se llama puertas de llamada (call gates), he de decir que conocia la existencia de este método pero no sabía bien bien de que iba hasta que me tope con él mientras analizaba un sistema de protección comercial que no viene al caso. La aplicación en cuestión utilizaba el método de comparación de vectores de la INT 1 y la INT 3 , descrito en la lección 2. Y que puede ser facilmente burlado con la utilidad Bang!. Aún así la aplicación se me resistía a funcionar con mi estimado SoftIce en marcha, asi que me decidí a trazar la aplicación paso a paso analizando los posibles comportamientos anti-debugging que pudiera encontrar. Tras una breve sesion de trazado (que aburrido es trazar código!), me encontré con algo que me hizo sospechar bastante. Primero encontré una llamada a la función VMM Get_DDB (Get Device Description Block) seguida de Test_Debug_Intalled, estas dos funciones forman parte del VXD VMM (Virtual Memory Manager) , son privilegiadas y solo pueden ser ejecutadas por código que corra en un nivel de privilegio igual a 0, además se suelen usar para detectar el SoftIce (ver lección 2). Entonces si las aplicaciones se ejecutan en nivel 3, ¿Cómo es posible que se pudieran estar ejecutando estas dos funciones privilegiadas ?. Pues de alguna manera la aplicación debía de poder entrar en Ring 0, y para ello debía de utilizar algún método. Primero pensé que era posible que la aplicación estuviera utilizando un método como el que describo en la lección 2, este método se basa en reemplazar un vector de interrupción de la IDT (Interrupt Descriptor Table), por la dirección del código que queremos que se ejecuta al llamar a esta interrupción, pero tras algunas comprobaciones deseche esa teoría. Porqué? ¿Pues entonces como leches se lo montán para sa?.

Pues me puse a trazar a lo retro (de adelante hacia atrás usando la pila que para eso está) y tras encontrar la parte de código que hacia entrar a la aplicación en Ring 0, me lleve una grata sorpresa, esto fue lo que encontré:

```
pushad                  // guarda el contenido de los registros
push    ebx             // y empuja EBX a la pila, esto lo hace así para no tener que guarda el resultado de la
                        siguiente instrucción en un buffer a parte
sgdt    fword ptr [esp-2] // lee el registro de la Tabla Global de Descriptores (GDT) de la tarea actual en la pila
pop     ebx             // carga base de la GDT desde la pila
xor     eax,eax         // borra EAX
sldt    eax             // y carga el registro de la Tabla Local de Descriptores en AX (Local Descriptor Table
                        Register LDTR)
and     al,f8           // borra nibble alto y ultimo bit del nibble bajo, esto elimina los bits de privilegio RPL y TI del
                        selector y deja el índice a la GDT en AX
add     eax,ebx         // suma la base de la GDT al índice
mov     ch,byte ptr
        [eax+7]         // usa resultado como índice para llenar CH
mov     cl,byte ptr
        [eax+4]         // y CL
shl     ecx,10          // desplaza ECX 16bits a la izquierda, quedando el valor anterior de CX en la parte alta de
                        ECX y CX a 0
mov     cx,word ptr
        [eax+2]         // lee resto de la base de la LDT en el word bajo de ECX
lea     edi,[ecx+8]     // suma a la base de la LDT 8, esto lo hace para modificar el segundo descriptor de
                        segmento de la LDT tal y como sigue
cld                     // borra flag de dirección (por seguridad,para que se copia hacia adelante)
mov     eax,esi         // copia dirección del código a ejecutar en Ring 0
```

```
stosw                   // y guarda la parte baja de esa dirección, en el campo límite del descriptor
mov     eax,ec000028    // establece los atributos en 0xEC00 y el selector en 0x28
stosd                   // lo guarda
shld    eax,esi,10      // desplaza parta baja de eax a la parte alta y llena la parte baja con los bits mas
                        significativos de ESI
stosw                   // y guarda parta baja de EAX,esto pertenece a la parte alta de la dirección a la que saltar
                        en Ring 0 + los atributos + los derechos de acceso
popad                   // restaura los registros
                        // y salta a la dirección de código a ejecutar en Ring 0, el selector es 0xF, ya que se
call    000f:00000000   modificó el segundo selector de la LDT, para folver de esta llamada se debe usar la
                        instrucción RETF ya que es una llamada lejana la que se realiza, en la rutina tambien deben
                        salvarse los selectores actuales de la CPU para ello usamos PUSHAD y POPAD
ret
```

