Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

Descubrimiento y barrido de nodos remotos en IPv6

por

Ing. Raúl Armando Fuentes Samaniego

Tesis

Presentada al Programa de Graduados de la

Escuela de Ingeniería y Tecnologías de Información

como requisito parcial para obtener el grado académico de

Maestro en Ciencias en Tecnología Informática

especialidad en

Redes y Seguridad Computacional

Diciembre de 2013

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

Escuela de Ingeniería y Tecnologías de Información

Programa de Graduados

Los miembros del comité de tesis recomendamos que la presente tesis de Raúl Armando Fuentes Samaniego sea aceptada como requisito parcial para obtener el grado académico de **Maestro en Ciencias en Tecnología Informática**, especialidad en:

Redes y Seguridad Computacional

Comité de tesis:

Dr. Juan Arturo Nolazco Flores
Asesor de la tesis

Dr. Arturo Servin
Sinodal

Ing. Gustavo Lozano
Sinodal

Ing. Oscar Robles-Garay
Sinodal

Dr. Lorena Gomez
Director del Programa de Graduados

Diciembre de 2013

Todo el trabajo realizado en esta tesis es la culminación de toda la experiencia profesional e incluso personal adquerida en estos años de estudios. Por lo mismo, la dedico a cada una de esas personas que han creído en mi y con su tiempo y esfuerzo me han educado, entrenado y guiado para la culminación de esta tesis.

Reconocimientos

Especial agradecimiento a mi asesor, Dr. Juan Arturo Nolazco Flores, primero, por su apoyo en la creación de has herramientas necesarias para introducir los temas de IPv6 en los laboratorios de redes, consiguiendo con ello los primeros pasos para la elaboración de esta tesis; y segundo, por su guía durante la elaboración de la misma.

A mi familia por el apoyo incondicional que me dieron durante este tiempo. A mis padres por el sustento moral y economico a lo largo de mi vida estudiantil. Y un especial, a mi hermana, Ing. Erika Fuentes, y su esposo, Ing. Eloi Piedallu, por ofrecerme un refugio en su hogar en donde la mayor parte de la tesis fue realizada.

A mis sinodales, en especial al Dr. Arturo Servin, por su apoyo y sugerencias durante la investigación y desarrollo de las herramientas aquí expuestas. Al profesor Alejandro Parra, por su ayuda durante los primeros trabajos con IPv6 y en especial, por ser quien me mostró el potencial de Nmap para esta investigación. Al Dr. Raúl Ramírez por su apoyo durante la maestría.

A mis amigos que me apoyaron durante todo el desarrollo de la maestría, en particular a la Ing. Lirida Hernández, Ing. Hector Yusel, Lic. Diana Martín e Ing. Myriam Alanis.

A la comunidad de desarrolladores, liderados por Fyodor, de la aplicación Nmap.

Raúl Armando Fuentes Samaniego

Instituto Tecnológico y de Estudios Superiores de Monterrey
Diciembre 2013

Descubrimiento y barrido de nodos remotos en IPv6

Raúl Armando Fuentes Samaniego, M.C.T
Instituto Tecnológico y de Estudios Superiores de Monterrey, 2013

Asesor de la tesis: Dr. Juan Arturo Nolazco Flores

Internet es uno de los logros más importantes creados en el siglo pasado, sin embargo, en el 2011 llegó a su límite de crecimiento, pues el rango de 32 bits de direcciones IP se agotaron. Como solución nace lo que se denomina IPv6 con 128 bits. Sin embargo, este nuevo protocolo trae nuevos paradigmas y nuevos retos. En esta tesis nos enfocamos en el aspecto técnico de cómo realizar una búsqueda de nodos en redes IPv6 remotas de forma eficiente. La fuerza bruta de IPv4 se enfrentaba a no más de 4 mil millones de direcciones, pero en IPv6, solamente con la porción de *hosts* estamos hablando de un número de 37 cifras. Para poder hacer esta numeración debemos recurrir a diferentes técnicas basadas en el comportamiento humano y en estándares de auto-configuración propios de IPv6. Para propósitos de utilidad, todas las técnicas han sido desarrolladas para la popular herramienta de código abierto Nmap. El producto final es una serie de guiones NSE para Nmap que permiten el uso de 4 técnicas para buscar nodos en sub-redes IPv6, un guión que valida la existencia de sub-redes mediante DHCPv6 (creada para esta tesis) y un reporte final.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La seguridad informática es un campo de muchos matices, que van desde aspectos legales, políticos y procesos de seguridad, hasta a la configuración de los equipos de computo y la realización de auditorías, junto al análisis de vulnerabilidades y pruebas de penetración (*pen testing* en inglés). **Es justo en estas pruebas que se requiere una etapa de descubrimiento, con la cual se enumeran los dispositivos que se analizarán en posteriores etapas**. Con ellas, se puede determinar las sub-redes que posee la compañía y los nodos individuales dentro de esta [8]. En IPv4, ya existen muchas técnicas bien conocidas, **no obstante con el crecimiento que empieza a tener el uso de IPv6 en el mundo, este acto de enumeración requiere el desarrollo de nuevas técnicas que por sí mismas son un nuevo desafío**.

Una de las características de IPv4 es el formato de las *direcciones IP* que permite identificar *hosts* o dispositivos que se hallan dentro de la red IP. Estas direcciones tienen un formato decimal dividido en 4 segmentos denominados octales, debido a que cada uno de ellos abarca números de 8 *bits*. Como resultado, se tienen 32 *bits* para identificar diferentes dispositivos, los cuales vienen representando 4,294,967,296 nodos en formato decimal. En 1981, el año en que IPv4 tomo el control de la red de redes [9, 10], este número parecía casi infinito; Pero para inicios de los 90 ya se resentían los efectos de la limitante de su dimensión. Como un intento de ganar tiempo se desarrollaron los conceptos de ***sub-mascara de red*** y *Network Address Translation* ***NAT***. El primero tiene como objetivo permitir fragmentar las redes IP de forma más eficiente mediante la técnica *Classless Inter-Domain Routing* (CIDR) la cual introduce un concepto de "sin clases" basado en sub-mascaras de red o prefijo. NAT en cambio introduce la capacidad de tener redes que utilizan las direcciones denominadas "privadas", permitiendo que diferentes entidades posean las mismas direcciones internas pero ostenten direcciones distintas en el exterior [11, 12, 13, 14]. Además de estas inovaciones se iniciaron las investigaciones para un nuevo heredero de Internet [15, 16].

Como solución al agotamiento de direcciones en IPv4 surge IPv6 que trae consigo varias novedades [16], entre ellas un esquema de direccionamiento con un espacio de 128

bits, un número que es absurdamente enorme y que incluso, con el pronóstico de crecimiento actual puede sostenerse por largo tiempo. Sin embargo, es este mismo número el motivo por el que surge el problema que esta investigación intenta resolver: **Mientras que en IPv4, los mecanismos lineales que se utilizan para la enumeración de nodos recaen prácticamente en el uso de fuerza bruta; en IPv6 no pueden ser aplicados pues podrían tomar años**. Ya que por ejemplo, si consideramos que una prueba por cada nodo tomará un segundo, realizar dicha prueba en todos los nodos posibles en IPv6 tomaría 5 mil millones de años y aún mejorando ese tiempo por cada nodo seguiríamos con valores que prácticamente son infinito [17].

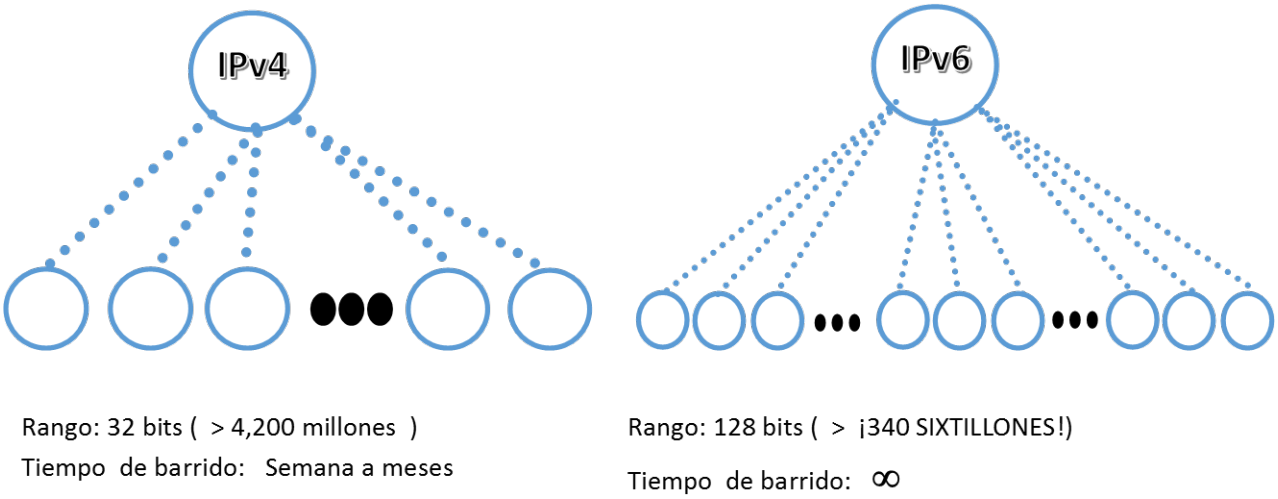

Figura 1.1: Estimaciones de tiempos para completar un barrido en IPv4 e IPv6.

Por esta razón, utilizar mecanismos lineales para enumerar los nodos de una red IPv6 no es viable y se tienen que buscar mecanismos alternativos eficientes aprovechando el diseño de IPv6. Para enfatizar más esta situación consideremos que cualquier institución o entidad para administrar su red interna, pasaría de poseer menos de 32 *bits* en IPv4, a tener hasta 90 *bits* en IPv6 [18, 19]. Como se verá en el siguiente capítulo, de estos posibles 90 *bits*, 16 están destinados para administrar sub-redes y los 64 *bits* restantes para asignárselos a los nodos. ¡La porción de nodos supera miles de veces la capacidad de IPv4! Es por esta enorme diferencia de capacidades de nodos en IPv6 que en **esta investigación proponemos un conjunto de técnicas que nos permite realizar una enumeración eficiente y escalonada de nodos en tales redes, permitiendo obtener tiempos adecuados para realizar dichas actividades**.

Con el propósito de que el resultado de esta investigación tenga aplicación practica en el proceso de análisis de vulnerabilidades, se implementa las técnicas aquí desarrolla-

das, en la aplicación de código abierto **Nmap**. Dicha aplicación ha sido nutrida, desde su concepción en 1997, por una fuerte comunidad de *hackers*, consiguiendo que sea una de las mejores herramientas disponibles para la realización de auditoría de seguridad y descubrimiento de redes. Dentro de sus bondades recae la capacidad modular de añadir escaneo de puertos, descubrimiento de sistemas operativos, uso eficiente de los recursos del sistema para la realización de las pruebas y principalmente ofrecer, mediante el lenguaje de programación Lua, la capacidad de realizar funciones especificas en diferentes niveles [7].

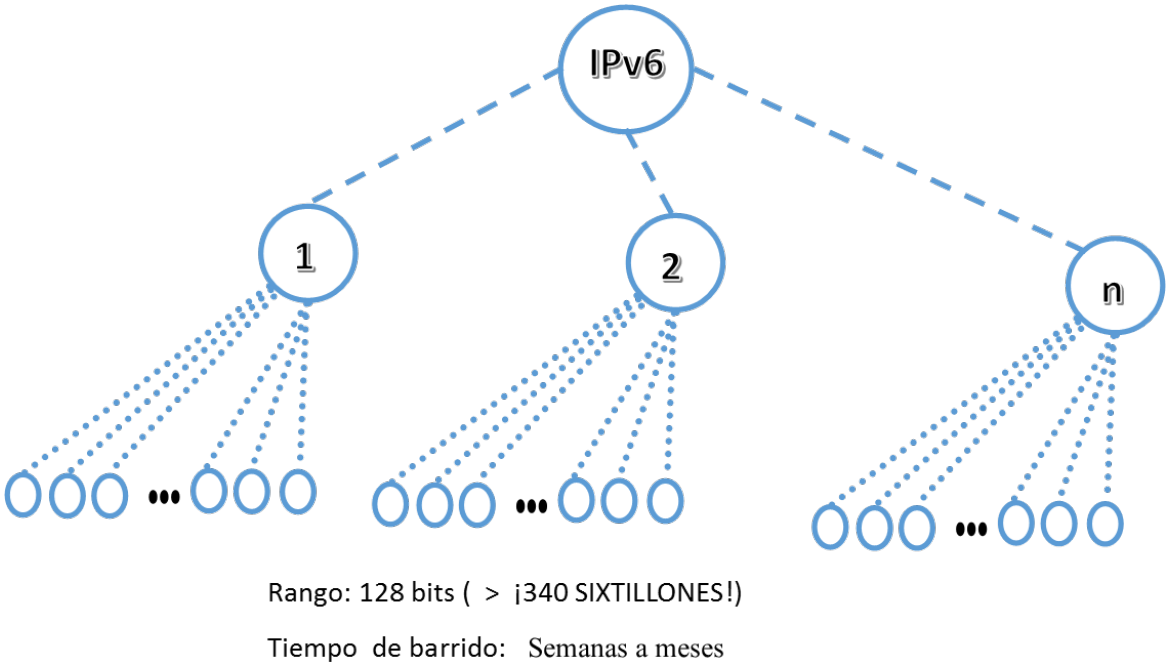

Figura 1.2: Propuesta de investigación para barrido de IPv6.

Dichas técnicas se dividirán en la exploración de sub-redes y la exploración de nodos dentro de estas. Las primeras tendrán como objetivo tener la capacidad de confirmar que sub-redes proporcionadas por el usuario existen dentro de una entidad. Y una vez validada su existencia, se pasarán a las técnicas de exploración de nodos para buscar en dichas sub-redes potenciales nodos. El énfasis de poder confirmar la existencia de sub-redes IPv6 es sencillo: Se pueden tener 16 *bits* destinados a sub-redes que significa que estamos enfrentando un escenario de más de 65 mil sub-redes, cada una con 2^64 *hosts* [19, 18]. **Es necesario acotar la búsqueda solamente en sub-redes que existan para reducir la carga de trabajo**.

Una vez realizada las exploraciones se generará un reporte que indique el total

de direcciones descubiertas. Y es justamente aquí donde Nmap resulta ideal, pues el usuario puede ejecutar dichas técnica a la par de otras capacidades de Nmap, tales como exploración de puertos, detección de sistemas operativos, entre otros. O bien, solo realizar la enumeración básica de los nodos existentes para futuros análisis.

Como resultado final de esta investigación, se integrará a Nmap, mediante guiones NSE (*Nmap Scripting Engine*), la capacidad de modular de las técnicas propuestas; de tal forma, que los usuarios que utilicen dicha aplicación puedan realizar exploraciones eficientes en IPv6, en conjunto de sus propias auditorías de seguridad.

Capítulo 2

Revisión Bibliográfica

2.1. Importancia de búsqueda de nodos en auditorías (IPv4)

2.1.1. Barridos masivos en IPv4

En IPv4 es relativamente fácil realizar un barrido de fuerza bruta, que en ocasiones se le denomina barrido completo (**sweep scan** en inglés) y el cual consiste en probar cada una de de las posibles direcciones de una red en particular. Se puede realizar mediante diferentes mecanismos, los cuales tienen en común generar paquetes de diferentes naturaleza para hacer que el *host* destino responda a ellas. En algunos casos, estos mensajes fabricados (*spoofed* en inglés) serán para ciertos servicios en las capas superiores, y en otras ocasiones para la capa de red; en escaneos avanzados pueden ser una amalgama de estos mensajes fabricados. Sin importar la técnica empleada a estos mensajes les denominan sonda (*probe* en inglés).

Así por ejemplo, podemos enviar un *probe* con un mensaje del tipo *HTTP REQUEST* del directorio principal a todos los nodos dentro de un rango de escaneo, es decir un barrido. Dicho *probe* solo tendrá sentido a aquellos nodos que ofrezcan servicios web y por lo mismo responderían a dicho *probe* [20]. Para que este ejemplo funcione, los *probes* que generemos deben ir lo más estipulado posible respecto a la aplicación o protocolo que queremos probar. Esto último significa que la creación de mecanismos para realizar estos escaneos tiene un costo variable en recursos y tiempos. Sin embargo, existen ya varias herramientas especializadas como ***Nmap*** que los generan de forma automática o facilitan la fabricación de los mismos [7].

Tanto IPv4 como IPv6 tienen un conjunto de mensajes planeados para obtener respuestas de los nodos, con el propósito de poder realizar análisis del estado de la red en un momento determinado. Dichos mensajes están definidos en el protocolo denominado ***Internet Control Message Protocol*** **(ICMPv4 e ICMPv6 respectivamente)**

el cual es un conjunto de herramientas dentro de sus respectivos protocolos de Internet. En IPv4 tiene como objetivo dar reportes de errores en el proceso de datagramas o problemas dentro de la red [21]. Sin embargo, algunos de sus recursos originales cayeron en desuso a lo largo de los 40 años de servicio debido, a que no han sido del todo útiles y/o seguros [22, 23]. Además, otros mensajes como `ping` y `traceroute` suelen quedar limitados dentro de las redes corporativas a causa de políticas de seguridad [22, 23].

Con ICMPv6 se intenta darle nuevas funcionalidades, particularmente se le dota de un sub-protocolo denominado *Neighbor Discovery* (ND) que mediante el paso de 4 mensajes permite la auto-configuración de los nodos, a partir de la información proporcionado por el *Gateway*. ND también valida que las direcciones que los nodos tomen sean efectivamente únicas dentro una red de área local. Además, permite validar la existencia de un nodo previo al paso de mensaje, por lo mismo elimina la necesidad de utilizar el protocolo *Address Resolution* (ARP). [24, 25].

En esencia, ICMP sería suficiente para realizar un barrido completo, pero ¿Qué tan eficiente sería realizar un barrido de fuerza bruta utilizando exclusivamente paquetes de ICMP? La respuesta debería ser **totalmente ineficiente**. Particularmente si ésto se realiza a redes exteriores, ya que se trata de una práctica común de seguridad el no permitir que tráfico de ICMP traspase las fronteras de una red empresarial, ya sea del interior al exterior o viceversa. Pero, ¿Se realista esto en la práctica? Hasta hace poco era difícil contestar a esta pregunta, pero durante el año 2013 han ocurrido dos eventos que permiten un nuevo panorama: El primero se trata de un barrido completo de Internet (a todos los bloques de direcciones de IPv4 públicos) mediante el uso de ICMPv4 [26, 27] y el segundo se trata de algo similar pero ejecutado mediante un mecanismo de *Botnets* (y posiblemente ilegal en múltiples países) cuyo autor se mantiene en el anonimato [1].

El primer caso, fue realizado por un miembro ejecutivo de la firma *Rapid7*, compañía especializada en seguridad. su experimento consistió en ejecutar una serie de `pings` por semana a todas las posibles direcciones de Internet durante el período del primero de junio al día 17 de noviembre del 2012. El objetivo era, no solo validar la disponibilidad del nodo, sino también validar la existencia del estándar de protocolo *Universal Plug and Play* (UPnP) que es ampliamente utilizada por la mayoría de los sistemas operativos modernos y que no debería estar expuesto al exterior, ya que al posee vulnerabilidades bien conocidas [28]. Mientras que el documento realizado por Moore se enfoca en los riesgos de poder manipular UPnP y detectar a los nodos que lo utilizan [27], en este documento nos interesan los siguientes resultados: Los *probes* fueron enviados a 3,700 millones de direcciones posibles, de ellos 310 millones (8 %) permitían el acceso a UPnP desde el exterior, es decir, no había ningún elemento de

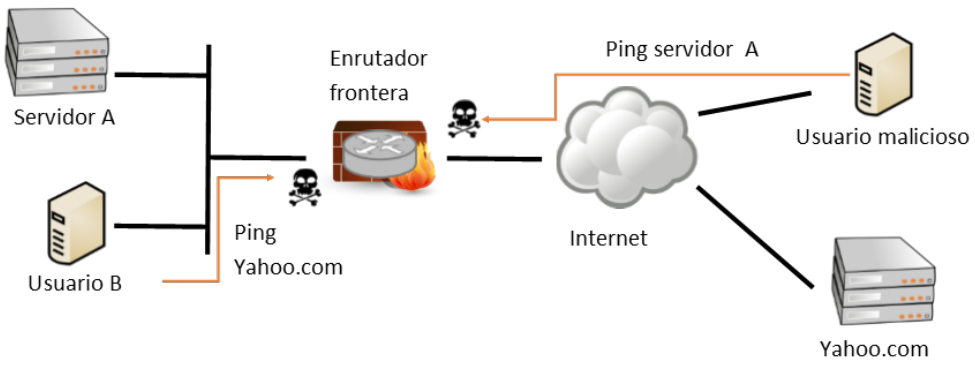

Figura 2.1: Usualmente los `pings` no se les permite atravesar las fronteras entre un dominio y el exterior.

control que bloqueara el trafico ICMPv4, y mucho menos el relacionado con UPnP.

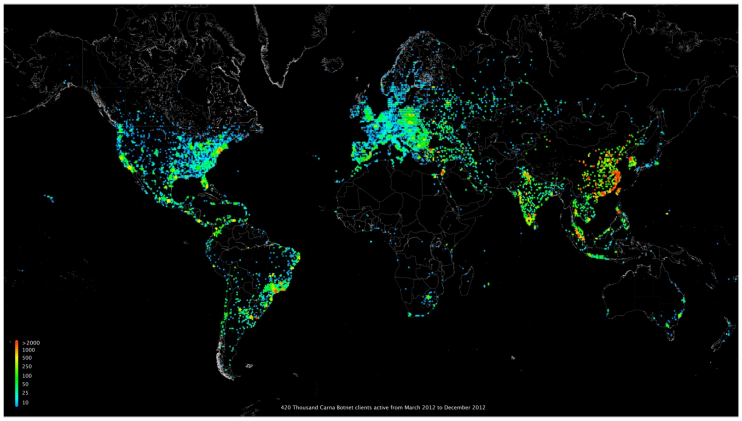

Figura 2.2: Concentración de nodos en el mundo, de acuerdo a la botnet CARNA [1, 2]

El costo del experimento de Moore a sus propias palabras fue elevado, requirió una gran cantidad de equipos de computó -que no se especificó-, un sistema de enfriamiento especial para la ventilación del mismo, y todo este costo que fue cubierto por el mismo Moore, ya que llevó a cabo este experimento por cuenta propia. No obstante, el segundo caso, la botnet Carna, no sufre de este problema, y se le puede denominar como un *censo no oficial de Internet* [29, 2, 1]. Carna, una *Botnet* o red de máquinas infectadas por un *bot* o programa malicioso que permite a un atacante tomar el control de un equipo infectado, logro propagarse, según palabras del autor anónimo, en más de 1,200 millones de *hosts* y selecciono como *agentes de trabajo* a 420,000 *hosts* comprometidos que poseían algún sistema operativo GNU/Linux, además de suficiente capacidad en el procesador y la memoria RAM para realizar el escaneo masivo y pasar desapercibido. Estas computadoras infectadas realizaron el proceso de escanear el resto de las

Total (Aprox)	3,700 millones				
Escenario	Nu. Nodos usados	meses	días	horas	minutos
Moore (a)	7	25864875	25864875	25864875	25864875
Moore (a)	10	1616555	1616555	1616555	1616555
Moore (a)	12	101035	101035	101035	101035
Moore (b)	16	25864875	862162	35923	599
Carna	420,000	1616555	53885	2245	37

Cuadro 2.1: Casos reales de escaneo de fuerza bruta en direcciones públicas IPv4 .

direcciones globales utilizando como herramienta el programa Nmap, y procedieron a realizar diferentes tipos de pruebas en cada nodo durante un lapso de 9 meses[29]. Entre sus principales resultados destaca: una exploración positiva de más de 1,300 millones de direcciones en uso, 141 millones de ellas detrás de un firewall. Y, un archivo de 9 Terabytes conteniendo una variedad de registros de diferentes tipos de *probes* [29, 2, 1].

En ambos escenarios hay algo en común, **un barrido completo y consistente en Internet (IPv4), con la posibilidad de detectar *hosts* cambiando de direcciones, toma meses**. Ambos actos tomaron alrededor de 6 a 9 meses para lograr su cometido. Es importante recalcar que se incluyo un escaneo de puertos y búsqueda especifica de vulnerabilidades en los *hosts*. En ambos casos estamos hablando de 3,700 millones de direcciones a revisar con diferente carga de trabajo; en el caso de Moore desconocemos el número de equipos utilizados, pero supongamos que se trato de 16 equipos, mientras que Carna utilizo 420 mil nodos para realizar el escaneo. En un lapso de 9 meses para realizar tanto barridos como escaneo, en el escenario de Moore, cada máquina debería revisar simultáneamente 599 nodos por minuto, mientras que Carna serían 38 nodos por minuto, tal como se aprecia en la tabla 2.1 .

Ambos números resultan ser pequeños para la realización de una auditoría y exploración de vulnerabilidades. Claro esta, que si la auditoría y las pruebas de penetración son más complejas este escaneo tomaría mayor tiempo. Además entra en juego si un nodo esta accesible por el barrido o no lo esta, el proceso puede variar según el mecanismo utilizado, en el escenario de Moore se iba por ping y peticiones UPnP, que podrian resolverse en pocos segundos, en el caso de Carna, utiliza herramientas de exploración de Nmap el cual podría elevar el tiempo de escaneo a varios segundos.

¿Qué tanto se verían afectados ambos escenarios si se hubiesen llevado a cabo en IPv6? Podemos contestar esta pregunta viendo dos aspectos de dicha red: el espacio de direcciones y la cantidad de nodos existentes en la actualidad. En el primer caso la IANA define un gran bloque que es el 2000::/3 destinado únicamente a direcciones globales [18], que son las direcciones que veremos en Internet, por lo mismo realizar un

barrido de direcciones en todo el dominio de Internet de IPv6 pasa de 32 bits a 125, que expresado en números decimales se trata de una cantidad de 37 cifras. Por otro lado, esta el segundo aspecto: al estar las redes IPv4 emigrando apenas a IPv6 el número total de dispositivos en línea puede no ser tan superior a los 3,700 millones de direcciones de IPv4 dando como resultado que gran parte del espacio de direcciones IPv6 estén desocupados por ahora. De forma teórica, los experimentos podrían llegar arrojar datos similares pero ocuparían mucho más tiempo para poder llegar a los mismos resultados.

2.1.2. Nmap

La aplicación Nmap, que es la abreviación para ***Network Mapper***, se trata de una aplicación de código abierto para auditorías de seguridad de redes, creada por G. Lyon. Nmap utiliza la generación de paquetes IP para el descubrimiento de nodos en la red, los servicios e incluso los sistemas operativos que poseen. Puede ofrecer la capacidad de detectar medidas de seguridad como *firewall* entre muchas otras cosas[7]. Esta herramienta existe desde 1997 y a lo largo de los años ha ido madurado hasta llegar a ser recipiente de grandes reconocimientos. Particularmente, la enorme comunidad de desarrolladores a lo largo de los años la han hecho una herramienta versátil y completa al añadirle gran capacidad modular, volviéndola una de las mejores opciones existentes para cualquier análisis sobre redes TCP/IP para descubrimiento, monitoreo y *troubleshooting* por más complejos que estos sean. Respecto a IPv6, a Nmap se le implementó soporte desde su quinta versión, que ha ido mejorando desde entonces.

Una de las grandes cualidades que ofrece Nmap en su versión 6.25 son los guiones (*scripts* en ingles) denominados ***Nmap Scripting Engine*** **(NSE)** que permiten al usuario escribir guiones, mediante el lenguaje Lua, para automatizar una gran variedad de tareas de administración de las redes. Estos guiones se ejecutan paralelos a las funciones base de Nmap y permite ofrecer una enorme diversidad de funcionalidades dando como resultado un gran incremento de análisis a Nmap. De los *scripts* añadidos por defecto en Nmap, se tienen guiones para analizar una gran variedad de tipo de servicios, la ejecución de revisiones de vulnerabilidades e incluso la posibilidad de explotar ciertas vulnerabilidades[7]. En palabras cortas, **NSE permite hacer uso de todos los recursos que ofrece Nmap para análisis de nodos sin tener que interactuar directamente con el código fuente**. Además, el lenguaje Lua es de fácil aprendizaje y la comunidad de Nmap ha desarrollado suficientes librerías para enfocar todo el potencial de Lua hacia las herramientas de Nmap.

Por lo anterior Nmap fue escogido como herramienta base para el desarrollo de todas las técnicas propuestas en esta investigación. Dichas técnicas estarán desarro-

lladas en *scripts* NSE. De esta forma permite enfocarse completamente en el objetivo de realizar barridos inteligentes sobre IPv6, dejando la parte de análisis y escaneo de *hosts* a Nmap. Al ser esta aplicación un elemento vital para el desarrollo de la investigación, este capitulo estará enfocado en cubrir los elementos claves de Nmap para la implementación de las técnicas y habrá material adicional disponible en los apéndices.

2.1.3. Fases de Nmap

Cuando se ejecuta Nmap, este puede recibir múltiples argumentos de entrada, tales como rango de direcciones y/o puertos, e incluso los tiempos que debe de dedicar a las exploraciones durante el análisis, entre otros. Todos estos argumentos entraran en juego según la fase en la que se encuentre la ejecución. Dichas fases son las siguientes:

1. *Script-Pre-scanning*
2. *Target Enumeration*
3. *Host Discovery (IPv4 / IPv6)*
4. *Reverse-DNS Solution*
5. *Port Scanning*
6. *Version Detection*
7. *OS Detection*
8. *Traceroute*
9. *Script Scanning*
10. *Output*
11. *Script post-scanning*

No toda las etapas son obligatorias, y el alcance de ellas puede estar controlado mediante argumentos dados por el usuario al momento de ejecutar Nmap. De hecho, un escaneo básico de la aplicación por lo general omite la fase 1,8,9 y 11. Particularmente la ultima fase no es ejecutada por ningún elemento existente en la instalación por defecto de Nmap (hasta la versión 6.25) y es más comumente usado para *scripts* personalizados de los usuarios. Los cuales se utilizarán cuando el usuario ejecute ciertos comandos.

Las técnicas aquí propuestas se hallan en las fases 1, 9 y 11. Específicamente tendremos guiones enfocados en el descubrimiento de sub-redes, otros enfocados al descubrimiento de nodos y uno final para generar un reporte.

2.1.3.1. *NmapScripting Engine* (NSE)

Como ya se dijo previamente, una de las características más poderosas de Nmap es que permite al usuario el desarrollar guiones personalizados para incrementar las capacidades de la aplicación. Estos guiones están escritos en el lenguaje de programación Lua [6]. De tal manquera que se puede integrar a cualquier aplicación escrita en lenguaje C como Nmap. Específicamente la aplicación provee librerías que convierten funciones escritas en código Lua a funciones escritas en C para ser ejecutadas por el programa. **Con Lua el usuario puede incorporar, mediante un lenguaje sencillo y eficiente, la capacidad de realizar exploraciones masivas, escaneos especifico e incluso la explotación de vulnerabilidades conocidas sobre el listado de nodos bajo análisis**.

Los guiones o *scripts* NSE se pueden realizar en 3 fases distintas de una ejecución de Nmap, al inicio en la fase de *pre-sccanning* (1), en la fase de *Script* (9) y en la última (11). La primera sirve para hacer análisis sobre la red con la capacidad de añadir nuevos nodos a la fase 2. La segunda hace análisis más profundos a los nodos y tiende a ser donde se concentran la mayoría de los guiones pre-existentes de Nmap. La ultima esta enfocada en la generaciones de reportes finales.

Un guión puede contener código para una o más de las fases en las que se ejecutan. Además, están compuestos por varios campos [7] que se pueden agrupar en los siguientes bloques:

Informativo: Datos del autor, licencia, categoría y dependencias (librerías) a la que pertenece. Usualmente son las primeras líneas del código.

Reglas: El guión se puede ejecutar en 4 fases distintas de Nmap. Para decidir en cuales de ellas se ejecutara se deben definir funciones especiales denominadas “Reglas”.

Acción: Las acciones a realizar cuando la regla se ha cumplido, existe la capacidad de diferenciar acciones para cada una de las posibles fases.

NSE es la razón primordial por la que se escogió Nmap como herramienta base de esta investigación, en lugar de otras existentes o de crear una propia, **ya que nos permite desarrollar las técnicas que se propondrán, sin gastar demasiados recursos**; pues será Nmap quien haga uso de sus recursos para validar la existencia de los nodos que nosotros le indiquemos. Los guiones creados para esta investigación

están disponibles en el apéndice correspondiente.

2.1.4. Técnicas de descubrimiento en redes IPv4 utilizadas por Nmap

Cuando se ejecuta el comando `nmap 10.0.0.0/24` en Nmap se prepara la ejecución, omite la primera fase, pues no ha sido solicitada y procede a enumerar los nodos que estén en el rango de 10.0.0.1 a 10.0.0.254 para posteriormente iniciar una fase de descubrimiento de *host*, la cual puede ser completada mediante diferentes mecanismos. Por ultimo, ejecutará la fase de enumeración de puertos abiertos en dichos *hosts*. En este ejemplo se crean *probes* por defecto que se tratan de mensajes *ICMP echo request*, un paquete TCP SYN al puerto 443, un paquete TCP ACK al puerto 80 y otro paquete *ICMP timestamp* (omitido en IPv6) [7]. Mediante argumentos adicionales se puede cambiar el tipo de *probes* por otros que simulan fragmentos de mensajes en diferentes protocolos, tales como TCP SYN/ACK, UDP, SCTP, INIT e ICMP [7] con lo que se incrementa la posibilidad de obtener mejores respuestas.

Adicional a los mecanismos bases de Nmap para el barrido de nodos se pueden ejecutar guiones NSE para búsquedas de otros tipos, por ejemplocon el comando `nmap --script=discovery` se estarían ejecutando una gran cantidad de guiones especializados en descubrimientos de nodos y/ o servicios con estas características [7]:

1. Servidores específicos (Apache, *bitcoin*, LLTTD, etc.), usualmente buscados mediante mensajes de solicitudes que utilizan direcciones multicast.
2. Resolución de nombres de dominio y direcciones IP a partir de estos.
3. Obtención de información de protocolos de enrutamiento del tipo IGMP.
4. Obtención información de nodos que utilicen direcciones *multicast* para otros servicios.

En general, casi cualquier técnica conocida de IPv4 puede ser ejecutada exitosamente en Nmap y aquellas que requieren un alto grado de especialización se pueden desarrollar mediante guiones NSE.

2.2. Arquitectura e implementación del protocolo IPv6

Como ya se ha mencionado anteriormente, un escaneo de fuerza bruta en IPv6, sobre todo el espacio público o global (Internet), resulta ridículo. Pero, si intentamos centralizarlo a una sola entidad podemos reducir la búsqueda hasta a 80 *bits* como consecuencia del diseño de IPv6. Aunque, utilizando un tiempo estimado basado en el ejemplo de Carna y Moore, mencionados previamente en este capítulo, estriamos hablando de aproximadamente 2 billones de milenios para poder realizarlo. Por lo tanto, debemos de especializar la búsqueda y enfocarnos en elementos que nos permitan reducir aún más esa cantidad de *bits*. Para ello debemos empezar comprendiendo a que nos enfrentamos con el diseño de una dirección IPv6, qué mecanismos se tienen para ser asignadas dichas direcciones y toda estadística que nos pueda ayudar a comprender como se tiende a utilizar estos mecanismos. Por consiguiente hablaremos de ello en dicho orden antes de proceder a la selección de herramientas a utilizar.

2.2.1. Arquitectura de una dirección IPv6

Las direcciones IPv6 están formadas por 128 *bits*, de los cuales se tiene un orden jerárquico que empieza con los *bits* más significativos de derecha a izquierda. Estos 128 *bits* se dividen en 8 segmentos de 16 *bits* cada uno y separados entre ellos por el carácter dos puntos (“**:**”). Todos los segmentos se representan en notación hexadecimal (es decir valores de 0-9 y de A-F donde F es 15). La dirección puede venir acompañada de un prefijo, el cual se escribe con la misma anotación CIDR que IPv4. La figura 2.3 muestra el diseño general.

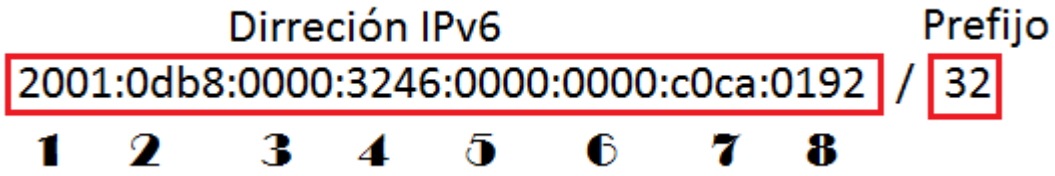

Figura 2.3: Sintaxis de una dirección IPv6

Este formato de dirección es posible de abreviarlo siguiendo dos simples reglas [19]: 1) los ceros a la izquierda de cada segmento pueden ser omitidos. 2) Si uno o más segmentos consecutivos tienen como valor cero, pueden ser omitidos con el símbolo **“::”**, pero solo puede ocurrir una única vez. Con estas dos reglas, el ejemplo de la figura 2.3 se podría acortar como sigue: 2001:db8::3246:0:0:c0ca:192 ó 2001:db8:0:3246::c0ca:192

pero nunca como 2001:db8::3246::c0ca:192 pues es demasiado ambiguo.

La asignación de los *bits* más significativos esta controlado por la IANA [30] y estipulado en el RFC 4291 [19]. Esta divide el espacio de direcciones en grandes bloques de 3 a 10 *bits* [18]. Estos bloques tienen propósitos bien definidos [19] y a continuación se hace mención de aquellos a los que se harán mayor referencia durante esta investigación:

::1/128 - Dirección lógica del host A diferencia de IPv4 solo existe una única dirección lógica (*loopback*).

2000::/3 - Prefijo globales Todas las direcciones públicas inician en este bloque. Las direcciones denominadas *Anycast* son indistinguibles de estas.

2001:db8::/32 - Prefijo TEST-NET Estas son direcciones para propósitos de documentación. **Durante el desarrollo de esta investigación se utilizarán este rango en lugar de los globales**.

FE80::/8 - Prefijo de enlace local Todos los nodos deben ser identificados con una dirección de este tipo, la cual debe de ser única dentro de una porción de red.

FF00::/8 Direcciones *multicasts*. En IPv6 las direcciones *mullticast* sustituyen a los mensajes broadcast y se vuelven mucho más complejas y avanzadas que su contraparte en IPv4

Cada bloque tiene sus propias reglas de asignación definidas por la IANA. Ciertos bloques, como la dirección de enlace local (FE80::/10) y global (2000::/3) se les denominan *scopes* o alcance. Además existen otros *scopes* tales como: local (FC00::/8), privacidad (FD00::/8) y de sitio (FEC0::/10). Es importante entender que, en un momento dado, un nodo cualquiera podría tener una dirección de cada uno de estos *scopes*, o incluso múltiples direcciones. Los *scopes* de privacidad y de local se tienen con el objetivo de manejar una red interna (son direcciones que no deben aparecer en el exterior); son opcionales y para esta investigación no serán motivo de mayor documentación.

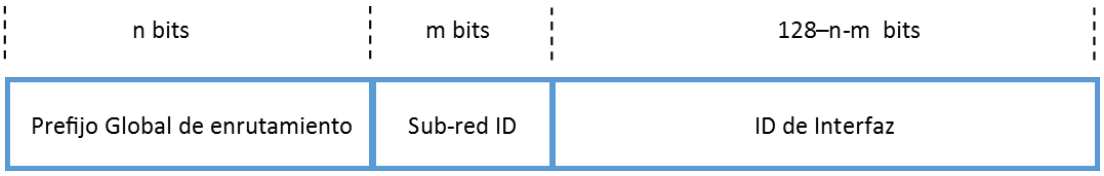

Figura 2.4: Distribución de los *bits* en una dirección IPv6 de alcance global (2001:db8::/4 sera utilizado con estas mismas reglas)

Respecto a la forma de distribuir dichas direccione de prefijo global se representa con la figura 2.4 donde se separa la dirección en 3 componentes: Prefijo global de enrutamiento, ID de sub-red e ID de Interfaz. La primera parte tiende a ser asignado de forma jerárquica y esta bajo control de los RIR e ISP [31]. La segunda parte esta bajo control de la entidad que tiene la dirección de red IPv6 y la tercera tiende a quedar para la asignación de las direcciones de los *hosts* dentro de dicha entidad. Queda de forma variable pues se pueden dar diferentes motivos por los cuales una entidad no se le de más que una única red IPv6 con poca capacidad de hosts (por ejemplo usuarios del hogar) [32, 33]. Un escenario típico es que el prefijo global alcance /48 dejando 16 bits de sub-redes y 64 para los nodos[31].

Un paradigma nuevo de IPv6, que se debe tener presente en todo momento, es que **una sola interfaz de red puede tener múltiples direcciones de red de uno o más bloques de red y de uno o más alcances**. Además, **Todos los nodos deben de tener una dirección del bloque FE80::/10** que corresponde a enlace-local y que tiene una estructuración más simple: Los *bits* 11 al 64 son dejados en ceros y los 64 *bits* restantes quedan configurados mediante SLAAC o un proceso pseudo-aleatorio. Tomemos como ejemplo la tabla 2.2 que posee información condensada de los datos arrojados por los comandos `ipconfig -6` y `route PRINT -6` en un nodo con el sistema operativo *Windows 8*.

Dirección	**Pref.**	**Alcance**	**Configuración**
2001:660:3203:1000:ad54:cc65:989a:f45d	64	global	privacidad
2001:db8:c0ca::dead	64	global	estático
2001:db8:c0ca:abba::1	64	global	estático
2001:db8:c0ca:abba::2	64	global	estático
2001:db8:c0ca:ffff::a	64	global	estático
2001:660:3203:1000:s36e:cdf4:270a:b15e	64	global	privacidad (Temporal)
fe80::ad54:cc65:989a:f45d	64	enlace local	EUI-64.

Cuadro 2.2: Crecimiento de nodos IP durante 1994.

En la tabla 2.2 podemos observar que el nodo posee 7 direcciones de red que corresponden a 5 sub-redes distintas en 2 *scopes*: Globales y de enlace-local (esta ultima siempre mandataria) de las 6 direcciones de alcance local, vemos que el nodo posee dos pares de direcciones en una misma sub-red (2001:db8:c0ca:abba::/64 y 2001:660:3203:1000::/64). Hay 3 formas en que se han configurado estas direcciones: De forma estática y con SLAAC [34], sin embargo en este último se pueden apreciar dos mecanismos distintos: las direcciones de la sub-red 2001:660:3203:1000::/64 han sido mediante un mecanismo de privacidad, que genera números aleatorios, y que cambie la

dirección cada cierto tiempo [35], y la dirección de enlace local que ha sido configurado con un mecanismo denominado EUI-64 [36]. Ambos métodos los veremos más a fondo en capítulos siguientes.

2.2.2. *Network Discovery Protocol* (NDP)

ICMPv6 juega un papel esencial mediante el sub-protocolo *Network Discovery Protocol* (NDP) ya que otro de los nuevos paradigmas de IPv6 es que un nodo debe confirmar primero la existencia de su vecino, incluyendo Gateways, antes de enviar otro mensaje (aunque se tratase de un `ping`). Esto lo hace mediante NDP [37]. Además, cuando un nodo está en proceso de adquirir una dirección o de asignarse una, debe primero confirmar que dicha dirección esté disponible, esto mediante *Neigh Address Duplication*(DAD) [34].

La función básica de NDP es poder realizar *Stateless Address Auto-configuratión* (SLAAC) y auto-asignarse una dirección de IPv6 para cada scope y sub-red correspondiente. Esto se realiza mediante los siguientes pasos:

1. Crear una dirección tentativa de enlace local (EUI-64 o de Privacidad) .
2. Comprobar mediante DAD la disponibilidad de esta dirección (que no exista ya un nodo utilizándola).
3. Si no esta utilizada, asignársela.
4. Utilizar un mensaje NDP denominado *Router Discovery* para propósito de localizar a un Gateway.
5. (Opcional) Solicitar mediante DHCPv6 una dirección. Esta puede sustituir el paso anterior.
6. En caso de respuesta positiva mediante un mensaje *Router Advertisment Message* tomar la información de este para conocer prefijos de otros alcances y/o si requiere hacer uso de DHCPv6 sin estado [38].
7. Para cada sub-red con prefijo de 64 *bits* aplicar los primeros 3 pasos, si hay información que se requiere DHCPv6 aplicarlo.
8. Un nodo con dirección ya asignada, antes de enviar un paquete a cualquier vecino (incluyendo enrutadores) debe primero confirmar su disponibilidad mediante un par de mensajes denominados *Neighbor Solicitation Message* y *Neighbor Advertisment Message*

Para propósito de descubrimiento de nodos, NDP tiene el impacto de permitir que un nodo en estado de escucha sea capaz de descubrir todos los nodos en un enlace local determinado al estar atento de los primeros pasos de SLAAC de los nodos en dicho segmento. Incluso para DHCPv6 uno de los pasos de asignación es el envío de un mensaje DAD. Por lo tanto, **omitiendo la configuración manual, el escaneo de nodos en IPv6 dentro de segmento de área local, es posible de realizarlo de forma pasiva** . Este aspecto de NDP no paso inadvertido desde su diseño, y por consiguiente , existen en la actualidad 2 formas de contrarrestarlo, el primero es la implementación de *Internet Protocol Secure* (IPsec) y el segundo es la implementación de *Secure NDP* (SENDP) [37, 39].

IPsec fue el mecanismo original que se considero para proteger a NDP [37]. Trata de cifrar toda la comunicación de la capa de IP mediante llaves y requiere una configuración manual en los nodos para asignarles dichas llaves. De esta forma, un nodo no autorizado (que no posea las llaves) es incapaz de inferir el contenido del tráfico, y por lo mismo se le dificulta el descubrimiento de los nodos. Pero IPsec implica demasiado trabajo manual [40, 39] y si un nodo queda comprometido será capaz de escuchar el tráfico en la forma tradicional de NDP.