Explico brevemente lo que hace este código. Se obtiene el registro GDTR mediante la instrucción SGDT para la tarea actual, se obtiene el registro LDTR mediante la instrucción sldt, se enmascara para obtener el índice a la GDT y se calcula la dirección del descriptor de segmento en EDI. Se modifican los datos del descriptor para que apunten a la dirección de código almacenada en ESI (esta dirección es una dirección dentro de la aplicación en Ring 3, que nos servirá de puerta de entrada al código en Ring 0), y también se modifican sus atributos y derechos de acceso. Por último se llama a la puerta mediante CALL 000F:00000000, si entramos dentro de la llamada el código siguiente se estará ejecutando en Ring 0 ;), y tal y como yo esperaba és el que se encarga de llamar a Get_DDB y Test_Debug_Installed.

Veamos ahora lo que son las tablas de descriptores globales y locales.

"Al entrar en modo protegido, deben estar residiendo en memoria principal las tablas de descriptores, que contienen las referencias precisas para los segmentos que va a usar el procesador. Un sistema multitarea se compone de un área global, en la que residen todos los objetos (segmentos) comunes a todas las tareas, y un área local para cada tarea, con los segmentos propios de cada una. Cada segmento del área global está definido por un descriptor, existiendo una tabla llamada TABLA DE DESCRIPTORES GLOBALES o TABLA GLOBAL DE DESCRIPTORES (GDT), que contiene todos los descriptores del área global. Asimismo existe una tabla para cada tarea, que recoge todos los descriptores de los segmentos de cada una de ellas. Se trata de las TABLAS DE DESCRIPTORES LOCALES (LDT) o TABLAS LOCALES DE DESCRIPTORES. Existirán tantas LDT como tareas soporte el sistema. En un momento determinado el 386+ estará ejecutando una tarea concreta y tendrá activas la GDT y la LDT correspondiente a la tarea en curso. Dos registros internos de la CPU, manejados por el programador de sistemas, apuntan a la base de la GDT y a la base de la LDT activa, denominanándose GDTR y LDTR, respectivamente. La GDT y la LDT actúan como segmentos del sistema y sólo son accesibles por el sistema de explotación. La estructura interna de una tabla de descriptores se muestra en la siguiente tabla y puede contener un máximo de 8K descriptores de ocho bytes cada uno. La LDT es una tabla local propia de la tarea en curso y una conmutación de tarea provocará automáticamente el cambio de la LDT a traves de la modificación del valor en el registro LDTR."

+N x 8	DESCRIPTOR N
...	...
+16	DESCRIPTOR 2
+8	DESCRIPTOR 1
0	DESCRIPTOR 0

Texto extraido del libro: Microprocesadores avanzados 386 y 486 Introducción al Pentium y Pentium Pro - Edit. Paraninfo.

Notese que los registros GDTR y LDTR apuntan al descriptor 0 de sus respectivas tablas.

Y por último la estructura de los registros GDTR y LDTR.

```
// estructura para GDTR
typedef struct
{

        word limit; // tamaño de la tabla
        dword base; // dirección base del descriptor 0


}FPWORD;

// estructura para el registro LDTR

        word selector; // 16 bits que actuan como un selector de un descriptor de la GDT.
```

Un selector se define de la siguiente manera:

(15-3)	(2)	(1-0)
INDÍCE	TI	RPL

Donde:

- INDÍCE: Apunta a una de las entradas a la tabla de descriptores seleccionada con TI.
- TI (Table Indicator) : Este bit es el indicador de tabla, cuando TI = 1, se selecciona la LDTn, mientras

que, si TI = 0, se hace referencia a la GDT.
- RPL: Este campo es el valor del nivel de privilegio del segmento, es decir, es el nivel de privilegio del peticionario (PL). Puede ser 0 (00 en binario) o 3 (11 en binario).

Comprobemos ahora el valor del segmento del código anterior. Se establece el selector a 0x0028, que en binario es 101000, vemos pues que los bits 1 a 0 son 0b (petición para un nivel de privilegio 0), y el bit TI es 0b tambien, por lo que estamos haciendo referencia a la GDT. Por último los bits 15 a 3 valen 101b que en decimal es 5, por lo tanto se hace referencia al 5 descriptor de la GDT (la dirección de este descriptor seria (GDTR BASE + (5 * 8)).