SENDP es un mecanismo de seguridad para enlaces inseguros, tales como señales inalámbricas, y aquellas redes donde ataques al protocolo NDP son tomados en cuenta [40]. SENDP permite a los nodos añadir mecanismos de autenticación para la ostentación de direcciones IPv6, es decir, demuestra que el nodo que dice tener la dirección realmente sea dueño de ella y tenga derecho a poseerla [40]. Su principal beneficio es reducir el trabajo manual que IPsec representa. Aún así, un nodo comprometido al igual que con IPsec sería capaz de hacer una escucha en estado pasivo para descubrir a los nodos vecinos.

Es importante aclarar que el proceso de asignación de dirección, puede variar según las implementaciones de los sistemas operativos. Por ejemplo, los sistemas operativos de Microsoft implementan varios de los RFC ya obsoletos [41]. Una característica muy particular de estas implementaciones de Microsoft es intentar contactar servidores DNS en alcances de la FEC0::/10 que ya están depreciados. Por otro lado, los sistemas operativos basados en GNU/Linux parecen estar también en la misma situación, la mayoría utilizan la compilación del proyecto USAGI (*UniverSAl playGround for Ipv6*) , que de acuerdo al sitio oficial del proyecto, no ha recibido actualización posterior al 2002 [29]. Ambos escenarios reflejan que al momento de escribir esta investigación existen ciertos comportamientos especifico de cada sistema operativo que pueden servir para obtener resultados adicionales.

2.2.3. *Dynamic Host Control Protocol for IPv6* (DHCPv6)

Durante el proceso de configuración mediante SLAAC existe un paso en el cual los enrutadores envían respuestas mediante un mensaje NDP denominado *Router adverstiment*, en el cual existen dos campos que le indican al cliente si debe de solicitar uno o más parámetros a un servidor DHCPv6, tales como dirección IPv6 e información de servidores DNS, dando como resultado que se puedan tener dos tipos de servidores DHCPv6: de-estado (*stateful*) y sin-estado(*stateless)*. Del primero, toda la información proviene de un servidor DHCPv6 evitando la auto-asignación de una dirección de red y el segundo evita la necesidad de actualizar cada enrutador cada vez que un servidor DNS cambie [37].

Se podría decir que DHCPv6 es redundante, pues los nodos tienen capacidad propia de auto-asignarse una dirección mediante SLAAC. Sin embargo, DHCPv6 esta más enfocado en proveer herramientas de auditoría al poder identificar a un nodo en particular en diferentes sub-redes, y en el caso de que se trate de un servicio DHCP de-estado, asignarle una dirección reservada para él, que es bien conocida por el administrador de redes.

De forma similar a su homologo en IPv4, en DHCPv6 tenemos clientes, servidores y agentes de paso (*relay*), estos ultimo tienen como objetivo pasar los mensajes de los clientes, en mensajes *multicast*, a servidores en forma de mensajes *unicast*. Los agentes se utilizan cuando el servidor y el cliente no están en el mismo segmento de área local y deben ser completamente transparentes para estos últimos. En la figura 2.5 se puede apreciar esta información.

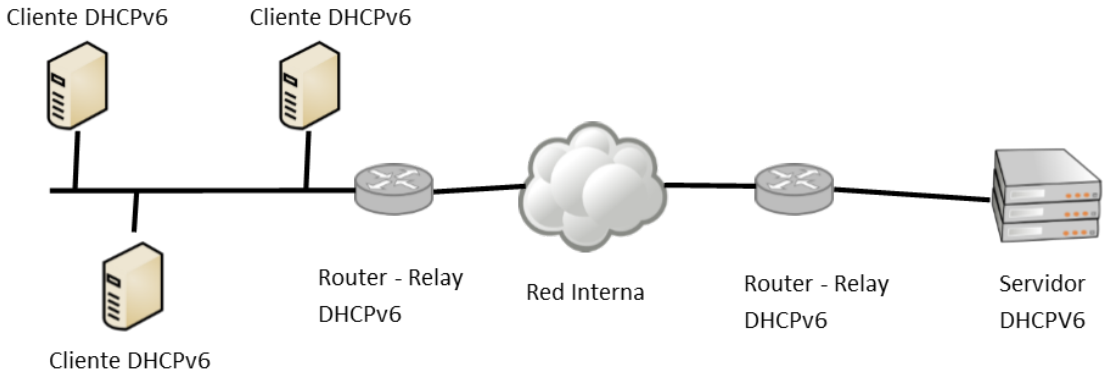

Figura 2.5: Representación de cliente, servidor y agente de paso en DHCPv6.

En ambos casos el nodo que solicita la dirección, un cliente DHCPv6, debe de comunicarse con el servidor DHCPv6 y realizar una serie de pasos para obtener la in-

formación necesaria. La primera vez que el cliente se comunique con un servidor, deberá de generar un número especial denominado *DHCP Unique Identifier (DUID)*, el cual lo identifica entre sesiones y reinicio. Además, tiene como objetivo garantizar la capacidad de procesos de auditoría del cliente.

Con el DUID del cliente, los servidores serán capaces de reconocerlo y volver a asignarle las direcciones correspondientes a dicho nodo a lo largo de la red interna de la entidad [38]. Los servidores también poseen DUID, pero estos serán proporcionados al cliente cuando se le de respuesta a sus peticiones, y tienen como función facilitar al cliente el comunicarse con un servidor en especifico después de la primera solicitud [38].

En general la solicitud de información de un cliente a un servidor DHCPv6 se maneja en dos esquemas: de 4-mensajes y 2-mensajes, siendo el primero el que corresponde al modelo de-estado (*stateful*) y el segundo al modelo de sin-estado (*stateless*) [38].

En el primer caso, el cliente envía un mensaje del tipo *SOLICIT* con dirección destino *multicast* (FF00::/8) en espera que uno o más servidores respondan con el mensaje del tipo *ADVERSTIMENT*, de los cuales el cliente escogerá una de las ofertas, usualmente la primera en llegar, y la solicitará mediante un tercer mensaje denominado *REQUEST*, que saldrá todavía con dirección *multicast*, pero incluirá el DUID del servidor seleccionado. El servidor enviara el cuarto mensaje, *REPLY*, para confirmar que se le han asignado esos recursos al nodo.

En el caso del mecanismo de 2-mensajes el cliente envía un mensaje *multicast* del tipo *INFORMATION-REQUEST*, que básicamente solicita el envió de los parámetros que posea el servidor DHCPv6, a excepción de una dirección IPv6, al cual se le responde con un mensaje del tipo *REPLY* con tal información. En estos mensajes los DUID no son de importancia [38].

Para el mecanismo de 4-mensajes existen algunos elementos de seguridad básicos para intentar reducir los peligros propios del servicio, mismos que existen también para IPv4, entre ellos esta la generación de un campo numérico aleatorio en el paso de los mensajes, además de mecanismos de autenticación entre nodo y servidor, entre otros [38]. Por último, antes de enviar el tercer mensaje, el cliente verifica la disponibilidad de la dirección ofrecida mediante un DAD[38], y existe la posibilidad de encriptar la comunicación entre agentes de paso y los servidores. Por seguridad, DHCPv6 toma medidas para proteger los DUID que conoce.

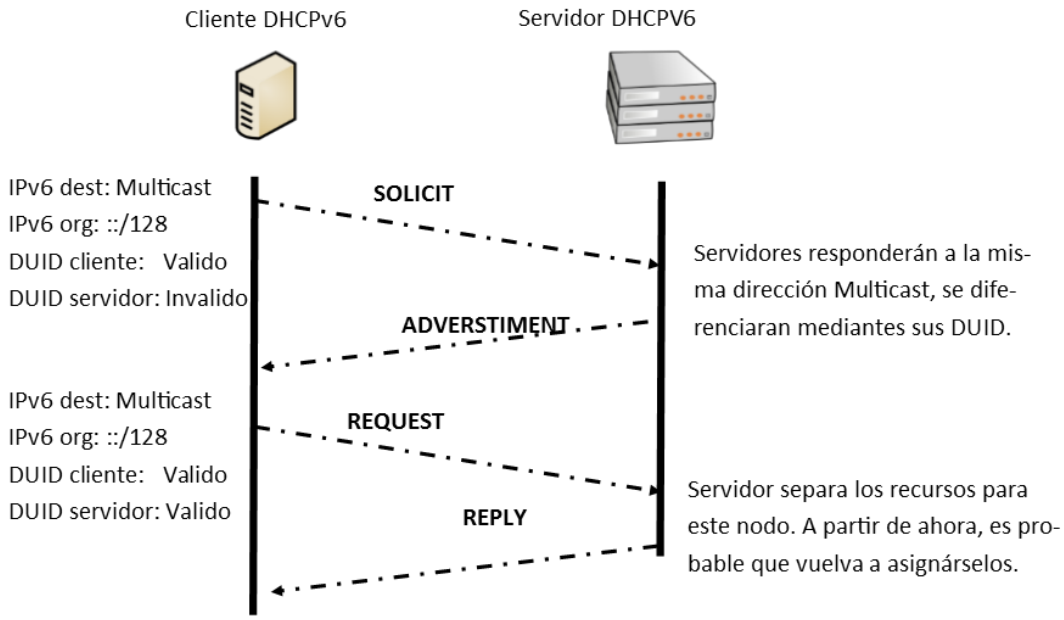

Figura 2.6: Representación de configuración de estado (4-mensajes) entre un cliente y servidor DHCPv6.

2.2.4. Direcciones de privacidad

Cuando un nodo se configura con SLAAC usualmente utiliza el mecanismo denominado EUI-64 que consiste en tomar los 48 *bits* de la dirección MAC y añadirle 16 *bits* constantes para generar la dirección IPv6. Esto en particular se realizará siempre que una sub-red indique que posee un prefijo de 64 *bits*. Como resultado, un *host* en movimiento, por ejemplo una laptop en la red casera, en red de una cafetería y la en red de una compañía, podría llegar a tener los últimos 64 *bits* de forma constante. Por lo mismo se abre la posibilidad de darle seguimiento a un nodo sin importar donde se encuentre. Como un intento de solucionar esto se desarrolló el estándar de direcciones de privacidad en el 2001, mediante la publicación del ***RFC 3041 - Privacy Extensions for Stateless Address Autoconfiguration in IPv6*** y que fue actualizado en el RFC 4941 del mismo nombre, publicado en el 2007. [35, 42].

La esencia de las direcciones de privacidad radica en generar de forma aleatoria los últimos 64 *bits* y a partir de este número crear una nueva dirección aleatoria, una vez transcurrido cierto tiempo (horas a días) para dificultar la posibilidad de relacionar las direcciones a un mismo nodo [35]. No todos los nodos en una red IPv6 se podrían beneficiar de esto. El mejor ejemplo son los servidores, ya que se podría dificultar el acceso a ellos si todas sus direcciones fuesen temporales y siempre cambiantes. No obstante, un beneficio que sí podrían usar los servidores es tener una dirección bien conocida para iniciar comunicaciones - de clientes hacia él - y el servidor haga uso de

sus direcciones temporales cuando desee iniciar una comunicación con cualquier otro nodo. Sin embargo, existen ciertas implicaciones negativas para el rendimiento de una o más aplicaciones pues estás se podrían ver afectadas por el cambio de dirección. [35].

Al utilizar el esquema de direcciones de privacidad, además de las direcciones globales añadidas por otros métodos, se pueden llegar a tener dos pares de direcciones temporales, una por cada prefijo de red que tenga el nodo [35]. Este par existe solo en ciertos periodos de tiempo, que es cuando la actual dirección temporal esta próxima a expirar por lo que se genera una nueva dirección temporal, con el propósito de evitar una condición de carrera o un periodo sin dirección. Por lo mismo se tienen que manejar periodos de tiempo en los que pueden depreciar una dirección temporal. Lo anterior se puede apreciar en la figura 2.7.

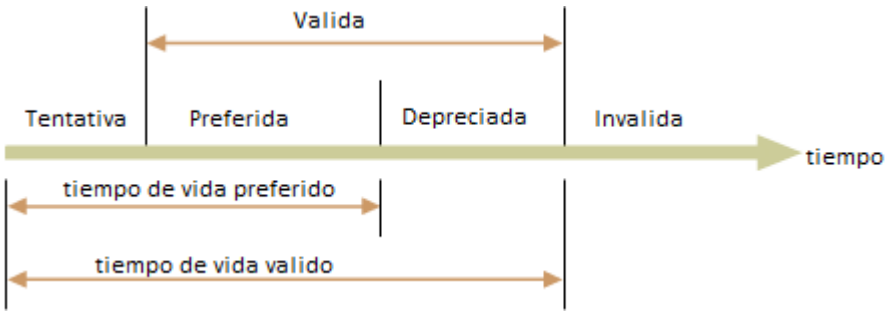

Figura 2.7: Manejo de direcciones de priviacía de Microsoft [3].

La dirección temporal tendrá 4 estados de vida: Tentativo, preferido (válido), depreciado e inválido. Propiamente dicho, el primer tiempo es solo mientras el DAD esta validando que la dirección es efectivamente única, la mayor parte de la vida de la dirección temporal estará en el "estado preferido". Cuando se acerque el momento de generar una nueva dirección temporal, entonces la actual pasara al "estado depreciado", tomen en cuenta que en este estado la dirección es válida y funcional pero es paralela a una nueva dirección temporal que se encuentra en ese instante en el "estado tentativo". "El estado inválido" es cuando ha sido desocupada y podría ser eliminada inmediatamente, pero es preferible esperar hasta que la ultima aplicación que la estuviese utilizando desde antes termine sus actividades.

Las direcciones de privacidad tienen un impacto fuerte en esta investigación, debido a que los mecanismos para generar números aleatorios, basados en MD5 [35], efectivamente eliminan muchos modos para poder hallar nodos que se estén comunicando exclusivamente con direcciones temporales. La única opción viable son los medios indirectos, tales como buscar en el caché de otros nodos, como los Gateways y aun así

debido a que las direcciones tienen tiempo finito, se debe de validar que las ya descubiertas sigan siendo válidas. Sin embargo, de acuerdo al mismo RFC 4941, las direcciones de privacidad deben estar desactivadas por defecto, además de que no necesariamente serán las únicas en funcionamiento.

En el caso de sistemas operativos contemporáneos, los de Mircosoft implementan una versión similar, pero mantienen la misma dirección de por vida y en todo momento posee activa la segunda dirección. En el caso de sistemas operativos basados en GNU/Linux la mayoría utiliza el conjunto de protocolos USAGI el cual no lo implementa por defecto.

2.2.5. Implementaciones reales

Como ya se mencionó anteriormente, una entidad tal como una compañía, puede poseer hasta 80 *bits* para administrar los nodos y sub-redes dentro de ella. Particularmente posee 16 *bits* para la administración de sub-redes, poco más de 65 mil sub-redes, y 64 *bits* para administrar los nodos. El incremento de la población máxima de nodos en una sub-red en IPv6, es indudablemente muy superior que en IPv4, sin embargo también es muy cierto que las empresas no incrementen la cantidad de nodos que poseen por sub-red [5] que no sobrepasaría los miles de usuarios, salvo casos muy específicos. Como resultado tendremos sub-redes con alta capacidad de población pero con muy pocos nodos dentro de ellas, es decir, serán sub-redes de baja densidad de población [5].

Como consecuencia, los métodos tradicionales de escaneo remoto en puertos UDP y/o TCP para descubrir servicios habilitados o en ejecución en un nodo, se vuelven menos factibles [17]. Ya que deberán abarcar barridos de grandes dimensiones, superiores a IPv4, para detectar los nodos. Sin embargo, esto no necesariamente tiene que quedar así, los barridos de fuerza bruta pueden, a simple vista, parecer obsoletos, sin embargo, si los enfocamos o los especializamos es posible reducir el rango a buscar con cierta probabilidad de resultados positivos. Pero, ¿Cómo los especializamos? ¿En qué enfocamos? Para responder esas preguntas debemos conocer la tendencia en cómo se administran las sub-redes IPv6 actualmente. Para ello, utilizaremos los análisis realizados por Malon que en el 2008 analizó los métodos de asignación de direcciones IPv6 obteniendo los resultados mostrados en la tabla 2.3 [4, 5].

En la tabla 2.3 se tienen los resultados que arrojaron los análisis en el 2008 de la distribución de direcciones IPv6 en los nodos *hosts*:

SLAAC utilizan la implementación de configuración sin estado, típicamente EUI-64

Tipo de Dirección	Porcentaje
SLAAC	50 %
IPv4-based	20 %
Teredo	10 %
Low-byte	8 %
Privacidad	6 %
Palabras (Wordy)	<1 %
Otros	<1 %

Cuadro 2.3: Técnicas empleadas para asignación de direcciones a *hosts* en sub-redes IPv6 [4, 5].

IPv4-Based están basadas directamente en las direcciones IPv4 que se tenían originalmente, una representación típica de ellas es:
X:X:X:X::a.b.c.d. ó ::a.b.c.d.

Teredo propiamente dicho, no son sub-redes configuradas por la compañía, si no que los nodos utilizan servidores ajenos a esta para acceder a una red IPv6 mediante túneles en IPv4.

Low-Byte son direcciones de red IPv6 que utilizan exclusivamente los *bits* más bajos, por ejemplo: ::1, ::A o ::ABCB.

Privacidad direcciones temporales de Privacidad.

Wordy es crear direcciones IPv6 que parezcan palabras, por ejemplo
::DEAD:BEEF.

Otros tipos de asignaciones tales como DHCPv6.

Cabe resaltar, que esto es del 2008, las tendencias pueden estar cambiando fuertemente, pero sirve como un punto de partida para el desarrollo de esta investigación. Posiblemente, las direcciones de Privacidad estén adquiriendo mayor auge debido a que son las implementadas por sistemas operativos basados en Microsoft y estos al momento de escribir la investigación dominan fuertemente el mercado [43]. Por lo mismo la situación de los túneles teredos puede haberse incrementado, ya que son soportados por defecto por los sistemas de Microsoft, sin embargo, conforme las instituciones adquieran sus propias redes IPv6 estos túneles provisionales pueden ir decreciendo en los clientes.

Los túneles teredo son un mecanismo de transición [44] y tienen la particularidad de ir por paquetes IPv4, por lo mismo quedan fueran del tema de estudio de esta investigación.

Tipo de Dirección	Porcentaje
Low-byte	70 %
IPv4-based	5 %
SLAAC	1 %
Palabras (Wordy)	<1 %
Teredo	<1 %
Privacidad	<1 %
Otros	<1 %

Cuadro 2.4: Técnica empleada para asignación de direcciones a enrutadores en sub-redes IPv6 [4, 5].

Además de como se configuran los nodos clientes, es necesario conocer como se configuran los enrutadores y servidores, utilizando la misma línea de investigación manejada por Malone [4, 5] tenemos la información desglosada en la tabla 2.4 donde se puede apreciar que existe una gran diferencia respecto a los nodos, por lo general, se va por la idea de dejar fácil la configuración de los mismos. Por ello es normal que utilicen principalmente el formato de direcciones predecibles como los de bytes más bajos. Esto es similar, a la práctica común de utilizar las direcciones más bajas o altas en sub-redes IPv4 para asignárselo a los enrutadores y otros servidores. Respecto a seguridad es posible que no importe mucho pues los Gateway pueden ser fácilmente descubiertos, pues responderán a mensajes de solicitud de NDP además que periódicamente se anunciarán [37].

En general, estas tablas marcan una cierta tendencia de configuración y en base a estas, se pueden crear herramientas con el objetivo de buscar direcciones bajo esos esquemas conocidos. **Durante esta investigación desarrollaremos particularmente 6 aproximaciones para tratar de cubrir las más comunes, que sobresalen en las tablas 2.3 y 2.4.**

2.3. Técnicas y herramientas conocidas para enumeración de IPv6

El escaneo de nodos en redes IPv4 es el primer paso de toda auditoría de seguridad y/o de cualquier inicio de un ataque, por lo tanto existe una gran variedad de herramientas para IPv4. Sin embargo no ocurre lo mismo con IPv6, que a pesar de que existe formalmente desde 1999 su lenta inclusión en las redes empresariales no dio necesidad de crear dichas herramientas. Por consiguiente el listado de herramientas disponibles para trabajar con IPv6 es algo más limitado, pueden ser extensiones de herramientas

ya existentes o en últimas instancias son herramientas simplistas como las funciones básicas de ICMPv6 para un solo nodo.

Lo siguiente es el listado de herramientas de mejor calidad recopiladas para trabajar con IPv6:

THC - Tiene su propio conjunto herramientas especializadas en auditoría de seguridad con IPv6. Denominadas THC-IPv6 No es posible acceder a todas las herramientas sin cooperar primero con ellos [45].

6Tools - Una aplicación escrita en C iniciado por Gont que tiene como objetivo el escaneo de nodos remotos [46].

Nmap - Una popular herramienta de escaneo de redes IPv4 que ha incluido soporte a IPv6 [7].

Topera - Una herramienta de escaneo de IPv6 hecha específicamente para probar la seguridad de sistemas de detección de intrusos. [47].

THC-IPv6 es el resultado de una comunidad de desarrolladores de auditoría de seguridad, aunque propiamente no es un escaner sino que es un conjunto de varias herramientas escritas en lenguaje C, que tienen como objetivo provocar ataques de negación de servicio, dificultar la comunicación o exponer nodos y que se han adaptado a utilizar el protocolo IPv6. Algunas de estas herramientas permitirían hasta cierta medida el descubrimiento de nodos. Sin embargo, el uso gratuito del código no incluye todo el *set* y para obtenerlo es necesario primero formar parte activa de la comunidad de este proyecto [45].

6Tools es un conjunto de herramientas escrita enteramente en lenguaje C con objetivos diversos como detección de nodos y ataques específicos para la realización de auditorías. Sin embargo, **algunas de estas herramientas requieren que los nodos objetivos se hallen dentro de un mismo segmento de área local** mientras que nuestro objetivo es particularmente nodos remotos. Algunas de sus herramientas ya incluyen atacar directamente los nodos para obtener respuesta, mientras que otros son más benévolos y solo intentan confirmar la existencia de los nodos mediante *probes* poco agresivos. [46]. 6Tools esta bajo constante desarrollo [46].

La aplicación Nmap es bastante popular y fue creada por otra comunidad de desarrolladores. Lleva existiendo desde 1999 [7] y es una de las herramienta más robusta y completas en redes IPv4, dando la capacidad de hacer búsquedas de nodos basándose solo en ICMPv4 o incluso una gran variedad de aplicaciones con puertos UDP o TCP. Además ofrece mucha flexibilidad para la creación de análisis más complejos y variados

según la necesidad del usuario. En el momento de escribir la investigación Nmap liberó la versión 6.25 y desde la versión 5.00 incluye un soporte a IPv6 aunqu,e este es limitado al excluir muchos de los mecanismos que ofrece para trabajar múltiples direcciones de IPv4.

Topera es el resultado de la publicación de un trabajo de investigación en la conferencia de seguridad "Navaja Negra", aunque propiamente es un escaner capaz de realizar pruebas similares a Nmap, tiene la desventaja que se trata más de un caso ejemplo que de un escaner generalizado, ya que Topera demuestra la debilidad de un producto de seguridad informática denominado "Snort", enfocado a la prevención de intrusos. Topera genera todo el escaneo mediante encabezados adicionales al protocolo IPv6 para engañar a "Snort". Topera difícilmente se actualizará en un futuro. Además, su escaneo es bastante básico [47].

A excepción de la herramienta 6Tools toda las demás herramientas no pueden soportar por defecto un rango de direcciones en IPv6, por lo tanto es requerido automatizar el proceso mediante guiones especiales, a través del lenguaje AWK, por ejemplo [48]. Sin embargo, 6Tools esta desarrollo, pues fue iniciado en el 2012. **Para propósitos de esta investigación se ha escogido tomar como punto de partida el proyecto Nmap para el desarrollo de todas las propuestas que a continuación describiremos.**

2.3.1. Modos pasivo

Debido a la implementación de NDP un nodo previo a enviar un mensaje a otro primero debe descubrir la existencia del nodo destino o del Gateway. Como resultado es posible realizar una enumeración de nodos disponibles en un segmento de área local al estar exclusivamente escuchando los mensajes NDP *Neighbor adverstiment*, *Neighbor Solicitation* y *Router Advertisement*.

Para realizar las escuchas no se requiere más que herramientas que permitan analizar el medio físico de la conexión o *sniffers*. Existe una gran variedad de ellas, siendo las más populares Wireshark y TCPDump [49].

2.3.2. Técnicas conocidas

Existe una gran variedad de técnicas para poder sustraer información de nodos en línea para IPv6, algunas operan de forma indirecta y otras de forma directa. La mayoría de estas técnicas, que provienen de IPv4, están documentadas en el *RFC 5157 IPv6 Implications for Network Scanning* y se enumeran a continuación [17]:

1. *DNS Adverstised Hosts*.
2. *DNS Zone transfer*.
3. Análisis de archivos de bitácoras (*logs*).
4. Grupos *multicasting* (como los ya mencionados con Nmap).
5. Comportamiento de aplicaciones (Por ejemplo, aquellas *peer-to-peer*).
6. Mecanismos de transición (*Dual-stack*, túneles, etc).

De los mecanismos de transición, *dual-stack* puede ser de interés, ya que se resume en que ambos protocolos, IPv4 e IPv6, operan a la vez. Por lo tanto, es posible explorar exclusivamente el espacio de IPv4 para posteriormente intentar extraer información de IPv6 de los nodos descubiertos. Como en esta investigación estamos enfocados exclusivamente en el dominio de IPv6 no ahondaremos más en esto.

Básicamente, todos estas técnicas consisten en extraer la información desde otros puntos y son igualmente validos para IPv4 e IPv6. Además de estos métodos indirectos también se tienen las técnica de exploración en áreas locales o bien mediante la transmisión de mensajes de *NDP* con impacto limitado, los cuales ya son implementados en Nmap y se verán a continuación.

Al momento de la realización de esta investigación estaba disponible la versión 6.25 de Nmap. La cual permite las mismas exploraciones que puede ofrecer para IPv4. Solo existe una distinción importante: *Nmap no permite el escaneo de rangos de direcciones para IPv6*. Por ejemplo, Nmap puede recibir como argumento 10.0.0.0/8 y realizaría un escaneo para todo los nodos dentro de esa sub-red mientras que en IPv6 es necesario entregar un listado de *hosts* especifico. Posiblemente la razón se deba a que una sub-red típica en IPv6 puede ser: 2001:db8:c0ca:fea::/64 lo cual se traduce a una carga enorme de trabajo incapaz de realizarse en un tiempo adecuado.

Previo a la publicación de esta investigación se libero la versión 6.40 de Nmap, la cual ya permite el barrido en IPv6 por fuerza bruta. Este barrido de fuerza bruta sigue siendo completamente inefiente por las razones ya mencionadas. Sin embargo, la

carga de trabajo de exploraciones nativas de Nmap nos puede servir como punto de referencia al revisar las generadas por las técnicas desarrolladas por esta investigación.

El mecanismo de barrido de fuerza bruta es una mala forma de iniciar el descubrimiento de nodos, por lo tanto la comunidad de Nmap ha desarrollado técnicas escritas en guiones NSE que intentan conseguir información de nodos en IPv6 mediante las siguientes formas:

- Envió de mensajes ROUTER ADVERSTIMENT del protocolo NDP con dirección MAC origen generada de forma aleatoria.
- Se envía un mensaje ICMPv6 *Echo request* a la dirección multicast FF02::1que corresponde a *all-nodes link-local*
- Se envía un mensaje ICMPv6 incompleto a la dirección multicast FF02::1 para forzar a los nodos a responder acerca del error.
- Se envía un mensaje *multicast listener discovery* (MLD) incompleto a la dirección multicast FF02::1 para forzar a los nodos a responder acerca del error.

La mayoría consiste en poder mandar mensajes *multicast*, los cuales pueden ser restringidos a funcionar exclusivamente dentro de las redes internas, mediante el campo *Hop Limit* o incluso que no puedan ser enrutados fuera de las sub-redes, tal es el caso de los enrutadores de la marca Cisco, que no permiten el ruteo de paquetes con direcciones multicast por defecto. Otro elemento importante es que no todos los sistemas operativos implementan de forma similar las respuestas. Por ejemplo, los sistemas operativos de Micrsoft no responden a los mensajes *echo request*.

En pocas palabras,**Nmap ofrece herramientas capaces de detectar nodos en IPv6 de forma local pero carece de herramientas por defecto para poder hacer búsqueda en redes remotas**.

2.4. Mecanismos sugeridos

Existe una gran variedad de mecanismos para descubrir y extraer información de nodos existentes. Prácticamente todos ellos nacieron en IPv4 y la mayoría radican en las capa de aplicación del modelo TCP/IP permitiéndoles operar con pocas modificaciones entre los protocolos IPv4 e IPv6. Por lo mismo estas técnicas serán poco comentadas en esta investigación. **El enfoque deseado es con técnicas propias para IPv6, que**

radican justamente en la capa de dicho protocolo y no en de la de aplicación, salvo una técnica en particular.

El conjunto de técnicas y propuestas aquí sugeridas tienen como objetivo el detectar nodos en líneas en sub-redes remotas o bien, redes de otras compañías y varias son sugeridas inicialmente por Gont [5]. Estas técnicas han sido seleccionadas principalmente porque son de bajo riesgo para los nodos que están analizando al no intentar explotar ninguna vulnerabilidad conocida. Sin embargo, **Existe un riesgo de una negación de servicio accidental debido a que los Gateways de la sub-red objetivo pueden llegar a saturarse como resultado de anexar a sus buffers cada dirección que intentemos verificar**. Este riesgo en particular es exclusivo de IPv6 y puede ser temporal en que los registros del dispositivo se limpian; o bien el efecto puede no aparecer si el dispositivo maneja adecuadamente los buffers y registros para el control de los vecinos que tienen conectados directamente.

2.4.1. Descubriendo sub-redes

Propiamente dicho, esto no tiene como objetivo descubrir nodos sino detectar las sub-redes existentes dentro de una entidad. No existe una forma única, realmente todo las técnicas conocidas son aplicables para este escenario. En general, si estamos por ejemplo dentro de una entidad 2001:db8:c0ca::/48 nosotros podemos estar dentro de la sub-red
2001:db8:c0ca:fea::/64 o incluso 2001:db8:c0ca:fea::/96. En el primer caso, es valido suponer que otra sub-red podría ser 2001:db8:c0ca:feb::/64 pero en el segundo se requiere un poco más de información. Dicha información en ocasiones es accesible fácilmente y en otras se requiere aplicar una o más técnicas distintas. A continuación se sugiere un listado de técnicas que se pueden realizar con herramientas previamente conocidas:

1. Intentar obtener información de los enrutadores y sus protocolos de enrutamiento (intentando enviar mensajes fabricados o bien escuchando los mensajes del enrutador).

2. Escuchar el trafico de área de segmento local para detectar comunicación entre nodos dentro de la misma entidad pero en distintas sub-redes.

3. Suponer que se utilizan los *bits* 49-64 para sub-redes y los 64 *bits* restantes para *hosts*.

4. Técnicas de ingeniería social para extraer información.

5. Intentar acceder a los registros de servidores claves, tales como DNS, que permitan obtener el listado de los nodos que han utilizado los servidores. Dando acceso a un escaneo indirecto de la información que buscamos.

2.4.2. Barridos de nodos

De los 128 *bits* que se manejan en IPv6 para el dominio de Internet, los 3 bits más altos quedan fijos en el bloque 2000::/3, de los bits 4 al 64 quedan bajo control de los organismos indicados por la IANA para administrar a las entidades que soliciten direcciones. Una posibilidad es entregar prefijos de 48 bits a las entidades, por ej. 2001:db8:c0ca::/48. Cuando se maneja este esquema de 48 bits, los siguientes 16 *bits* ya se hallan bajo control de la entidad y tienen como propósito servir para la administración de sub-redes. Finalmente, los últimos 64 *bits* tienen como propósito identificar a los *hosts* en cada una de las sub-redes [32, 33].

Cuando no se maneja el esquema de bloques de 48 bits, la entidad puede recibir una cantidad menor o incluso una única sub-red con 64 bits. Para mayor información referirse a la figura 2.4 [32, 33].

Aún a pesar, que estamos solamente con la mitad del espacio de redes de 64 *bits* es un número que se mide en cuatrillones y resultaría en un gasto enorme de recursos realizar un barrido de fuerza bruta sobre estos *bits*. Por lo mismo se propone una variedad de técnicas que se pueden aplicar de forma modular para poder cubrir la mayoría de los escenarios de asignación de direcciones. De tal forma que se pueda especializar el esfuerzo de descubrimiento y reducir los tiempos de forma significativa.

2.4.2.1. Multicastng

Debido a las reglas de IPv6 muchos nodos pertenecen por defecto a uno o más grupos *Multicasting* [19], por lo tanto sería tentador utilizar direcciones *multicasting* para intentar obtener respuesta. Esto podría funcionar dentro del mismo segmento de área local, pero es probable que no funcione al intentar utilizarlo para otras sub-redes ya que existen enrutadores que por defecto descartan cualquier mensaje *Multicasting* que les llegue. Un claro ejemplo son los enrutadores de la marca Cisco, que al habilitar IPv6, una de las primeras entradas a su tabla de ruteo es "FF::/8" a la interfaz "*null*" que se traduce en no rutear ni un solo paquete *Multicasting*, evitando así su propagación por la red. Aún y cuando el campo de *hops* siga teniendo un valor valido.

Por esta limitante, se ha optado por no desarrollar ninguna herramienta en particular para fabricación de mensajes *multicasting* en esta investigación.

2.4.2.2. Escaneo de Low-bytes

Como se vio en las tablas 2.3 y 2.4, al parecer existe una cierta tendencia a utilizar los bytes más bajos de una dirección IPv6. Esta tendencia puede deberse a la simplicidad de escribir por ejemplo 2001:db8:c0ca::1/128 en lugar de `2001:db8:c0ca:0:aaaa:bbbb:cccc:dddd/128`. Durante los análisis del 2008 esta es una tendencia que estuvo más asociada en enrutadores. Técnicamente no es mala idea si la dirección de todas formas es accesible desde otros recursos, pues permite facilitar su escritura y recordarla pero en otros caos puede suponer un riesgo.

Los nodos que utilicen esta tipo de direcciones son configurados de forma manual. Por lo tanto, usualmente serán máquinas con funciones especificas tales como el *Gateway*, servidores u otros servicios. Se podría decir que existen 3 técnicas que caen en esta categoría [5]:

1. Utilizar valores bajos, tales como 1-255, por ejemplo 2001:db8:c0ca::1 ó 2001:db8:c0ca:222.

2. Representar el puerto de servicio al que responden. Por ejemplo 2001:db8:c0ca::80 para un servidor web.

3. Un híbrido, por ejemplo 2001:db8:c0ca::80:1 y 2001:db8:c0ca::80:2 para diferenciar dos servidor web en la misma sub-red.

Los tres escenarios se pueden resolver intentando un escaneo de fuerza bruta sobre los *bits* más bajos. Sin embargo, no es recomendable excedernos de 16 *bits*, pues se trataría de demasiada carga y se corre el riesgo de saturar el cache de los *Gateway* a los que están conectado los nodos que intentamos localizar. De esta forma, si deseamos utilizar este método sobre la sub-red 2001:db8:c0ca:fea::/64 haríamos un escaneo de 2001:db8:c0ca:fea::1 a 2001:db8:c0ca::ffff o un rango menor. La razón por la que se omite la primera dirección es que se trata usualmente de una dirección *anycast* a la que más de un enrutador podría contestar. Sin embargo, es opcional no emplearla, pues una dirección *anycast* es indistinta a una dirección *unicast* [19].

2.4.2.3. Escaneo en base SLAAC

Aunque el RFC respecto a SLAAC indica que pueden existir más de un mecanismo para la asignación de los 64 *bits* de *host*. A la fecha la mayoría de los sistemas operativos implementan solamente dos mecanismos: EUI-64 y direcciones temporales de Privacidad [34, 35]. En esta sección nos enfocaremos en EUI-64 utiliza los siguientes pasos:

1. Toma la dirección MAC de la tarjeta de red y lo separa en segmentos de 24 *bits* cada uno, llamemosles **MacH y MacL**, siendo el primero el que tenga los *bits* más significativos.

2. De los 64 *bits* de la porción de host, los 24 *bits* más significativos serán el segmento **MacH**.

3. Los siguientes 16 *bits* serán una constante con el valor hexadecimal de: 0xFFFE.

4. Los ultimos 24 *bits* corresponderán a **MacL**.

Como se puede observar, de los 64 *bits*, la búsqueda se reduce a 48 *bits*, es decir las direcciones MAC. Sin embargo, esta búsqueda aun se puede reducir más si tomamos en cuenta que tales direcciones están bajo control de la IANA. Los 24 *bits* más altos se denominan *Organizationally Unique Identifier* (OUI) [36, 5] e identifican a una compañía en particular. Mientras que los 24 *bits* más bajos están reservados para el uso interno de una compañía. Por lo tanto, si existiera la posibilidad de conocer de antemano el tipo de productos que maneja una empresa, es posible reducir la búsqueda de 48 *bits* a 24 *bits* por cada OUI controlado por una compañía.

Por ejemplo, asumamos que una empresa tiene contrato con DELL, ellos tienen registrados 49 OUI ante la IANA, por lo tanto, deberíamos explorar 2ˆ24 * 49 nodos, es decir, buscaríamos solamente 822,083,584 nodos - equivalente a menos del 1 % de los 48 *bits* - aun puede mejorar sustancialmente si especializamos más la búsqueda. De estas 49 OUI reservadas, muchas fueron asignadas hace muchos años atrás y es probable que la compañía ya no las utilice todas, y solo esté utilizando unas cuantas OUI. Es más, si las máquinas se vendieron en lote, es posible que todas pertenezcan a una misma OUI y al obtener la dirección de una sola máquina habremos podido reducir aún más la búsqueda, incluso a solo a 24 *bits* (16,777,216 de direcciones) que aunque sigue siendo un número grande es posible manipularlo en un tiempo adecuado.

Hay un factor también que también resulta importante, las máquinas virtuales cada vez ganan más terreno y se están volviendo un estándar común para la implementación de servidores. Estas máquinas virtuales, deben simular ser una máquina real para el Sistema Operativo anfitrión y entre los elementos que deben generar están las direcciones MAC. Estas se generan con ciertos OUI y algunas plataformas de máquinas virtuales tienen reglas estrictas para su asignación. Por ejemplo, varias plataformas populares de máquinas virtuales son: Parrallels, VMware, Virutal-Box y en menor medida Virtual PC para sistemas operativos de Microsoft y QEMU para sistemas operativos basados en GNU/Linux. Todas ellas tienen su propia OUI y se pueden manejar de forma similar a las máquinas sin virtualizar. Sin embargo, VMware ofrece información detallada y particularmente nos indica que en el caso de configuraciones dinámicas, 40 de

los 48 *bits* de MAC serán siempre fijos, permitiéndonos reducir aún más la búsqueda [5].

2.4.2.4. Escaneo en base a palabras

Esta técnica consiste en buscar direcciones de nodos que fueron configurados mediante juego de palabras. Tienen la ventaja que pueden llegar a ser fáciles de recordar. En IPv6 resulta útil debido a la notación hexadecimal que utilizan las direcciones, pues es posible escribir "palabras" como "C0CA", "CAFE", "BEEF", "DEAD" entre otras.

Por otro lado, también resulta sencillo explotarlo, estos juegos de palabras no son nada nuevo, al contrario, llevan existiendo desde hace tiempo y por consecuente existen una gran variedad de archivos que contienen grandes cantidades de "palabras", por lo tanto es posible generar un ataque denominado de diccionario, que pruebe cada una de estas palabras almacenadas como si de un *host* se tratase.

2.4.2.5. Escaneo en base Mapeo 4 a 6

Originalmente IPv6 tenía reservado un bloque especial que era ::FFFF:a:b:c:d/96 [19], con dicho bloque se permitiría la traducción directa de direcciones IPv4 a IPv6, o lo que se conoce como "mapeo". Sin embargo, este bloque quedo rápidamente en desuso y en las actuales metodologías las direcciones IPv4 son "mapedas" (escritas) directamente sobre la dirección que maneje la entidad. Por ejemplo, la dirección IPv4 192.168.10.10 , en una sub-red IPv6 2001:db8:c0ca:fea::/64 sería representada como 2001:db8:c0ca:fea::192.168.10.10. Observe que se cambio la sintaxis, la parte que correspondía a IPv4 se maneja aún con un punto en vez de dos puntos. Es meramente una representación visual para el ser humano. En realidad, la dirección IPv4 será convertida a una notación hexadecimal.

Capítulo 3

Descubrimientos de nodos en IPv6

En este capítulo se desarrollan las técnicas propuestas anteriormente, cada una de ellas esta escrita como un guión NSE para Nmap, de tal forma que se puedan utilizar de manera independiente o en conjunto con las demás. Todo lo desarrollado en esta investigación fue utilizando la versión 6.25 de Nmap. Durante las pruebas, se liberó la versión 6.40 y los guiones NSE fueron probados satisfactoriamente en dicha versión.

3.1. Técnicas desarrolladas

De las técnicas propuestas en el capítulo anterior, se han escogidos aquellas que Nmap todavía no manejaba de forma directa, o bien que no existiesen elementos ya publicado de ellos. Todo fue desarrollado en guiones de NSE y por lo tanto se requieren los argumentos para pasar una o más variables a los guiones NSE durante la ejecución del programa Nmap.

Cada una de estas técnicas puede ser invocada por el usuario de forma modular, y cada una de ella puede recibir diferentes argumentos para permitir mayor flexibilidad según lo que se requiera. En el apéndice C.1 se encuentran escritos todos los argumentos posibles que podrían recibir estos *scripts*.

3.1.1. Enumeración de sub-redes: DHCPv6

DHCPv6 y su contraparte para IPv4 es un punto delicado, pues permite obtener bastante información de los nodos, particularmente de DHCPv6 por el concepto de DUID que permitiría conocer el historial especifico de un nodo a lo argo de sus conexiones, dentro de la entidad con el propósito de poderlo monitorear [38, 50]. Por lo mismo, DHCPv6 obtiene muchos elementos de seguridad para evitar la interferencia con los mensajes entre clientes y servidores, o bien engañar directamente al servidor fabricando DUID reales. Sin embargo, existe una forma de explotar DHCPv6 hasta

cierto punto para obtener información de sub-redes.

Debido a que la comunicación entre clientes y servidores debe ser transparente, un agente de paso (*relay*), tomará el mensaje de otro agente y lo pasará sin revisarlo. Esto nos permite la posibilidad de fabricar mensajes de falsos agentes. El contenido de dichos mensajes consistirá en la petición de asignación de una dirección de red de un *host* ficticio, siguiendo todas las pautas. Debido al hecho que la transmisión de mensajes entre agentes de paso es mediante multicast, esta garantizado que el primer agente real tomará el paquete y lo pasará hacia los demás agentes hasta llegar al servidor. El servidor tomará la petición del *host*, pero además la dirección *unicast* del primer agente (el fabricado por nosotros) para decidir de que sub-red proviene,0 y entonces responderá al cliente enviando el segundo mensaje de una fase de 4 de ellos.

Es justamente aquí donde radica la propuesta de está técnica de nuestra auditoría, nuestro agente falso intentará parecer miembro de otra sub-red, si esta existe el servidor nos responderá con el segundo de cuatros mensajes de configuración y con ello, obtendremos una respuesta válida de que la sub-red efectivamente existe. En caso contrario, el servidor simplemente no responderá. Además, suspender la comunicación en este punto no perjudica a la capacidad del servidor, pues no es hasta el cuarto mensaje cuando realmente reserva la dirección que ofrece, como se puede apreciar en la figura 2.6.

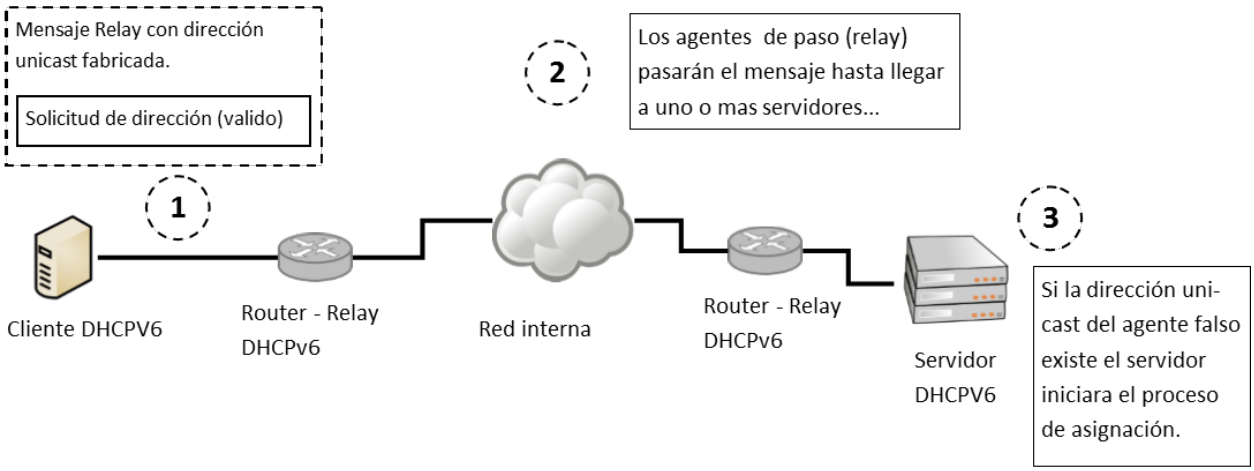

Figura 3.1: Técnica propuesta para adquirir información de sub-redes mediante DHCPv6.

Una configuración por defecto de DHCPv6 permitiría la realización de esta búsqueda de información que resulta mucho más práctico que fabricar DUID pre-existentes, aunque una red empresarial que coloque alguno de los mecanismo de seguridad sugeridos por el mismo RFC, tales como autenticación entre agentes de paso y servidores

podrían terminar de invalidar esta técnica [38].