Según los libros que he podido consultar un descriptor de segmento es una estructura de datos formada por 8 bytes, estos bytes contienen los parámetros que definen completamente el segmento referenciado, es decir: la base, el límite, los derechos de acceso y sus atributos.

BITS	31 -24	23-20				19-16	15-7					7-0
DIRECCIÓN N+4	BASE bits (31-24)	G 1 bit	D/B 1 bit	1 bit	AVL 1 bit	LIMITE (bits 19-16)	P 1 bit	DPL 2 bits	S 1 bit	TIPO 3 bits	A 1 bit	BASE (23-16)
DIRECCIÓN N	BASE (15-0)						LIMITE (15-0)					

Segun esto podriamos definir la estructura de un descriptor como lo siguiente:

```
// estructura de un descriptor
typedef struct {

    WORD limit_low;
    WORD base_low;
    BYTE base_m;
    BYTE access;
    BYTE limit_high;
    BYTE base_high;

}Descriptor;
```

Pero por alguna razón que desconozco, a la hora de la verdad (en el código) se usa la siguiente estructura

```
//compuertas del 386+
typedef struct {

    WORD offs_low; // word alto con la dirección de la memoria a la que apunta el
    descriptor/compuerta
    WORD selector; // selector
    WORD attributes; // atributos
    WORD offs_high; /// word bajo con la dirección de la memoria a la que apunta el
    descriptor/compuerta

}Compuerta;
```

El tamaño es el mismo pero no así el asignamiento de bytes de cada campo del descriptor/compuerta.

- BASE: Campo de 32bits que contiene la dirección lineal donde comienza el segmento.

- LÍMITE: Campo de 20 bits que expresa el tamaño del segmento. Ya que con 20bits el tamaño máximo es de 1MB, hay otro bit complementario en el campo de atributos, llamado de granularidad., G, que indica si el límite está expresado en bytes (G=0) o en páginas (G=1). Si fuera en páginas el tamaño máximo del segmento sería de 1M x 4Kb = 4GB.

- En verde los atributos: Es un campo de 4bits de los cuales uno de ellos debe estar a 0 para mantener la compatibilidad con los procesadores superiores como los 486 y los Pentium.

  - G-> GRANULARIDAD: Los 20 bits del campo límite del descriptor indican el tamaño del segmento, que estará expresado en bytes si G = 0, y en páginas si G = 1.

  - D/B -> DEFECTO/GRANDE: En los segmentos de código el bit D (Defecto) y en los segmentos de datos este mismo bit llamado B (Grande), permite distinguir los segmentos nativos de 32bits para el 386+, de los que pertenecen al 286. Así se mantiene una compatibilidad total con el software creado para el 80286, sin penalizar las instrucciones que aporta el 386+.

  - AVL-> DISPONIBLE: Este bit esta a disposición del usuario para poder diferenciar ciertos segmentos que contengan un tipo determinado de información o que cubran alguna función específica.

- En rojo los derechos de acceso:

  - A-> ACCEDIDO: Este bit se pone a 1 cada vez que el procesador accede al segmento.

  - P -> BIT DE PRESENCIA: Indica si el segmento al que referencia el descriptor está cargado, o sea, se halla presente en la memória principal (P=1), o bien, está ausente (P=0).

- DPL -> NIVEL DE PRIVILEGIO: Indica el nivel de privilegio del segmento al que se referencia el descriptor. Su valor puede variar entre el 0 y el 3 y consta de dos bits.

- S -> TIPO DE SEGMENTO: Si S=1, el segmento correspondiente al selector es "normal", o sea, un segmento de código, de datos o de pila. Si S=0, se refiere a un segmento del sistema, que referencia a un recurso especial del sistema, como puede ser una puerta de llamada (este interesa ;)), un segmento TSS, etc.

- TIPO: Los tres bits de este campo distinguen en los segmentos normales si se trata de uno de código, da datos o de pila. Además determinan el acceso permitido: lectura/escritura/ejecución.

TIPO		
E	C	R

Si el bit E del campo tipo es 1, TIPO se define como:

TIPO		
1	C	R

Donde:

- C -> AJUSTABLE: Si C = 0, al ser accedido el segmento no cambia su nivel de privilegio. si C = 1, se llama segmento ajustable, porque, cuando se accede a él, su nivel de privilegio toma el valor del que tiene el segmento que lo ha pedido.