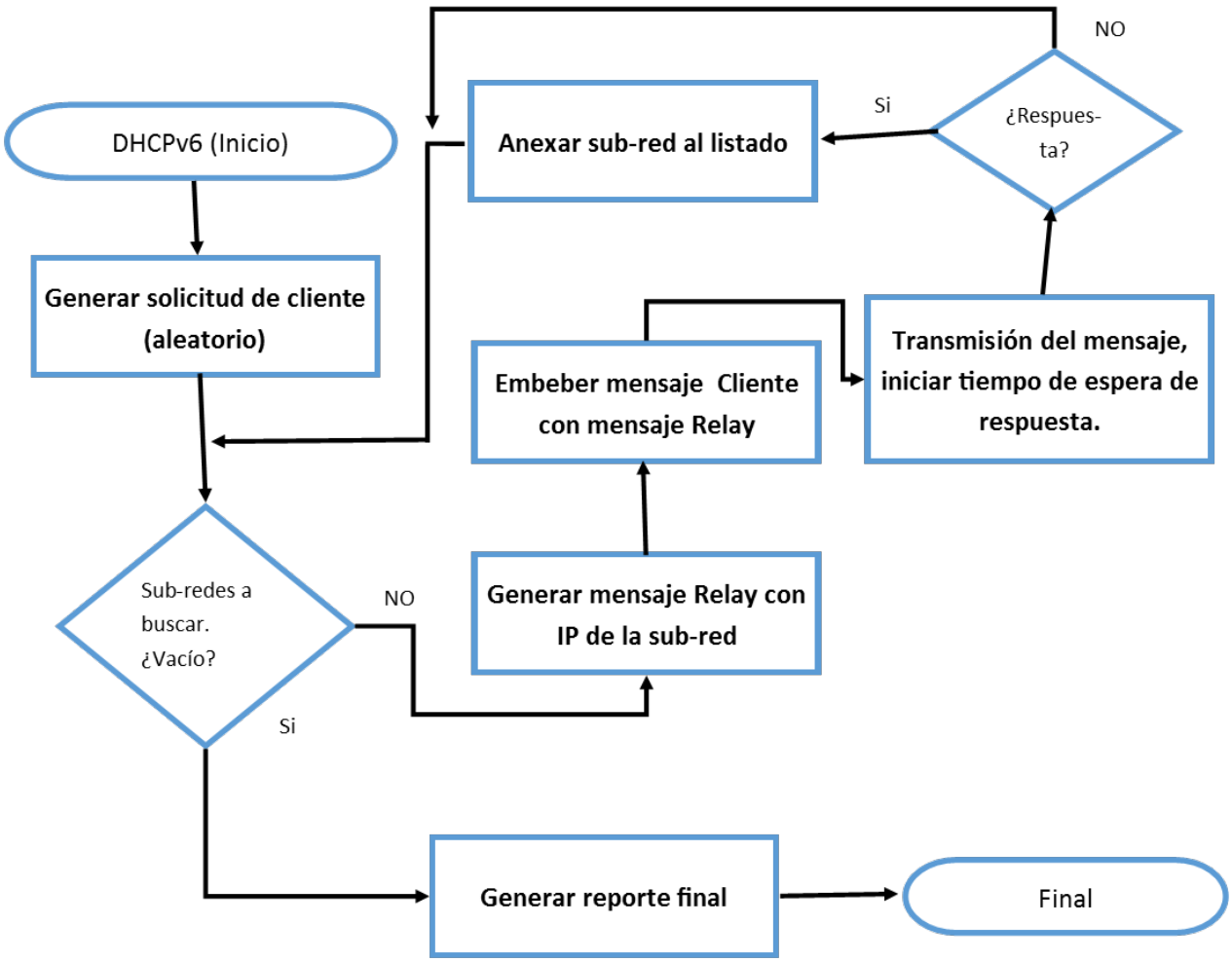

Figura 3.2: Diagrama de Escaneo de sub-redes: DHCPv6

Para el desarrollo de está técnica en Nmap, el guión se ejecuta exclusivamente en la fase de pre-scanning. **Su objetivo es mediante mensajes fabricados para servidores DHCPv6, validar la existencia de sub-redes IPv6** proporcionadas por el usuario mediante el argumento `itsismx-dhcpv6.subnet` , y una vez confirmado añadirlos a un registro especial - `nmap.registry.itsismx.PrefixesKnown` - para ser utilizados por todo los demás guiones enfocados en la búsqueda de nodos. Por lo mismo, este guión es el menos invasivo de los aquí propuesto y al no completar los 4 mensajes no le quita memoria a los servidores.

La figura 3.1 muestra la idea detrás de fabricar mensajes de un agente *relay* con uno o más campos variables. Sin embargo, el ultimo campo del mensaje a duplicar

se trata de un mensaje de solicitud de un cliente, es decir, un mensaje encapsulado dentro de otro. Así que este guión fabricará ambos mensajes, nosotros generamos un cliente fantasma "válido". Es decir, que los campos del cliente estarán con un formato y contenido válidos, pero no se intentará duplicar a ningún *host* real.

El campo más importante a crear para el mensaje del cliente es el DUID, este puede estar conformado de 3 maneras distintas y de longitudes variadas, pero solo uno corresponde a nodos típicos de usuarios con laptops, computadoras de escritorio y/o dispositivo móviles. Este DUID esta conformado de la siguiente forma [38]:

1. 16 *bits* que poseen el valor de 1, que indica que se trata de un DUID-LTT (plus time) definido en el RFC 3315.
2. 16 *bits* del tipo de hardware. Específicamente le hemos asignado Ethernet (0x0001).
3. 32 *bits* de tiempo. Fecha en que se genero el DUID por primera vez a partir de Enero 1, del 2000 con un modulo de 32 *bits*.
4. Una dirección IPv6 de alcance local representando a nuestro nodo falso.

Debido a la enorme cantidad de *bits* que utilizan los DUID se opto por fabricar el mensaje conservando el formato válido. El tiempo de creación se asigna al vuelo, haciendo que los DUID cambien en cada ejecución del *script*. La dirección IPv6 de enlace local por defecto es la misma del nodo que fabrica el mensaje. Particularmente, porque la respuesta del servidor es enviada de forma *unicast* y por lo mismo se debe de responder a un mensaje *NDP (Neighbor Solicitation)* antes de recibir la respuesta. Mediante el argumento `itsismx-dhcpv6.Spoofed_IPv6Address` es posible indicar otra dirección pero es requerido poder responder adecuadamente al mensaje NDP.

La dirección del nodo fantasma pertenece por defecto a una OUID de DELL de tal forma que la dirección oscilará entre FE80::24B6:FDFF:FE00:00 a FE80::24B6:FDFF:FEFF:FF. Es posible utilizar el argumento
`itsismx-dhcpv6.Company` para utilizar un OUID especifico, pero los últimos 24 *bits* seguirán siendo generados de forma aleatoria.

Con la DUID formada generamos una solicitud de *host*, el cual tiene la siguiente composición:

1. 8 *bits* indicando el tipo de mensaje. Valor estático de 0x01 que representa SOLICIT.
2. 24 *bits* que corresponden al campo *transacation-id*. Será fabricado de forma aleatoria.

3. 224 *bits* de opción de DHCP denominado *Client Identifier Option* el cual consiste en:

   a) 16 *bits* constantes de: 0x0001.

   b) 16 *bits* indicando la longitud del DUID generado.

   c) 192 *bits* del DUID.

4. 288 *bits* de opción de DHCP denominado *IA-TA option* el cual consiste en:

   a) 16 *bits* constantes de: 0x0004.

   b) 16 *bits* indicando la longitud de esta opción.

   c) 32 *bits* indicando un ID de IA generado aleatoriamente.

   d) 224 *bits* de un campo de de opción de IA que a su vez esta conformado por:

      1) 16 *bits* con el valor : 0x005
      2) 16 *bits* indicando la longitud de esta opción.
      3) 128 *bits* de dirección de enlace local (debe ser la misma que la del DUID).
      4) 32 *bits* para el campo *preferred-lifetime* que contendrá el valor de cero.
      5) 32 *bits* para el campo *valid-lifetime* que contendrá el valor de cero.

5. 48 *bits* de opción de DHCP denominado *Elapsed Time* el cual consiste en:

   a) 16 *bits* constantes de: 0x0008.

   b) 16 *bits* indicando la longitud del mensaje, el cual es fijo y por lo mismo será 0x002.

   c) 16 *bits* con el valor de cero o bien uno proporcionado mediante el argumento `itsismx-dhcpv6.TimeToBeg`.

Este mensaje fabricado tendrá un tamaño fijo de 592 *bits* y estará incrustado dentro de otro mensaje que posee la siguiente estructura [38]:

1. 8 *bits* con el valor: 0x0c para indicar que es un mensaje del tipo *Relay-Forward*.

2. 8 *bits* para indicar el numero de *hop-counts*. Para propósito de esta investigación, el valor será constante de: 0x02

3. 128 *bits* que simulan ser la dirección *unicast* de alcance global (o cualquiera que no sea Local) del agente *relay*. Por defecto es 2001:db8:c0ca::1 pero puede ser cambiado por el argumento `sitsismx-dhcpv6.subnets`.

4. 128 *bits* que corresponden a la dirección de enlace-local del cliente fantasma previamente generado.

5. 624 *bits* formados de un cuerpo de opciones de agente *relay* el cual contiene la siguiente estructura:

   a) 16 *bits* constantes de: 0x000.

   b) 16 *bits* indicando la longitud del mensaje de solicitud del *host*

   c) 592 *bits* del mensaje de solicitud de *host* previamente generado.

Opcional al mensaje IA-TA (que corresponde a solicitar direcciones de privacidad) se puede utilizar el argumento `itsismx-dhcpv6.IA_NA` para solicitar una dirección IA-NA, que consiste en direcciones *estatful* y son las más parecidas a las solicitudes de la contraparte de IPv4. Opcionalmente existe un argumento `itsismx-dhcpv6.Option_Request` que permite agregar una solicitud de opción de dominio. Esto para fabricar un mensaje lo más fiel posible a solicitudes de clientes reales, pero es un campo que no se utiliza para validar la respuesta.

Cabe aclarar, que se generara un mensaje de agente *relay* por cada sub-red proveida mediante el argumento `itsismx-dhcpv6.subnets`. Por defecto, cada mensaje se envía en un tiempo aleatorio de 1 a 200 microsegundos pero es posible incrementar el rango proporcionando un nuevo valor mediante el argumento `itsismx-dhcpv6.utime`. Una vez que comience a transmitir los argumentos se estará también prestando atención a la captura de los mensajes.

La respuesta que podamos obtener dependerá de la versión de servidor que se utilizando. En esta investigación se utilizo el servidor `wide-dhcpv6-server` para Sistemas Operativos GNU/Linux y el DHCPv6 de Windows Server 2012. El primero, da respuestas laxas y al configurar piscinas de direcciones por interfaz es posible que nos envié falsos positivos a nuestras sondas. En el caso del servidor implementado por Microsoft, este verifica la dirección del agente *relay* contra cada una de sus piscinas, dando el comportamiento teórico que esperamos. Si existe una piscina que concuerde mandará una respuesta, en caso contrario se abstendrá de enviar más mensajes.

3.1.2. Enumeración de nodos: Low-bytes

Este guión se ejecuta en la fase de *pre-scanning* con el objetivo de generar un rango de direcciones de *hosts* a buscar, basados en asignaciones de los bytes más bajos y también en la fase de escaneo para validar los *hosts* en línea que fueron descubiertos mediante está técnica, tal como se aprecia en la figura 3.3. La técnica se trata de fuerza bruta, por defecto realizará la búsqueda de 8 *bits*, pero es posible cambiar el rango mediante el argumento `itsismx-LowByt.nbits` de 1 a 16 *bits*. El escaneo de fuerza

bruta lo realizará por cada sub-red proporcionada por el *script* `itsismx-dhcpv6` o bien mediante el argumento `itsismx-subnet`.

Como ejemplo, si se proporciona la sub-red 2001:db8:c0ca::/64 por defecto agregara a la lista de *hosts* las direcciones 2001:db8:c0ca::1 a 2001:db8:c0ca::FF, y si se cambia el parámetro puede recorrerlos hasta 2001:db8:c0ca::FFFF, es decir 65,536 direcciones. Este guión corre un pequeño riesgo de saturar *buffers* de *Gateways* si estos no los manejan adecuadamente.

Por ultimo, existe un mecanismo de control mediante el argumento `itsismx-LowByt.OverrideLock` que si es proporcionado, el argumento `itsismx-LowByt.nbits` cambiará su restricción a 3 a 64 *bits*, convirtiendo este guión en un ataque de fuerza bruta completo. **No es factible realizar un escaneo de 64 *bits***. Además, con la liberación de la versión 6.40 de Nmap se duplica la capacidad de este barrido.

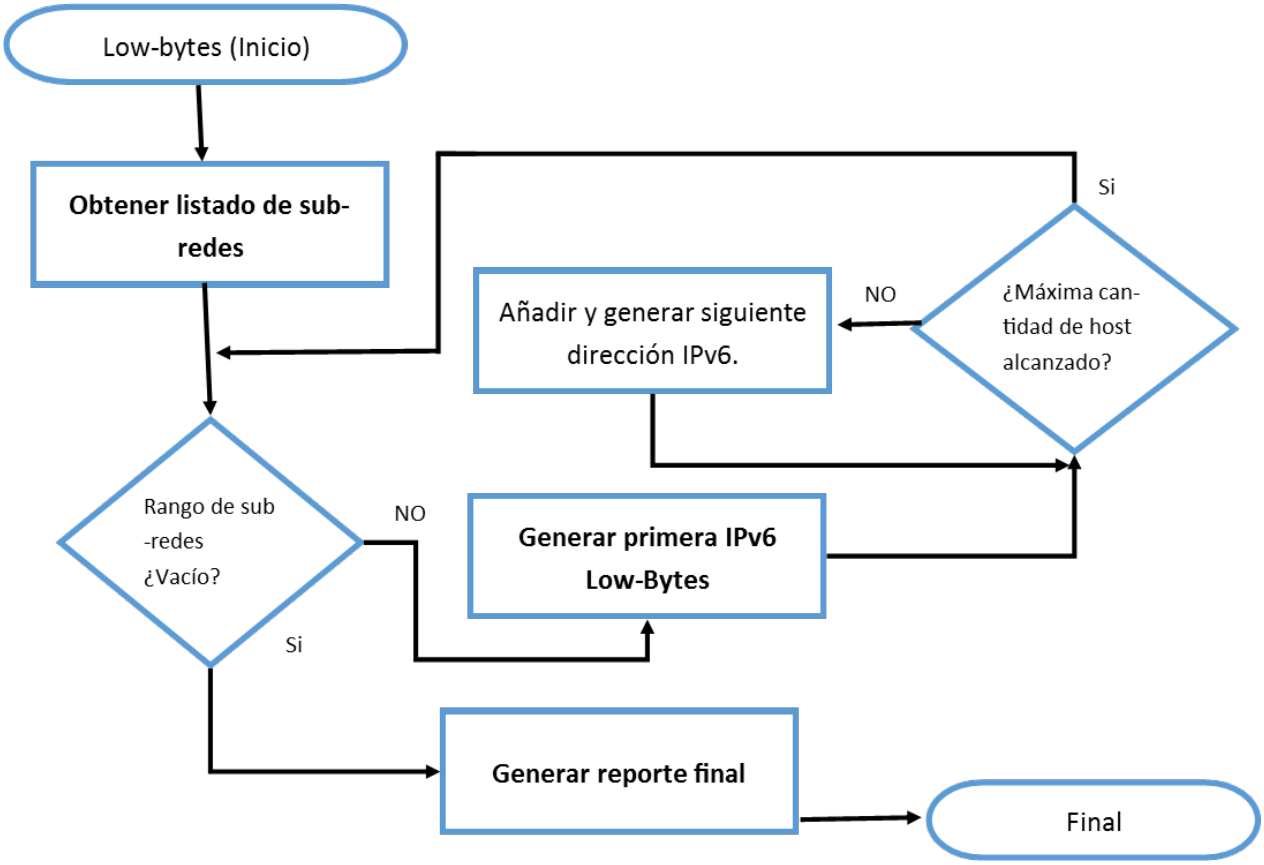

Figura 3.3: Diagrama de Escaneo de nodos: Low-bytes (*bits* bajos)

3.1.3. Enumeración de nodos: SLAAC

Este guión se ejecuta en la fase de pre-scanning con el objetivo de generar un rango de direcciones de *hosts* a buscar, basados en el formato EUI-64, y también en la fase de escaneo para validar los *hosts* en línea que fueron descubiertos mediante está técnica, tal como se aprecia en el diagrama de flujo de la figura 3.5. El mecanismo de EUI-64 queda explicado en la figura 3.4. En general, está técnica reduce la búsqueda de fuerza bruta a 24 *bits*, por cada OUI que se desee explorar, pero esto sería contra-producente pues hablamos de múltiplos de 16,777,216 de direcciones. Debido a ello por defecto se buscan de forma aleatoria o secuencial 11 *bits* de los 24, reduciendo el rango de busqueda a 4,096 por cada OUI que se desee buscar para cada sub-red proporcionada por el *script* `itsismx-dhcpv6` o bien mediante el argumento `itsismx-subnet`.

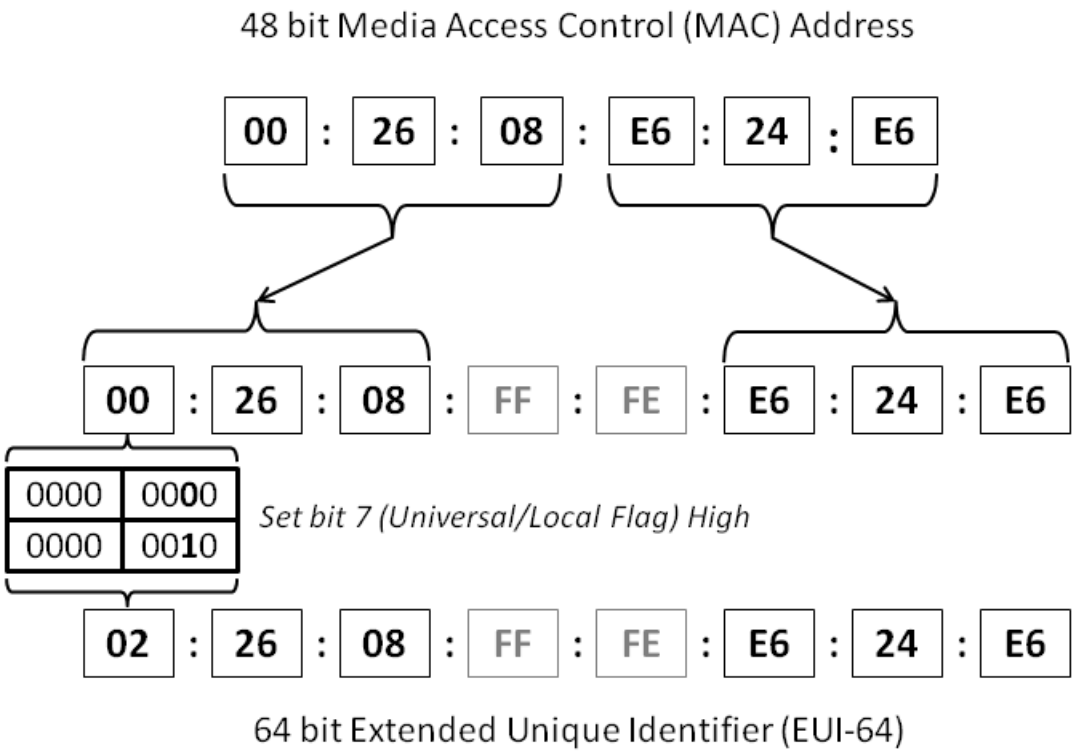

Figura 3.4: Formato de configuración de una dirección usando EUI-64.

El argumento `itsismx-slaac.nbits` permite cambiar la cantidad de los 24 *bits* que se desee explorar por cada OUID. El `script` también puede diferenciar entre generar dichos *bits* de forma aleatoria o bien de forma secuencial mediante el argumento `itsismx-slaac.nbits`, el cual también altera la forma de interpretar el argumento `itsismx-slaac.nbits` de la siguiente forma: Si el mecanismo es secuencial, los *nbits* indicarán cuantos de los *bits* menos significativos serán barridos. En cambio, si el me-

canismo es aleatorio, *nbits* representara la cantidad de muestras aleatorias de 24 *bits* que se generaran. Si la cantidad de *bits* es igual o superior a 21, entonces se genera un barrido de fuerza bruta exclusivamente.

Como ejemplo, consideremos que tenemos como sub-red 2001:db8:c0ca::/64 y la parte alta del EUI-64 queda en 0226:08FF:FE, y *nbits* es proporcionada con el valor de 2. Si el mecanismo es secuencial, las direcciones que generará son:
2001:db8:c0ca:0226:08FF:FE00:0, 2001:db8:c0ca:0226:08FF:FE00:1,
2001:db8:c0ca:0226:08FF:FE00:2 y 2001:db8:c0ca:0226:08FF:FE00:3. En cambio, si el mecanismo que se proporciona es aleatorio, un posible resultado será:
2001:DB8:C0CA:0226:08FF:FED0:8345, 2001:DB8:C0CA:0226:08FF:FE00:FF,
2001:DB8:C0CA:0226:08FF:FE00:1 y 2001:DB8:C0CA:0226:08FF:FEFF:FFFF.

La elección del OUI es fundamental. Si el usuario conoce algo de los equipos de computo de la entidad objetivo, puede reducir su rango de búsqueda, por ejemplo, si el usuario conoce que los equipos de computo son de la marca DELL entonces puede utilizar el argumento *vendors* introduciendo "Dell", y obtendrá todos los OUI registrados a dicha compañía. Sin embargo, en este ejemplo hablamos de 49 registros. El usuario puede indicar el OUI manualmente. Como ejemplo, lo siguiente sería una entrada válida para el argumento: `(5855CA, 6C9B02, 0CD292, Dell, Intel)`. En general, **qué tan eficiente sea el uso de este guión, dependerá directamente de que tanta información posea el usuario de los productos físicos de las redes que esta explorando**.

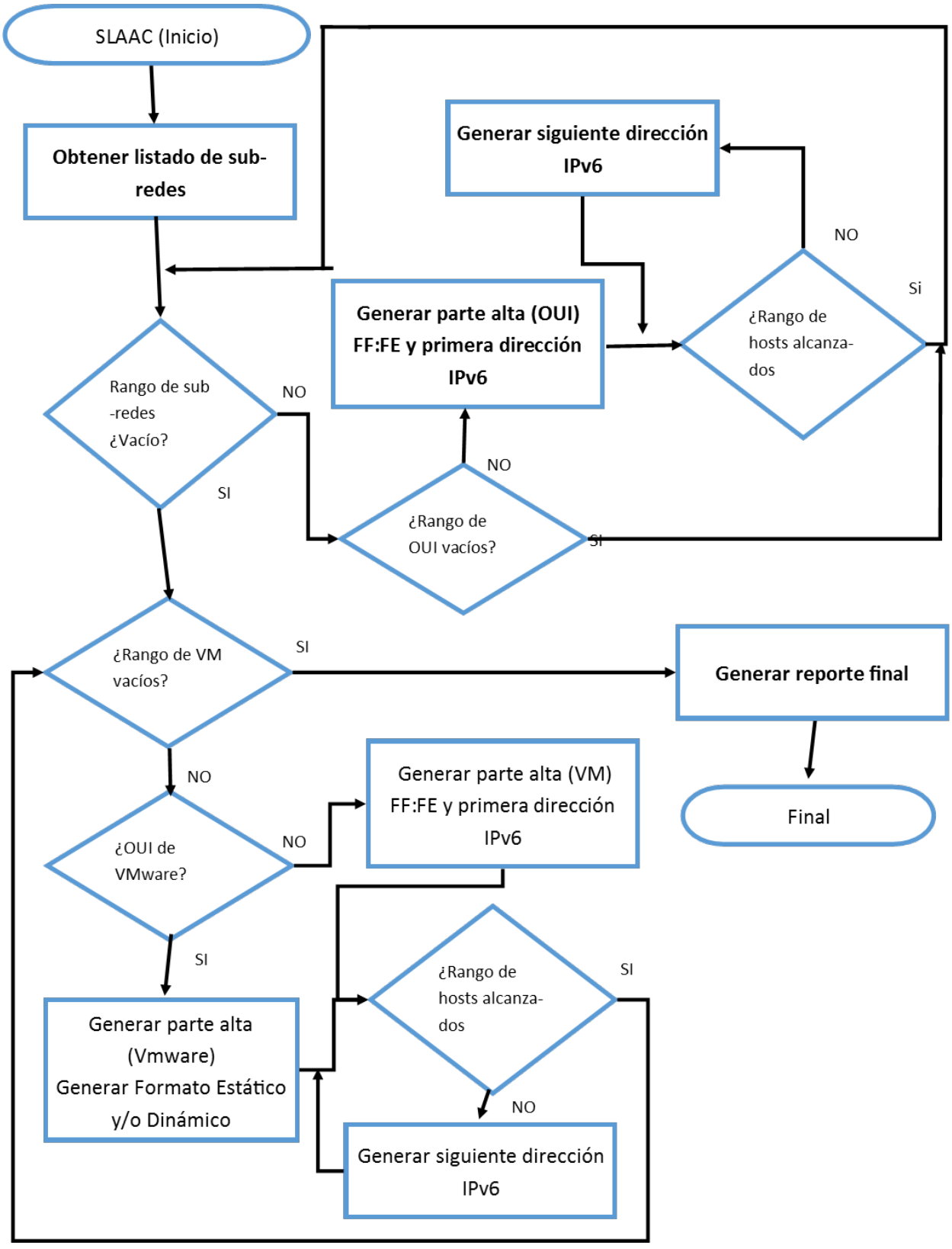

Figura 3.5: Diagrama de Escaneo de nodos: SLAAC (y máquinas virtuales)

Este guión también opera directamente con máquinas virtuales, de forma simple se le podría indicar al guión, mediante el argumento `vendors`, los OUI de las compañías que desarrollan máquinas virtuales. Sin embargo, algunas de estas compañías no utilizan todas estas OUI para sus máquinas virtuales y/o tienen reglas especiales. El mejor caso para esto, es la compañía VMware que posee 4 OUI pero solamente utiliza 2 para sus distintos productos de virtualización. Estos OUI tienen sus propias reglas de configuración según sea Manual (00:50:56:XX:YY:ZZ) o dinámica (00:0C:29:WW:TT:UU).

La configuración manual es cuando el administrador del equipo virtual decide colocar una dirección MAC de forma "manual", pero esta limitado a siempre estar dentro de ese rango,de hecho, por cuestiones de diseño interior, de los 48 *bits*, 30 se hallan fijos haciendo que la búsqueda solo requiera ir de de 3 a 18 *bits* como máximo [51].

La configuración dinámica tiene reglas interesantes, los primeros 24 *bits* son conocidos, los siguientes 16 *bits* también lo son: pertenecen a los 16 *bits* más bajos de la dirección IPv4 que posea la máquina virtual. Dejando los últimos 8 *bits* como un valor aleatorio nacido de un *hash* al nombre de la máquina [51].

El guión maneja una configuración especial donde es posible proveer direcciones IPv4 que son tentativas para el servidor, esto mediante el argumento `itsismx-slaac.vmipv4` y como resultado solo requiere hacer un barrido de los últimos 8 *bits*. En caso que este no sea proporcionado, generará muestras aleatorias de los 16 *bits* (a menos que el mecanismo elegido sea secuencial) y por cada uno de ellos generará el barrido de los últimos 8 *bits*.

Existen otras máquinas virtuales sin embargo, solo VMware da una explicación de cómo opera la asignación de sus direcciones y las demás, al parecer, hacen algún tipo asignación de número aleatorio de los 24 *bits* más bajos, permitiendo usar las técnicas ya mencionadas previamente. Sin embargo, para facilitar la operación el usuario, puede introducir el argumento `itsismx.slaac.vms` con los valores mostrados en el apéndice correspondiente.

El guión permite buscar las máquinas virtuales con sus argumentos paralelamente a los OUI de proveedores reales, volviéndose una herramienta bastante completa, por supuesto que para hacer más eficiente la búsqueda, es requerido conocer las direcciones IPv4 de los que creemos son máquinas virtuales, algo que el mismo Nmap puede realizar por sí mismo en un paso previo.

3.1.4. Enumeración de nodos: Palabras

Este guión se ejecuta en la fase de *pre-scanning* con el objetivo de generar un rango de direcciones de *hosts* a buscar, basados de un diccionario , y también en la fase de *scanning* para validar los nodos en línea que fueron descubiertos mediante está técnica. El proceso a seguir esta en el diagrama de flujo de la figura 3.6. Y en síntesis, se trata de hallar direcciones IPv6 que estén creadas a partir de palabras, las cuales están previamente registradas en un diccionario de palabras. Dicho diccionario esta dividido en 3 columnas que contiene la siguiente información:

1. Numero de segmentos de 16 *bits* que alberga.
2. Representación ASCII de la palabra.
3. Representación binaria de la palabra.

Por defecto el *script* tomará cada prefijo conocido y probara cada una de las palabras contenidas en el archivo diccionario. El usuario puede obtener un poco más de control al utilizar el argumento `itsismx-wordis.nsegments` con el cual puede definir que tan grande serían las palabras a buscar. Con las palabras que son menores a 4 segmentos, queda la necesidad de rellenar el espacio faltante con ceros, y se puede decidir si esto sería a la izquierda o derecha mediante el argumento `itsismx-wordis.fillright`. Por ejemplo, para un prefijo 2001:db8:c0ca::/64 se puede tener como dirección : 2001:db8:c0ca::dead:bef o 2001:db8:c0ca:dead:bef::0.

Naturalmente, la eficiencia de este guión dependerá directamente del tamaño del diccionario, pero las palabras escritas con caracteres hexadecimales han sido usadas por mucho tiempo, por lo tanto es relativamente fácil crear un diccionario de buen tamaño para incrementar las posibilidades de éxito.

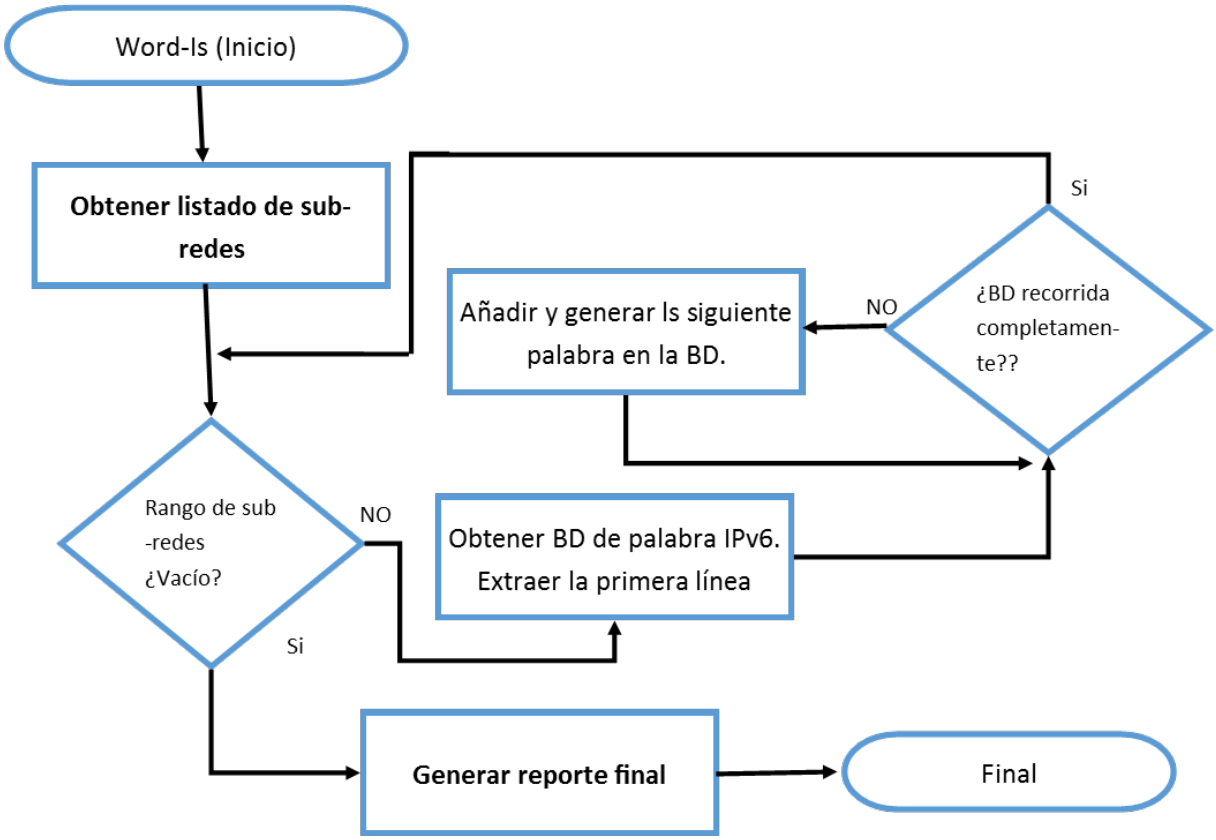

Figura 3.6: Diagrama de Escaneo de nodos: Palabras

3.1.5. Enumeración de nodos: Mapeo 4 a 6

Como ya se explicó en el capitulo anterior, el objetivo es revisar si una red que se adapto a IPv6 recibió una transición casi literal de su espacio de IPv4 a IPv6, el modo de trabajo está detallado en el diagrama de flujo en la figura 3.7. Es importante recalcar que esta técnica no tiene como objetivo utilizar la dirección ::FFFF:a.b.c.d, que ya se halla depreciada, sino que el objetivo es convertir direcciones IPv4, previamente descubiertas, a IPv6 para cada prefijo que se conozca. Por ejemplo, si se conoce el prefijo 2001:db8:c0ca:fea::/64 y las direcciones 192.168.1.1 y 192.168.1.2 el guión generará las siguientes direcciones: 2001:db8:c0ca:fea::192.168.1.1 y 2001:db8:c0ca:fea::192.168.1.2. Es importante recalcar, que para motivos prácticos, hemos combinado ambas notaciones, pero el *script* regresará los valores de IPv4 ya en hexadecimal, por ejemplo, 2001:db8:c0ca:fea::c0a8:0101.

Mediante el argumento `itsismx-Map4t6.IPv4Hosts` el usuario puede introducir un listado de *hosts* específicos (192.168.1.1) y/o sub-reds (10.0.0.0/24) a explorar. La mayor eficiencia vendría si los nodos previamente se detectan como activos. Lo ideal,

seria buscar los nodos primero en IPv4 y si se encuentra en línea entonces buscarlo en IPv6. Sin embargo, Nmap solo permite operar en uno de los dos protocolos a la vez, por lo tanto, si se quiere trabajar solo nodos que se conozca estén en línea deberán ejecutar por separado las pruebas en IPv4.

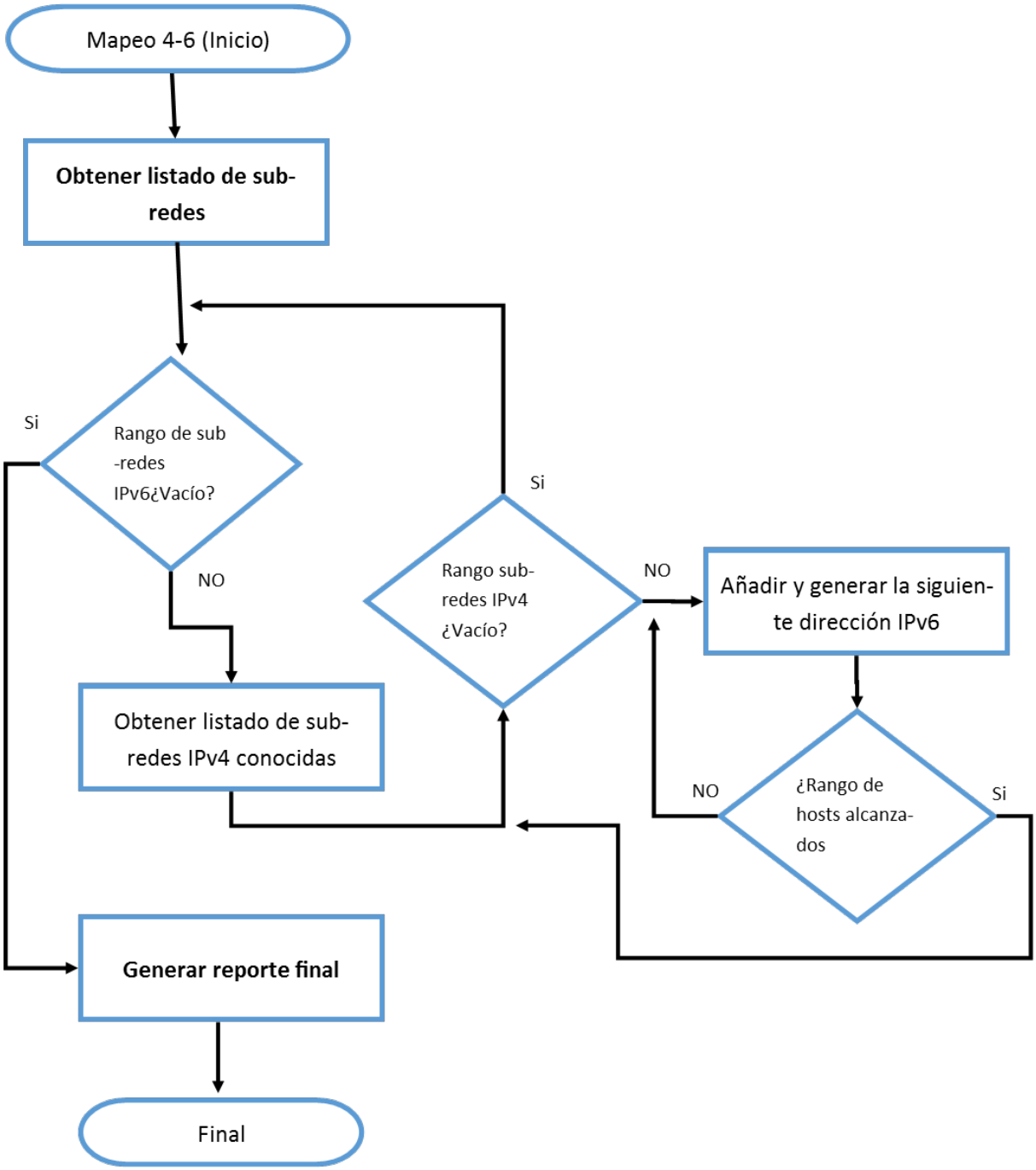

Figura 3.7: Diagrama de Escaneo de nodos: Mapeo de direcciones IPv4 a IPv6

El tiempo de ejecución de este guión puede variar según la cantidad de nodos introducidos mediante el argumento `itsismx-Map4t6.IPv4Hosts`.

3.1.6. Reporte final

Se trata llanamente de un reporte del total de nodos y sub-redes descubiertos mediante las técnicas ya desarrollada. Toma de cada una de ellas, el listado de nodos que validaron su existencia en línea e indica la eficiencia de cada una de las técnicas respecto al total de nodos descubiertos. Despliega por separado las técnicas destinadas al descubrimiento de sub-redes y los de *host*.

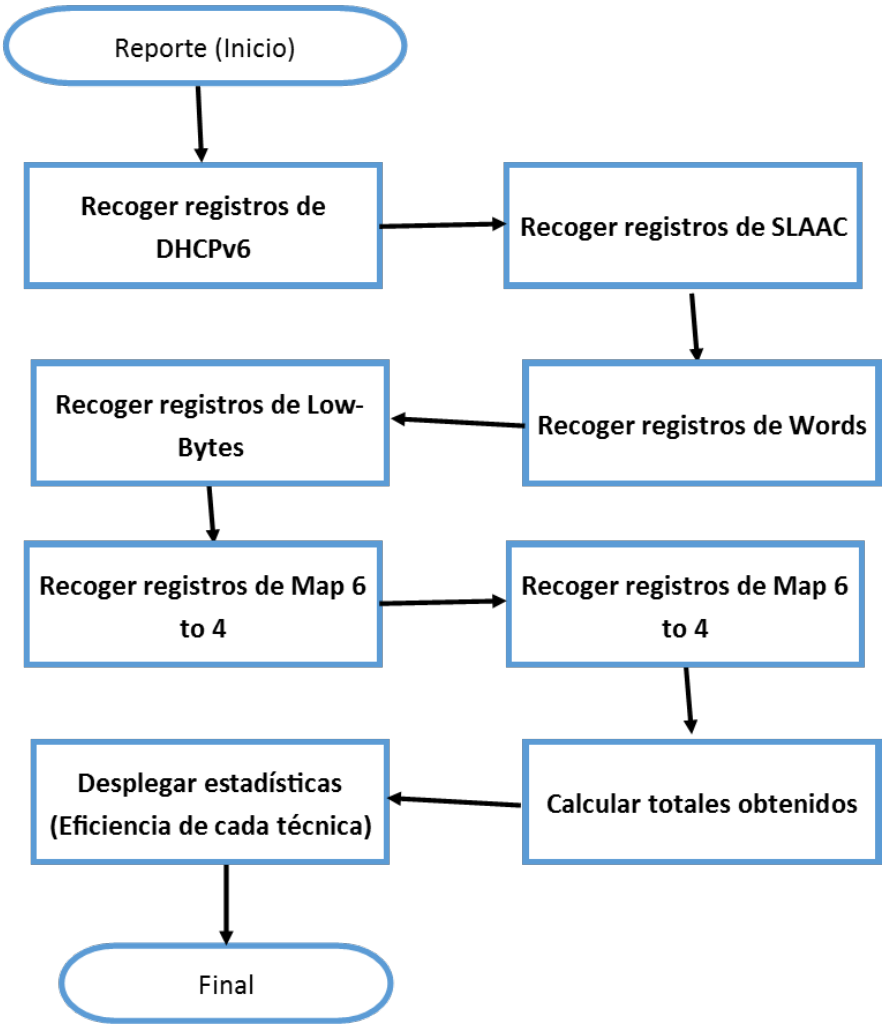

Figura 3.8: Reporte final

3.2. Ejecución de los guiones

Como se mencionó al inicio de este capítulo, es posible ejecutar de forma modular cada uno de estos guiones mediante Nmap. De esta forma se pueden ejecutar experimentos complejos simultáneamente o por separado. La siguiente serie de comandos de Nmap ejecutan diferentes módulos:

1. `nmap -6 -v -e eth0 --script itsismx* --script-args itsismx-dhcpv6.subnets=2001:db8:c0ca::/64, itsismx-Map4t6.IPv4Hosts=192.168.1.0/24,newtargets`
2. `nmap -6 -v -e eth0 --script itsismx-slaac,itsismx-dhcpv6 --script-args itsismx-dhcpv6.subnets=2001:db8:c0ca::/64,newtargets`
3. `nmap -6 -v -e eth0 --script itsismx-words-known,itsismx-map4to6.nse --script-args itsismx-dhcpv6.subnets=2001:db8:c0ca::/64, itsismx-Map4t6.IPv4Hosts=192.168.1.0/24,newtargets`
4. `nmap -6 -v -e eth0 -A -sC --script itsismx* --script-args itsismx-subnets=2001:db8:c0ca::/64, itsismx-Map4t6.IPv4Hosts=192.168.1.0/24,newtargets`

El primer comando ejecuta todos los *scripts* desarrollados en esta investigación, con los argumentos a NSE mínimos para su correcta ejecución. Los comandos 2 y 3 son ejemplos de la ejecución selectiva de los guiones desarrollados, mientras que la ultima opción muestra la capacidad modular de Nmap, ya que a cada nodo descubierto durante la ejecución se le añade el barrido de puertos, detección de sistema operativo y todos los *scripts* pertinentes a la categoría "*default*" de Nmap.

Para exploraciones rápidas, puede convenir utilizar las técnicas de *low-bytes*, además de los basados en palabras ya que suelen a ser más utilizados para servidores y dispositivos intermedios, tal como se vio en la tabla 2.4. El *script* basado en "'mapeo" de IPv4 a IPv6, es funcional si previamente se conoce que la red de la entidad maneja ese esquema, que puede llegar a ser útil en entidades que solo posean un único bloque de 64 bits. Además, en caso de ser utilizado este esquema de configuración, seguramente se asignarán las direcciones a los nodos mediante servidores DHCPv6 *stateful* o en su defecto manualmente. La principal característica de este esquema es que tendrán el siguiente comportamiento en sub-redes: X:X:X:X::0000:0000 - X:X:X:X::FFFF:FFFF con una variación de prefijos de 96 a 126 *bits*.

Respecto a la técnica de DHCPv6 esta puede ser utilizada, incluso de forma local, para descubrir la existencia de servidores DHCPv6. La primera prueba puede ser directamente con la dirección que el *host* atacante reciba, esto con la intención de validar

la existencia previa de un servidor DHCPv6. Si hay agentes de pasos estos tomarán el mensaje, si esta el servidor en la misma red, él responderá. En caso de una nula respuesta, puede significar la no existencia de DHCPv6, o bien que los mensajes de agentes de paso estén protegidos mediante cifrado [38].

Las exploraciones basadas en SLAAC, ya sea para máquinas físicas o servidores hospedados en máquinas virtuales, deben ser manejados de forma cuidadosa, ya que la confiabilidad de los resultados dependerá de la cantidad de bits a explorar, y esto repercutirá fuertemente en el tiempo que se requiera para su exploración. Como se mencionaba en el sub-capítulo correspondiente, está técnica será más útil, si previo al escaneo se realiza una investigación de fondo, posiblemente con interacción humana, para intentar determinar modelos de computadoras y tipos de máquinas virtuales implementados en la entidad objetivo.

Nmap es ejecutable en una gran variedad de sistemas operativos, principalmente GNU/Linux, Microsoft y MAC OSx [7] por lo mismo, es requerido recordar que las sintaxis de las consolas de comando de dichos sistemas operativos pueden variar y por consiguiente, la forma de introducir los argumentos puede cambiar. Típicamente, los sistemas operativos GNU/Linux tendrán problemas para pasar un argumento en forma de lista a Nmap, ya que utiliza caracteres reservados. Una forma de remediar esto, es el uso del argumento `--script-args-file <archivo>`. Tomemos la siguiente ejecución en un ambiente GNU/Linux como ejemplo:

```
nmap -6 -v -e eth0 -A -sC --script itsismx*
--script-args-file $HOME/Argumentos-NSE
```

El comando anterior toma todos los argumentos a partir del archivo `$HOME/Argumentos-NSE` y ejecuta todos los guiones desarrollados en esta investigación, de tal forma que la sintaxis interna del archivo, no se ve afectada por los caracteres especiales de la consola de comando.