- R -> LEÍBLE: Si R=1 el segmento de código se puede leer.En ningún caso se puede escribir un segmento de código.

Cuando E = 0 y se hace referencia a un segmento de datos, los otros dos bits de TIPO tiene el siguiente significado

TIPO		
0	ED	W

Donde:

- ED-> EXPANSIÓN DECRECIENTE: Si ED = 0, se trata de un segmento de datos normal, lo que supone que el crecimiento del mismo se realiza incrementando el valor de la dirección. Cuando ED = 1, se trata de un segmento de pila pues su crecimiento se efectua decrementando el valor de la dirección que apunta a su cima.

- W -> ESCRIBIBLE: Si W=1 el segmento de datos se puede leer y escribir, mientras que, si W = 0, sólo se puede leer.

Tal y como podemos ver en la estructura el parámetro más interesante es el de los atributos (en rojo), en el programa en cuestión se establecen los privilegios a 0xEC00, veamos que significa esto.

0xEC00 en binario -> 1110110000000000 , si comprobamos cada campo de los atributos.

P (bit 15) -> Está a 1, por lo tanto el segmento al que hace referencia el descriptor está cargado, esto es lógico ya que el código que ejecutará la aplicación se encuentra en un segmento ya cargado, el de la aplicación en Ring 3.

DPL (bits 14-13) -> Indica el nivel de privilegio, vale 11, que es 3, por lo tanto Ring3.

S (bit 12) -> Esta a 0, el segmento correspondiente al selector es un segmento de sistema que pertenece a un recurso especial como una puerta de llamada, correcto, es lo que buscabamos.

TIPO (bits 11- 9) -> Bit 11 a 1, indica que se hace referencia a un segmento de código, los bits 10 y 9 indican que el segmento es ajustable (adoptará el nivel de privilegio del segmento que lo solicitó, que tal y como vimos era 0x0028 y tenia un RPL de 0), y que no es legible respectivamente.

Visto el funcionamiento de los selectores, descriptores, GDT y LDT, podemos comprender el funcionamiento del código usado por el programa para entrar en Ring0. Otro punto a tener en cuenta es que deberiamos hacer una copia de el descriptor antiguo antes de modificarlo para luego dejarlo tal y como estaba todo, he realizado diversas pruebas y si no restauraramos el descriptor no sucede nada ya que solo se salta a la dirección que marca el descriptor cuando se accede directamente a ella, pero es de buena educación dejar las cosas como estaban. Por ultimo aquí teneis código fuente en C que muestra como saltar a Ring0, y llamar a la función del VXD VMM Test_Debug_Installed, y si se encuentra debugger se cuelga la máquina intencionadamente lllamando a la INT 19.

La INT 19, reinicia el sistema sin limpiar la memoria y sin restaurar los vectores de interrupción. Debido a que los vectores son preservados, esta interrupción causará un cuelgue del sistema si cualquier programa captura alguno de los vectores entre el 00 y el 1Ch, particularmente la INT 8. Este cuelgue sucedera siempre bajo Windows ya que estos vectores estan modificados por Windows.

Desafortunadamente esta técnica no funciona bajo Windows NT y el sistema generará un error de aplicación, de todas maneras yo lo he probado bajo Windows 98 y Windows Me, y funciona de maravilla, con lo que deduzco que probablemente funcionará con Windows 95.

Puedes bajarte el fichero ya compilado desde aquí -> testdebug.zip.

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

char caption[]= {"Test Debug by Mr. Silver / WKT!"};

// alinea las estructuras a BYTE
#pragma pack(1)
// estructura para la base de la GDT
typedef struct
{

    WORD limit;
    DWORD base;

}FPWORD;

// estructura de un descriptor
typedef struct {

    WORD limit_low;
    WORD base_low;
    BYTE base_m;
    BYTE access;
    BYTE limit_high;
    BYTE base_high;

}Descriptor;

// Estructura de un salto lejano, para las CALLGATES
typedef struct {

    DWORD offset32;
    WORD seg;

}FARJMP;


//compuertas del 386+
typedef struct {


    WORD offs_low;
    WORD selector;
    WORD attrib;
    WORD offs_high;


}Compuerta;