Capítulo 4

Resultados de las técnicas propuestas

4.1. Ambiente de pruebas

Para la realización de esta investigación se utilizó equipo físico para generar la topología de red mostrada en la figura 4.1. Se utilizaron nodos que tienen implementando tanto IPv4 como IPv6. Las sub-redes A,B, C, ESXi y DHCP son redes Ethernet aunque las últimas dos solo poseen un nodo cada una. La sub-red B cuenta exclusivamente con un único nodo dedicado a realizar la ejecución de las herramientas desarrolladas en esta investigación. Las sub-Redes A y C cuentan con dos nodos cada una y la red ESXi cuenta con un servidor VMware ESXi que alberga 8 máquinas virtuales. Además contamos con un servidor DHCP para ambos protocolos en una sub-red independiente para reflejar la administración de la red. Y, finalmente están las sub-redes W1 y W2 de baja velocidad que representan enlaces a distancia de la compañía.

Los enrutadores son modelos Cisco 2811 corriendo con sistema operativo IOS 12.4(T) , cada uno cuenta con 512 MB de RAM; las configuraciones de cada enrutador están disponibles en el apéndice correspondiente pero poseen una configuración básica y el único protocolo especial que ejecutan es EIGRP, que se trata de un protocolo de enrutamiento. Como consecuencia de esta selección, la carga de trabajo de los enrutadores es mínima con objetivo de garantizar que los enrutadores no sufran saturación de memoria, ni de procesos durante la operación normal de la red.

Los nodos seleccionados son heterogéneos, tanto en hardware como en sistema operativo, esto es con el objetivo de tener diferentes muestras de asignación de direcciones de IPv6 (tales como basadas en SLAAC, de privacidad, etc.) y cada uno de ellos genera poco tráfico en la red, garantizando que la carga de trabajo de los enrutadores sean lo más baja posible. Los nodos poseen los sistemas operativos GNU/Linux (tales como Ubuntu, OpenSuse y Kubuntu) así como sistemas Operativos Windows XP, Windows Server 2012 R2 y Windows 7 SP1; la tabla 4.2 ofrece mayor información sobre cada nodo.

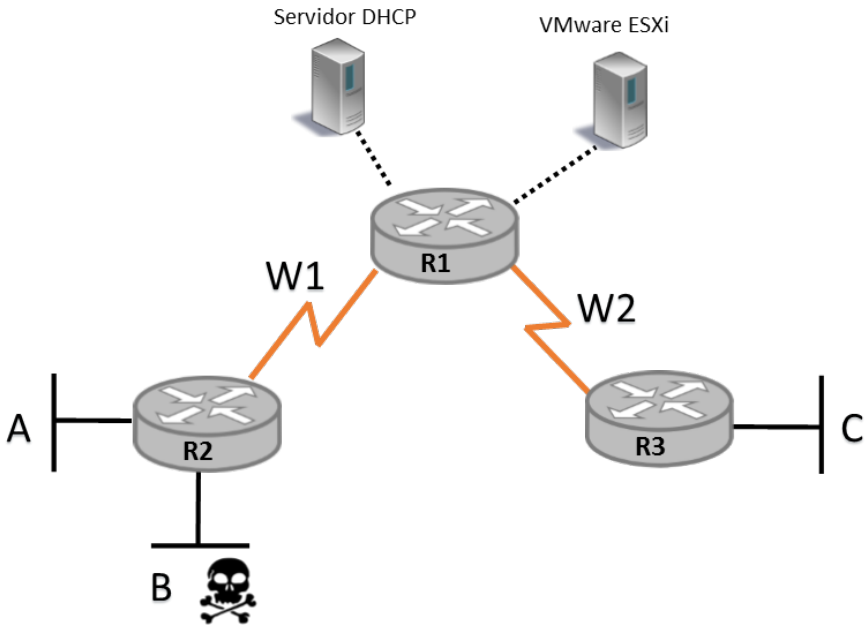

Figura 4.1: Topología de red empleada.

Respecto las direcciones de red, en IPv4 se utiliza un diseño simple y convencional dividiendo cada LAN física en sub-redes pertenecientes al bloque 192.168.0.0/16. No obstante, con IPv6 se opto por aprovechar las cualidades de múltiples direcciones de red, colocando en cada LAN física 2 direcciones de sub-redes lógicas diferentes. De esta forma podemos ampliar las tablas de ruteo y los enrutadores tratarán cada sub-red de forma independiente. La topología de red final se puede apreciar en la tabla 4.1.

Tal como se puede apreciar en las tablas 4.1 y 4.2 los enrutadores y el servidor DHCPv6 están configurados con las direcciones bajas de cada sub-red que les corresponda mientras que los *hosts* están configurados con *SLAAC* y algunos incluso con direcciones tipo *stateful* (DHCPv6). En el caso particular del *host* con Windows XP posee direcciones con el formato EUI-64 en vez de privacidad y esta diferencia se puede entender, ya que el *stack* original fue implementado en forma de prototipo a partir del Windows XP SP2, mientras que se implementó de forma definitiva en la familia NT 6.x (Vista, 7, 8,) [41].

4.1.1. Selección de Servidor DHCPv6

En el momento de desarrollo de la investigación existen dos servidores DHCPv6 populares: 1) *wide-dhcpv6* de código abierto y liberado para sistemas operativos GNU/Linux

Sub-red	IPv4	IPv6
R2 Sub-Red A	192.168.1.1/24	2001:DB8:C0CA:6001::1/64 2001:DB8:C0CA:FEA::1/64
R2 Sub-Red B	192.168.2.1/24	2001:DB8:C0CA:FEB::1/64
R2 W1	192.168.3.1/24	2001:DB8:C0CA:FE0::1/64
R1 Subred DHCP	192.168.6.1/24	2001:DB8:C0CA:AE00::1/64
R1 Subred ESXi	192.168.5.1/24	2001:DB8:C0CA:CED0::1/64
R1 Subred W1	192.168.3.2/24	2001:DB8:C0CA:FE0::2/64
R1 Subred W2	192.168.4.2/24	2001:DB8:C0CA:EE00::2/64
R3 Sub-Red C	192.168.7.1/24	2001:DB8:C0CA:6003::1/64 2001:DB8:C0CA:EE01::1/64
R3 W2	192.168.4.1/24	2001:DB8:C0CA:EE00::1/64

Cuadro 4.1: Topología de red empleada

Nodo	Operativos	Subred	IPv4	IPv6
1	Windows XP SP3	A	DHCPv4	Map4to6, EUII-64, Privacidad
2	Ubuntu 13.04	A	DHCPv4	Map4to6,EUI-64,Privacidad,Wordis
3	Ubuntu 13.04	B	DHCPv4	EUI-64,Privacidad
4	Xubutun 12.04	C	DHCPv4	Map4to6,EUI-64,Privacidad,Wordis
5	OpenSuse 12	C	DHCPv4	EUI-64,Privacidad
6	Windows Server 2012 R2	DHCP	Estática	Lowbyte, Privacidad
7	Windows 7 SP1	ESXi	DHCPv4	Map4to6, Privacidad
8	Windows 7 SP1	ESXi	DHCPv4	Map4to6, Privacidad
9	Windows 7 SP1	ESXi	DHCPv4	Map4to6, Privacidad
10	Windows 7 SP1	ESXi	DHCPv4	Map4to6, Privacidad
11	Windows 7 SP1	ESXi	DHCPv4	Map4to6, Privacidad
12	Ubuntu 12.04 LTS	ESXi	DHCPv4	EUI-64, Privacidad
13	Ubuntu 12.04 LTS	ESXi	DHCPv4	EUI-64, Privacidad
14	Ubuntu 12.04 LTS	ESXi	DHCPv4	EUI-64, Privacidad
15	Ubuntu 12.04 LTS	ESXi	DHCPv4	EUI-64, Privacidad
16	Ubuntu 12.04 LTS	ESXi	DHCPv4	EUI-64, Privacidad

Cuadro 4.2: Configuración de los nodos.

[52]. y 2) DHCPv6 de Microsoft [53]. Ambos son funcionales y operan adecuadamente con agentes de paso de otros productos, tales como los enrutadores Cisco, y con diferentes tipos de clientes. Por sencillez, se escogió el servidor de Microsoft, ya que sigue un lineamiento del RFC 3315 que omite el *wide-dhcpv6*: Revisar el *scope* del agente de paso origen para decidir que dirección asignar al nodo cliente [38]. Al contrario, *wide-dhcpv6* se basa mediante configuración en sub-interfaces, dando como resultado que se requieran pasos adicionales para que pueda operar con múltiples *scopes* de direcciones IPv6.

Se realizaron pruebas pre-eliminares con ambos protocolos con lo que se observó un comportamiento particular con *wide-dhcp*, ya que al estar configurado mediante sub-interfaces, ofrecerá una respuesta al recibir un mensaje desde cualquier agente de paso sin importar que exista incogruencias en la información de los agentes paso con la piscina que posea el servidor en dicha sub-interfaz, por lo tanto **es posible obtener falsos positivos cuando se envían mensajes fabricados a los servidores DHCPv6**. Este riesgo ha sido tomado en cuenta y el *script* de DHCPv6 se cerciora de que sea una respuesta válida.

Aunque el RFC 3315 habla acerca de dos tipos de solicitudes que se llaman direcciones IA-NA (*IA Nontemporary*) y IA-TA (*IA Temporary Address*) [38] el RFC deja ambiguo la forma de solicitar direcciones IA-NA y por el momento esta capacidad ha sido deshabilitada en el *script*. La figura 4.2 muestra un mensaje fabricado del tipo IA-NA que es idéntico a los generados por *wide-dhcpv6* y es aceptado por ambos tipos de servidores DHCPv6.

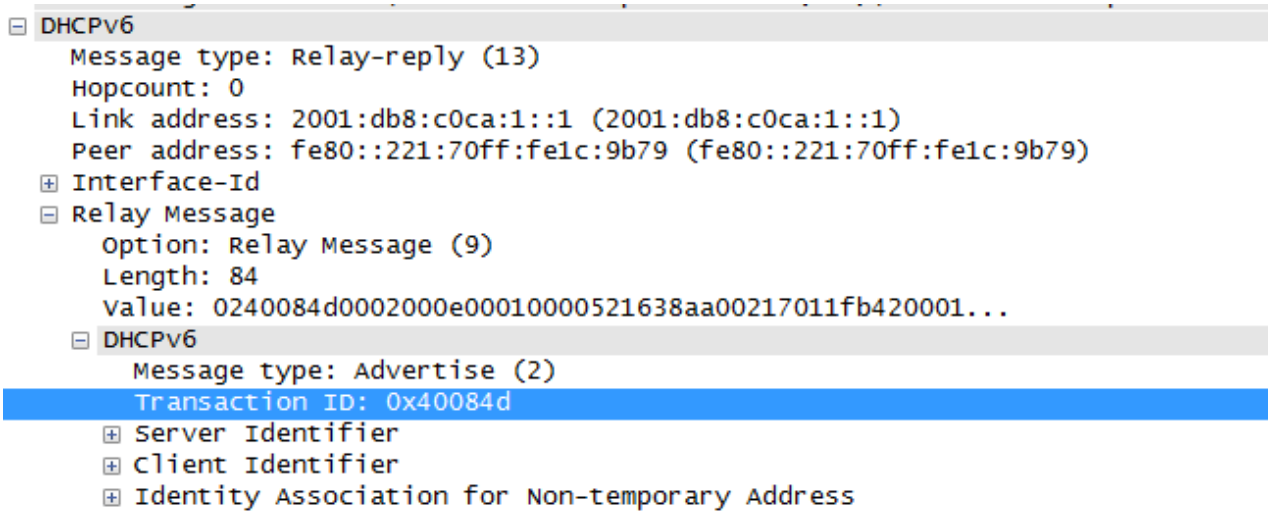

Figura 4.2: Mensaje REQUEST de DHCPv6 fabricado.

Originalmente, se tenía la idea de generar agentes de pasos falsos para intentar ocultar nuestras acciones, sin embargo, la naturaleza de IPv6 exige la necesidad de validar la existencia de un nodo antes de entregar un mensaje *unicast* al mismo, esto es importante debido a que las respuestas del servidor DHCPv6 al cliente son del tipo *unicast* y por lo mismo, el agente de paso primero realizará una solicitud de *ND Solicitacion* previo a pasar la respuesta. Si no existe un nodo que responda a la solicitud, el mensaje será descartado. Además, debido al riesgo de recibir un falso positivo no podemos asumir que un mensaje *ND Solicitacion* sea una respuesta segura para nosotros. El *script* requiere tomar la dirección de la interfaz de red ethernet de la maquina que lo esta ejecutando, aunque, tiene la opción de utilizar direcciones distintas proveidas por el usuario, pero queda a disposición de él que se pueda responder las solicitudes.

La figura 4.3 muestra un caso de éxito, donde se fabricaron mensajes de solicitud y el servidor responde con un mensaje de *Adverstiement*, como nuestro código solo le interesa el segundo mensaje, el nodo en cuestión nunca separó su dirección puerto y por lo mismo responde con un mensaje *ICMPv6 Destination Unreachable*.

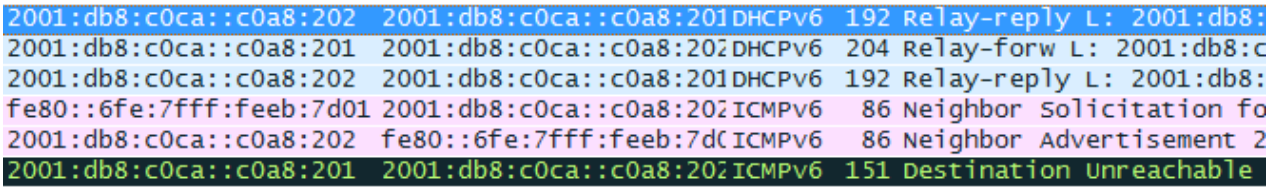

Figura 4.3: Respuesta de 4 pasos interrumpida por NDP.

4.2. Ejecución de pruebas

A excepción del *script* `itsismx-report`, los demás *scripts* desarrollados para Nmap pueden ejecutarse por separado o en cualquier conjunto que se desee. Aunque, el *script* `itsismx-dhcpv6` al ser ejecutado exclusivamente en la fase de *pre-scanning* debe garantizarse que se ejecute previo a los demás *scripts* o bien de forma independiente para pasar sus resultados como argumentos a los demás *scripts*. Mientras que lo segundo puede ser útil cuando se quiere hacer solamente exploraciones rápidas de posibles sub-redes lo primero basta con indicar el nombre del *script* para que se ejecute antes que los demás.

La sintaxis más sencilla para ejecutar los scripts sería la siguiente:

```
nmap -6 -e <Interfaz>--script itsismx* --script-args <Argumentos>
```

El primer argumento `-6` será siempre obligatorio pues se debe indicar a Nmap que correrá - exclusivamente - en IPv6 mientras que `-e <Interfaz>` es requerido para la ejecución de `itsismx-dhcpv6` y de cualquier otro *script* que requiera fabricar mensajes a insertarse en interfaces ethernet. Los argumentos de los *scripts* son varios y con propósitos especifico, y se puede revisar el apéndice de los códigos fuentes para ver una lista más detallada de ellos. De cualquier manera este mecanismo puede resultar difícil de manipular. Por lo mismo se sugiere el manejo de un *script* de sistema, ya sea un *script* de GNU/Linux o de Microsoft, como el siguiente:

```
#! /bin/bash
Scripts=‘‘itsismx*’’
Script-Args-File=‘‘UBICIACION-DE-ARCHIVO-CON-ARGUMENTOS’’
Args=‘‘-6 -v3 -sn -e eth0’’
nmap $Args --script $Scripts --script-args-file $Script-Args-File
```

Este *script* de sistema permitirá preparar la ejecución de Nmap con sencillez, particularmente si no queremos ejecutar todo los scripts NSE disponibles. La tercera línea hace referencia a un archivo donde se hallarán los argumentos a utilizar para cada uno de los *scripts* NSE. Un ejemplo de su posible contenido se muestra a continuación:

```
itsismx-dhcpv6.subnets=2001:db8:c0ca:6001::/64,2001:db8:c0ca:6002::/64,
2001:db8:c0ca:6003::/64,2001:db8:c0ca:6006::/64
itsismx-subnet=2001:db8:c0ca:fea::/64,2001:db8:c0ca:ced0::/64,
2001:db8:c0ca:ee01::/64
itsismx-Map4t6.IPv4Hosts192.168.1.0/8,192.168.5.0/8
itsismx-slaac.vendors=IBM,DELL
itsismx-LowByt.nbits=8
itsismx-slaac.nbits=8
itsismx-slaac.vms=Wd
newtargets
```

Con estos argumentos estamos dando suficientes datos para ejecutar todo los *scripts* para la topología de red que se ha implementado. Una vez ejecutado Nmap iniciará su trabajo de exploración e inmediatamente iniciará el escaneo. Por supuesto, es posible realizar pruebas con argumentos adicionales de Nmap, el más socorrido por estos *scripts* NSE es el nivel de *verbosity*, a nivel 5 despliega cada una de las direcciones

añadidas al escaneo - una forma peligrosa de engordar el archivo resultante - mientras que con *verbosity* de nivel 3 se obtiene los grandes bloques de avance. En lo general, utilizar *verbosity* de nivel 2 o 1 es más que suficiente para obtener información adecuada de cada *script* NSE.

4.2.1. Impacto en el *host*

Es importante recordar, que si queremos explorar 2^24 direcciones IPv6, estamos hablando de almacenar 16 bytes 2^24 veces, lo cual se refleja en demasiada memoria a almacenar. Mientras que una ejecución típica de Nmap maneja pequeños bloques y memoria paralela, al estar indexando las direcciones mediante guiones NSE, estas se deben de ir guardando en memoria. Como resultado, **es posible desbordar la memoria RAM de la máquina anfitrión cuando se realizan exploraciones bastantes grandes**. Durante los experimentos, el desborde ocurrió aún cuando los nodos contaban con 8 GB de memoria RAM.

Para compensar esta limitante a los *scripts* se les añadió una capacidad más "ahorrativa" de memoria la cual se activa con el argumento a *scripts* NSE `itsismx-SaveMemory`. Al estar activado, añade directamente las direcciones a un registro de Nmap para la fase de *hosts*. Esto tiene el costo de disminuir la certeza en el reporte final que se genera con el *script* `itsismx-report`, pero es posible obtener toda la información leyendo los resultados arrojados previamente.

El mecanismo de ahorro en memoria tiene un límite, si se hacen bastantes exploraciones masivas en las sub-redes, por ejemplo buscando direcciones basadas en *SLAAC*, es posible agotar la memoria, y en estos casos será necesario hacer granular la búsqueda dividiéndola en diferentes ejecuciones de Nmap.

4.2.2. Dimensión de las exploraciones

En general, las técnicas ofrecidas en esta investigación son eficientes, pero no por ello rápidas. La técnica ofrecida para calcular direcciones basadas en *SLAAC* que utilizan el formato EUI-64 es un excelente ejemplo: **Permite el descubrimiento de todos los nodos que estén usando cierto tiraje de una compañía en particular**. Pero esto involucra exploraciones de 24 *bits*, estas dimensiones son el peor escenario cuando uno quiere explorar TODA UNA RED IPv4 de una entidad en particular. Y en estos casos *Nmap* requiere una cantidad de tiempo significativa, en experimentos realizados sobre la topología se excedieron de 24 horas.

Asumiendo que la exploración en IPv6 fuese tan eficiente como IPv4, podríamos obtener por cada día de trabajo un *set* completo de posibles nodos. Sin embargo, solamente DELL posee 49 OUI. **Si nosotros quisiéramos explorar una red interna que tiene como proveedor a DELL, ¡estriamos ejecutando 49 exploraciones de 24 *bits* por cada sub-red que poseyera la empresa!** Como se puede apreciar, es necesario que previo al escaneo podamos realizar una investigación de fondo para poder descubrir la serie de OUI que la empresa esté utilizando para poder acotar la búsqueda y particularmente estar seguros de conocer todas las sub-redes. En general, **Las exploraciones en IPv6 requieren investigación de fondo para poder optimizarlas lo más posible y evitar realizar exploraciones a ciegas**.

4.2.3. Riesgo de DoS en los equipos intermedios y su mitigación

Tal como se mencionaba en capítulos anteriores, puede existir un riesgo de negación de servicio durante escaneos masivos. Estos eventos, no ocurren como consecuencia de la carga en escaneos masivos en el protocolo de IPv6. De hecho, es una carga muy similar a lo que ocurre en el protocolo IPv4, solo que con encabezados de IP más grandes. Como consecuencia, si un enrutador en IPv4 tiene suficiente potencia para atender el tráfico nativo de la red y a la vez el de las herramientas de análisis, entonces tiene igual probabilidad de soportar la carga durante un análisis en IPv6.

Sin embargo, estos eventos de negación de servicio, ocurrieron constantemente durante las primeras pruebas de esta investigación. Específicamente, con las características de los enrutadores Cisco 2811 mencionados en el capítulo anterior, la negación se reflejo como la perdida de comunicación de EIGRP, provocando errores en las tablas de ruteo. Este problema se puede apreciar mejor observando los datos desplegados de uno de los enrutadores:

```
R2# show processes memory | include IPv6
131 0 0 0 7204 0 0 IPv6 Echo event
189 0 0 0 7204 0 0 IPv6 Inspect Tim
295 0 15488 1172 11476 0 0 IPv6 RIB Event H
296 0 55580 1144 30820 0 0 CEF: IPv6 proces
298 0 63644 67124 12600 0 0 IPv6 IDB
299 0 23048 0 13440 0 0 IPv6 Input
300 0 182728 9905216 23632 0 0 IPv6 ND
301 0 0 0 7272 0 0 IPv6 Address
```

```
304 0 4299048 63602488 30860 0 0 IPv6-EIGRP
307 0 394361232 335344664 7204 0 0 IPv6-EIGRP Hello
```

El comando despliega la memoria RAM destinada a diferentes procesos de IPv6. El enrutador posee un total de 512 MB, de ellos, en el instante en que se ejecutó el comando , 182 MB estaban ocupados exclusivamente por el caché de los nodos vecinos, mismo que se ve considerablemente engordado por la exploración que se esta realizando en ese instante. Dicho caché se debe de manejar de esa forma, ya que es un requisito del protocolo *Neighbor Discovery*. Además, esta asignación de memoria es dinámica, conforme el número de nodos a explorar se incremente, el enrutador ira asignando más memoria al proceso de IPv6 ND provocando que otros procesos pierdan memoria. Por lo mismo, la negación de servicio se ve reflejado en mi escenario de prueba como la pérdida de las tablas de ruteo, ya que eventualmente EIGRP pierde tanta memoria, que se ve forzado a eliminar sus tablas internas y reiniciar el proceso de aprendizaje.