__declspec( naked ) void MyProc()
{
_asm {

    xor eax,eax
    int 0x20 // Esto ejecuta la llamada VXD Test_Debug_Installed, las VxdCalls se
    utilizan a traves
    _emit 0xc1 // de la INT 20, tras la interrupción siguen dos WORDS, el primer
    identifica el número
    _emit 0x00 // de servicio y el siguiente el identificador VXD, en este caso
    se usa el servicio
    _emit 0x01 // 0x00C1 del VXD VMM, para obtener una descripción de los
    servicio y sus identificadores
    _emit 0x00 // mirate la lista de interrupciones de RalfBrown.
    jz NoDebug // hay debugger ?
    int 0x19 // si, colgamos la máquina a saco ;)
    NoDebug:
    retf // no, salimos

}
}
__declspec( naked ) void TestDebugVXD(void * addr)
{

DWORD GateAddr; // dirección de la puerta
Compuerta OldGate; // estrutura donde copiar valores actuales
FPWORD GDT;
FARJMP CallGate;

    _asm
    {
```

```
            push ebp
            mov ebp, esp
            sub esp, __LOCAL_SIZE // con este definición el compilador calcula
            automáticamente el tamaño de pila necesario para las variables
            locales que usa la función

        }

    _asm {

        pushad // guarda el estado de los registros antes de ejecutar nada
        mov ebx,[addr]
        mov esi,ebx // copia dirección de la función a ejecutar en Ring 0
        sgdt fword ptr GDT
        mov ebx,GDT.base
        xor eax,eax // borra EAX
        sldt ax // y carga el registro de la Tabla Local de Descriptores en AX (LDT)
        and al,0xf8 // borra nibble alto y ultimo bit del nibble bajo
        add eax,ebx // suma la base de la GDT al valor calculado de la LDT
        mov ch,[eax+7] // usa resultado como índice para llenar CH
        mov cl,[eax+4] // y CL
        shl ecx,0x10 // lo desplaza 16bits a la izquierda
        mov cx,[eax+2] // carga parte baja de la base, con lo anterior se obtiene la
        base de la LDT en ECX
        lea edi,[ecx+8] // y carga en EDI la base calculada más 8
        mov dword ptr [GateAddr],edi // hace una copia de la dirección de la
        compuerta
        cld // borra flag de dirección para establecer
        mov cx,word ptr [edi] // copia antigua parte baja
        mov [OldGate.offs_low],cx // del desplazamiento de la compuerta
        mov eax,esi // guarda parte baja de la dirección de la función que se
        ejecutará en ring 0
        stosw // como límite de segmento
        mov ecx,[edi] // guarda antiguo
        mov dword ptr [OldGate.offs_low+2],ecx // selector y atributos
        mov eax,0xec000028 // y establece la base a 28 y los niveles de privilegio a
        ECh
        stosd
        shld eax,esi,0x10 // desplaza parte baja de eax a la parte alta y llena la
        parte baja con los bits mas significativos de ESI
        mov cx,word ptr [edi] // copia antigua parte alta
        mov [OldGate.offs_high],cx // del desplazamiento de la compuerta
        stosw // y guarda parta baja de EAX,esto pertenece a la parte alta de la
        dirección a la que saltar en Ring 0
        mov CallGate.seg,0x0f
        mov CallGate.offset32,0x0
        call fword ptr CallGate
        mov edi,dword ptr [GateAddr] // restaura vieja compuerta
        mov ax,[OldGate.offs_low]
        mov [edi],ax
        stosw
        mov eax,dword ptr [OldGate.offs_low+2]
        stosd
        mov ax,[OldGate.offs_high]
        stosw
        popad // restaura los registros

    }

    _asm {


        mov esp, ebp
        pop ebp
        ret

    }

    }

    void main()
    {

        int resp;

        resp=MessageBox(NULL,"Esta aplicación salta a Ring 0 y comprueba la
        existencia de un debugger. Si existe debugger la máquina se colgará
        irremediablemente. ¿ Desea continuar ?",caption,MB_YESNO);

        if (resp==IDYES)
```

```
    {

        TestDebugVXD(&MyProc);
        MessageBox(NULL,"No hay ningún debugger activo",caption,MB_OK);

    }

}
```

You are inside Reversed Minds pages.

Por Mr. Silver / WKT!.
La información aquí vertida es exclusivamente para uso educacional, no puedo hacerme responsable del uso que se haga de esta, por lo que atiendo a la honradez de cada uno :), recuerda que debes comprar el software que utilices :)