Es importante aclarar que estos efectos negativos durante un escaneo **dependerán en gran medida del modelo, cantidad de memoria y/o servicios de los dispositivos intermedios que operen con IPv6, y por lo mismo los efectos pueden ser distintos**.

La mitigación de estos riesgos son variados, algunos los pueden tomar la misma compañía que administra la red y muchos otros el usuario que realiza la exploración. **Estas mitigaciones pueden ser: reducir la cantidad de nodos a buscar por exploración, incrementar el tiempo entre cada escaneo y/o configurar los enrutadores para que guarden por muy poco tiempo los registros de los nodos**. De hecho, en esto ultimo es como se diferencian principalmente los sistemas GNU/Linux con *USAGI* de los sistemas de Microsoft, los primeros solo guardan las entradas de los nodos por unos pocos milisegundos.

Durante el instante que los enrutadores perdían las tablas de ruteo, se generaban varios falsos negativos ya que los paquetes eran inmediatamente rechazados por el Gateway, por lo mismo fue necesario reducir el tiempo entre los envíos de cada *probe* modificando los controles de tiempo y rendimiento que están indicados en el apéndice de Nmap. Para estos escenarios, basto con indicar que se hiciera un escaneo "político" (`-T2`), aunque en los experimentos más grandes sí fueron capaces de provocar el evento de negación de servicio. Por lo mismo es posible que convenga utilizar escaneo "fortuito" o "paranóico" (`-T1` ó `-T0`) o bien modificar directamente el tiempo de espera con los argumentos `--scan-delay/--max-scan-delay` (Nmap sugiere modificar estos parámetros antes de usar los últimos modos [7]).

El hecho de tener que hacer más lentos las exploraciones provoca que las mismas tomen mucho más tiempo. Las figuras 4.4 y 4.5 son excelentes ejemplos. La primera muestra una exploración "agresiva" (`-T5`) mientras que la segunda una "política". Se pueden apreciar principalmente la diferencia del tamaño de mensajes a transmitir. Sin embargo la exploración "agresiva" dio los peores resultados, pues desde los primeros instantes provoco la negación del servicio. Los resultados con las exploraciones "política" dieron un mejor resultado por mayor tiempo pero eventualmente provocaron una negación de servicio.

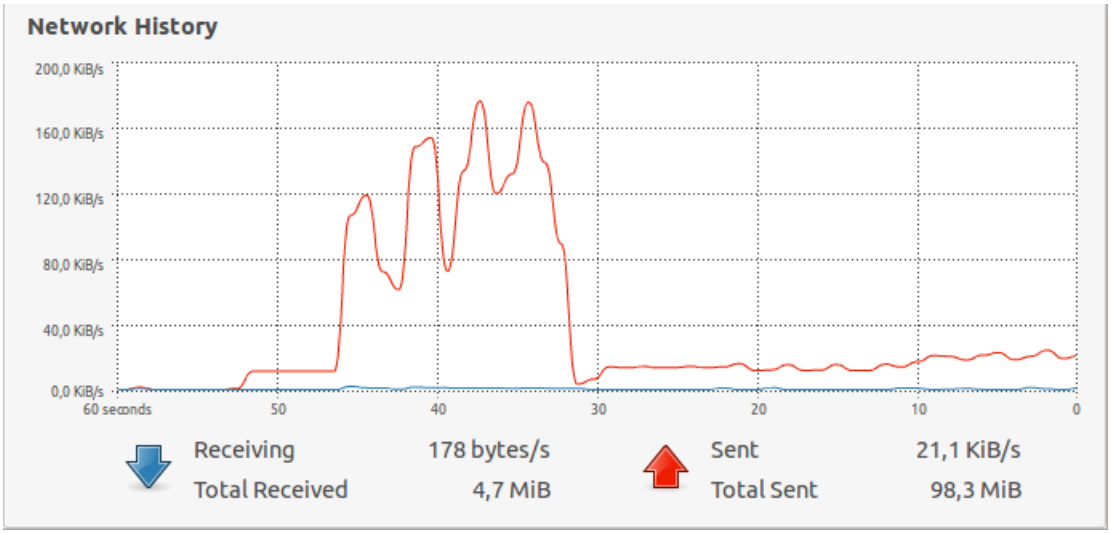

Figura 4.4: Efectos de una exploración agresiva (`-T5`).

En otras pruebas se utilizó el argumento `--scan-delay 2` para provocar una espera de 2 segundos entre cada mensaje, **al parecer este mecanismo tuvo un mejor efecto, sin embargo tiene el costo de incrementar el tiempo de la exploración de forma significativa, y en exploraciones grandes se vuelve impráctico** . Esto se agrava si la exploración es bastante amplia. Por supuesto, estos valores variarán de los lugares donde se realicen las pruebas pues mayor capacidad de memoria RAM en los enrutadores posiblemente permita una búsqueda más rápida.

El RFC 6583 y en menor medida el RFC 5157, tratan este mismo tema e indican que una pobre implementación una pobre implementación del protocolo ND puede provocar la aparición de los DoS [54, 17]. Sin embargo, al tratarse de documentos liberados apenas en el 2008 y 2012, pocos o ningún sistema operativo puede que ya halla implementado las recomendaciones. Tal es el caso de todos los sistemas operativos utilizados en esta investigación. El RFC 6583 también indica que un escaneo malicioso, tanto de forma remota como local, podría provocar la negación del servicio del protocolo

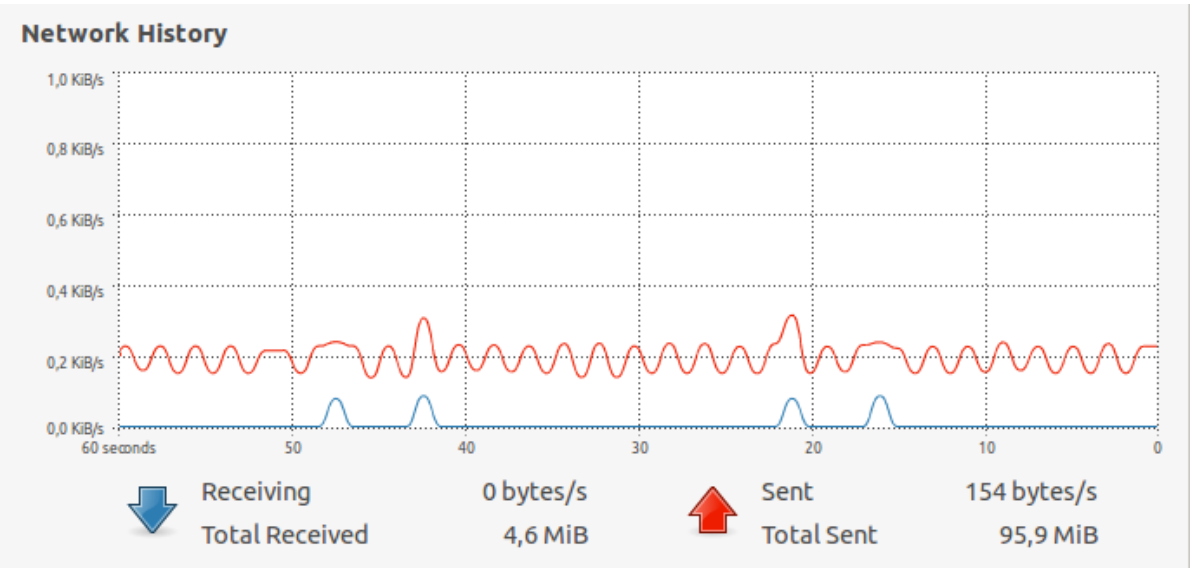

Figura 4.5: Efectos de una exploración política (`-T2`).

ND, principalmente la solicitud de información de la red [54]. El mismo RFC propone los siguientes modelos para reducir el riesgo de DoS:

1. Filtrado del espació de direcciones sin usar.
2. Tamaño minino de sub-red (Que el prefijo refleje el total de nodos existentes).
3. Mitigaciones de enrutamiento (utilizando interfaces nulas).
4. Controlar los tiempos de NDP.

Con las primeras tres opciones, **es posible mitigar el riesgo mediante listas de control de acceso o por rutas nulas de los espacios de direcciones que no se están utilizando**. Sin embargo, esto puede resultar tedioso o difícil de implementar y obliga a utilizar un esquema de direccionamiento *stateful* [54]. La segunda opción también requiere el mismo espacio de direccionamiento, además de que facilita la exploración de la red al dejar claro de antemano la máxima cantidad posible de dicha sub-red.

La cuarta opción involucra tener controles de afinación de los procesos de NDP, sin embargo esto requiere que el sistema operativo lo soporte, lo cual no es el caso en esta investigación, además, valores incorrectos pueden resultar en funcionamiento adverso en la red de la entidad.

4.2.4. Caso de VMware ESXi

Posiblemente, el caso de descubrir máquinas virtuales manejadas por servidores VMware erán los más interesantes, pues ofrecían la posibilidad de realizar escaneos en base a 8 *bits*. De acuerdo a la documentación ofrecida por la compañía VMWare [51, 5] la forma de asignar direcciones MAC están basado en dos formatos: 00:50:56:XX:YY:ZZ y 00:0C:29:WW:TT:UU siendo el primero utilizado para asignaciones manuales y el segundo para asignaciones dinámicas.

Es justamente el segundo tipo de dirección el que generaba bastante interés pues la parte WW:TT sería tomada a partir de los 16 *bits* más bajos de la dirección IPv4 que poseyera el servidor VMware. Por ejemplo, si el servidor tuviese como dirección 192.168.10.12 la dirección MAC tendría el formato 00:0C:29:0A:0C:UU siendo los últimos 8 *bits* asignados de forma aleatoria. Sin embargo, este no es el caso de las máquinas virtuales que se crearon en el servidor VMWare ESXi. A continuación se despliegan las direcciones que dichas máquinas poseen:

Ubuntu 00:0C:29:6A:02:0D y 00:0C:29:6A:02:17 (. . . 106.2)

Ubuntu 00:0C:29:20:8E:3D y 00:0C:29:20:8E:47 (. . . 32.142)

Ubuntu 00:0C:29:74:C4:0E (. . . 116.196)

Ubuntu 00:0C:29:75:ED:93 (. . . 117.237)

Win7 00:50:56:00:DA:DA (Configuración manual)

Win7 00:0C:29:DB:6D:A9 (. . . 219.109)

Win7 00:0C:29:OC:02:8A y 00:0C:29:OC:02:94 (. . . 12.2)

Win7 00:0C:29:6A:66:3E (. . . 219.109)

La asignación de estas direcciones MAC parecen ser arbitrarias, aunque los últimos 8 *bits* si son generados aleatoriamente, y se puede comprobar en aquellas máquinas que se les añadió interfaces adicionales. El porque de este comportamiento de VMware no fue posible ubicarlo, sin embargo pudo ser debido a que al momento de la asignación de las direcciones MAC, el servidor estuvo temporalmente sin dirección de IPv4 y una vez creadas las máquinas, estas simplemente no generaron nuevas direcciones de MAC. De cualquier manera, la generación de nuevas máquinas virtuales continuo generando direcciones arbitrarias.

Como resultado de esta situación, el localizar máquinas con direcciones dinámicas de VMware es tan complejo como cualquier otra máquina física o de otros tipos de máquinas virtuales.

4.2.5. Configuración para evitar DoS, a través de una búsqueda optimizada.

Como síntesis de los resultados anteriores, para poder desarrollar pruebas de escaneo en la topología de red, **fue necesario afinar todos los componentes para obtener respuesta en tiempos aceptables**, medido en horas, y para ello **se utilizaron los siguientes argumentos para los *scripts***:

```
itsismx-dhcpv6.subnets={2001:db8:c0ca:6001::/64,2001:db8:c0ca:6003::/64,
2001:db8:c0ca:6006::/64}
itsismx-subnet={2001:db8:c0ca:fea::/64,2001:db8:c0ca:ced0::/64,
2001:db8:c0ca:ee01::/64,2001:db8:c0ca:ced0::/64,2001:db8:c0ca:ae00::/64}

itsismx-LowByt.nbits=10
tsismx-slaac.nbits=10
tsismx-slaac.vms-nbits=10
itsismx-slaac.compute=random
itsismx-slaac.vendors={00096b,002170}
itsismx-Map4t6.IPv4Hosts={192.168.1.0/24,192.168.5.0/24}
newtargets
```

Originalmente se había incluido la capacidad de buscar en el dominio de las máquinas virtuales de VMware con el argumento `itsismx-slaac.vms=wD` sin embargo, al utilizar una exploración de 10 *bits* para SLAAC significaba que además de las 1024 direcciones para cada OUI, estarían los bits introducidos mediante `itsismx-slaac.vms-nbits` a los cuales serían añadidos 8 *bits* adicionales de barrido. Por ejemplo, si se introdujeran 10 bits, se estarían hablando de un total de 1024x255 direcciones. Mismas que engrosarían el total de direcciones a explorar en un 60 %. Por consiguiente, se opto por realizar las búsquedas de estas en ejecuciones separadas. Los OUI que aparecen, corresponden a los dos que toda las máquinas físicas tienen, el primero es de IBM y el segunda el de DELL.

Además de los argumentos para NSE se utilizaron los siguientes argumentos para Nmap: `-6 -v3 -e eth0 -T2 -sn` donde los primeros 3 no tienen impacto alguno en

los tiempos, en cambio el argumento `-sn` reduce la búsqueda de cada nodo al eliminar un escaneo completo de los puertos. Y el argumento `-T2` reduce el tiempo y tamaño de cada *probe* para intentar evitar un escenario de negación de servicio en los enrutadores, aunque no se tuvo éxito, ya que hubo un instante donde ocurrió el evento. A partir de ahí las siguientes búsquedas refinadas manejan el argumento `--scan-delay 2` para reducir los riesgos de otros eventos.

El pronóstico de exploración con estos argumentos es explorar 8 sub-redes, en cada una de ellas buscar 2 x 1024 nodos (SLAAC) + 1024 (LowBytes) + 512 (Mapeo 4 a 6) además de otros 7 (Basados en palabras), dándonos un total de exploración de 28,728 direcciones. El número total de direcciones que se anexaron a memoria fue casi de 2$\hat{1}$5. Es decir, **añadido al tiempo que le toma a NSE pasar cada una de las direcciones al listado de Nmap, este íltimo debió hacer una exploración similar a una de fuerza bruta de 15 *bits***. El tiempo aproximado para ejecutar este experimento fue de 19 horas.

Lo siguiente es una captura de pantalla del reporte final de esta prueba:

```
Post-scan script results:
| itsismx-report:
| Subnets:
| 4 Were confirmed to exits using the spoofing technique with DHCPv6
| Hosts:
|
| List:
| 2001:db8:c0ca:6001::c0a8:10d
| 2001:db8:c0ca:6003::c0a8:50d
| 2001:db8:c0ca:6001::c0a8:10c
| Technique: Map6to4 - Discovered 3 nodes online which is 25 Of total nodes
discovered.
|
| List:
| 2001:db8:c0ca:fea::1
| 2001:db8:c0ca:ae00::1
| 2001:db8:c0ca:ced0::1
| 2001:db8:c0ca:ae00::2
| 2001:db8:c0ca:ee01::1
| 2001:db8:c0ca:6001::1
| 2001:db8:c0ca:6003::1
```

```
| Technique: Low Bytes - Discovered 7 nodes online which is 58.333333333333
Of total nodes discovered.
|
| List:
| 2001:db8:c0ca:ee01::10ca
| 2001:db8:c0ca:fea::dead:beef
| Technique: Words - Discovered 2 nodes online which is 16.666666666667 Of
total nodes discovered.
|_ Repeats:
```

4.2.5.1. Análisis para la optimización de tiempos de ejecución

La técnica de "Mapeo" 4 a 6 puede ser manejada en el formato original, que era ::a.b.c.d, sin embargo esta depreciado y no se recomienda [19]; por lo mismo se omito de las pruebas. Sin embargo, es posible ejecutar pruebas con esa sub-red de forma independiente a las demás. Este formato también puede servir para buscar aquellas interfaces relacionadas a los túneles para conectar islas IPv6.

Cada ejecución de está prueba podría arrojar resultados distintos para la técnica de SLAAC, ya que por defecto utiliza un mecanismo de búsqueda de direcciones aleatorias, en estos casos simplemente no tuvimos suerte en acertar en alguna de 10 direcciones que estaban en uso. Para obtener todo el resultado deberíamos utilizar los 24 *bits*, pero en solo un experimento serían 268,435,456 direcciones a buscar **y debido a los argumentos dados a Nmap, sería imposible realizar el barrido en tiempos adecuados a causa de la espera entre cada nodo**.

Como se indicaba previamente, los guiones son modulares y estos pueden ser ejecutados en cualquier orden. Además, los argumentos utilizados en estas pruebas, mencionados más arriba, fueron pasados a los *scripts* mediante el argumento `--script-args-file`, y la ejecución fue granular. En una primera instancia, cuando se realizaban exploraciones de pocos bits en cada una de las técnicas todas fueron realizadas en una misma ejecución de Nmap. Sin embargo, al incrementar el número de bits a buscar, principalmente en los guiones de las técnicas de SLAAC y "mapeo 4 a 6", estos últimos se ejecutaban de forma separada. Primero para obtener un reporte rápido de las técnicas que ocupaban una cantidad pequeña de nodos a explorar, y segundo para reducir aún más el riesgo de quedarse sin memoria.

Por el diseño de Nmap, se genera una estructura de datos que posee además de la dirección, otros elementos a partir de los argumentos dados. Es debido a ello que

se requiere memoria para los 2^24*16 *Bytes* de direcciones como para la información arrojada por Nmap para cada una de dichas direcciones. Como resultado si ejecutamos exploraciones completas de 3 OUI, sin importar si son máquinas virtuales o no, para 3 diferentes sub-redes se puede exceder por mucho la cantidad de RAM del *host* que esta ejecutando la prueba, tal como ocurrió con esta investigación. Como consecuencia, sería necesario ejecutar cada una de las exploraciones por separado. Además, se puede hacer uso del argumento `itsismx-SaveMemory` para reducir la memoria a consumir durante las exploraciones.

Las particiones dependerán de los elementos a explorar así como de los argumentos dados a Nmap. Sin embargo, en esta investigación resulto evidente que explorar dos espacios de OUI pueden a llegar a ocupar hasta 6 GB de la memoria RAM del *host*.

Entre más grande sea la exploración, será mayor el consumo de memoria y de tiempo. cada *script* tiene un argumento propio que controla las dimensiones de exploración para cada sub-red proveida. De forma breve, estos elementos son:

SLAAC es el caso más emblemático, pues maneja dos, uno para las exploraciones de OUI y la mayoría de las máquinas virtuales, y uno secundario, no utilizado por defecto y exclusivo para explorar espacios de VMWare dinámicos. Por defecto son 11 bits (2,048 direcciones) y 2 bits (4 barridos de 256 nodos cada uno para dar un total de 1,024).

Low-Bytes posee un barrido por defecto de 8 bits (256 direcciones), puede ser cambiado a un rango de 1 - 24 bits.

Palabras es un barrido por cada palabra en el diccionario (7 entradas en esta investigación).

Mapeo 4 a 6 definido por cada dirección proporcionada. El peor escenario serian grupos de 24 bits y en el más óptimo unas cuantas direcciones.

Utilizando como argumentos el siguiente listado:

1. `itsismx-dhcpv6.subnets={2001:db8:c0ca:6001::/64,2001:db8:c0ca:6002::/64, 2001:db8:c0ca:6003::/64,2001:db8:c0ca:6006::/64`
2. `itsismx-subnet={2001:db8:c0ca:fea::/64,2001:db8:c0ca:ced0::/64, 2001:db8:c0ca:ee01::/64,2001:db8:c0ca:ced0::/64,2001:db8:c0ca:ae00::/64`
3. `itsismx-Map4t6.IPv4Hosts={192.168.1.0/24,192.168.5.0/24}`

4. `itsismx-slaac.vendors={00096b,002170}`

5. `newtargets`

6. Para nmap: `-6 -sn -n -e eth0`

Dejando los argumentos relacionados a las dimensiones de las exploraciones en sus valores por defecto (11 bits SLAAC, 8 Lowbytes sin incluir máquinas virtuales), se tiene un rango de exploración de 86,868 (56 de palabras, 2048 de Low-bytes, 18,288 de "mapeo" y 63,476 de SLAAC) nodos con un tiempo estimado de 21 horas. Si a esta misma ejecución se le omite el tercer argumento, la cantidad de tiempo estimado sería de 5 horas.

Si cambiamos el espacio a recorrer a solamente 8 bits, sin exploración de "mapeo" 4 a 6, la cantidad de nodos a explorar es de 10,040, con un tiempo estimado de 3-7 minutos. Si al mismo experimento le regresamos el argumento de "mapeo" 4 a 6, se le agregan 18,288 direcciones, y tomará un tiempo estimado de 5-7 minutos, aunque tendrá un riesgo de que ocurra la negación de servicio, si no se toman medidas preventivas (con el consecuente incremento en el tiempo de exploración). Si la cantidad de nodos se incrementa, por ejemplo añadiendo la capacidad de explorar máquinas virtuales, el DoS ocurrirá forzándonos a tomar medidas preventivas.

De esta series de experimentos, las técnicas más eficaces fueron las de palabra y las de direcciones bajas, esto debido a que no habían más de 16 máquinas configuradas con SLAAC y la probabilidad de haber acertado a dichas direcciones es muy baja. Las ejecuciones de estás mismas técnicas son las más rápidas. Por ultimo, **si se realizan barridos menores a los 20 mil direcciones se tiene un bajo riesgo de provocar una negación de servicio, que puede ser controlado sin necesidad de alterar los tiempos de exploración de Nmap**.

En un escenario similar a los anteriores, pero con un espacio de 10 bits, 2 OUI y 508 de IPv4 (descontando primera y ultima dirección de cada sub-red), tenemos un total de 294,000 direcciones a explorar y se estarían manejando tiempos de alrededor de 18 horas, dependiendo de los argumentos dados para reducir el riesgo de un DoS.

Para exploraciones nativas de Nmap en IPv4 con barridos de 24 bits, toma más de 24 horas. Para IPv6, estos tiempos son mucho peores, pues es requerido controlar la cantidad de *hosts* a explorar simultáneamente para evitar un DoS. Esto trae como consecuencia un incremento significativo en el tiempo. Algunas pruebas que se realizaron, llegaron a requerir más de 4 días y aun así sufrieron de negación de servicio.

4.3. Comparativas con 6Tools

Como se había mencionado en el capítulo 2, dentro del as herramientas previamente conocidas, existe una denominada 6Tools [5], desarrollada por Gont. Estas herramientas están escritas en lenguaje C y se especializan más en ataques para auditorías de seguridad en nodos IPv6. Para facilitar las comparaciones **denominaremos a las técnicas desarrolladas como itsismx**.

En el momento de realizar esta comparación se halla disponible la versión 1.4.1 de 6Tools la cual esta integrada por varias herramientas, la mayoría de ellas enfocadas para realizar ataques específicos [5] que van desde fabricación de los diferentes tipos de mensajes de *Neighbor Discovery*, como mensajes irregulares de IPv6. Pero para esta investigación es la herramienta denominada **scan6 que tiene capacidades similares a lo aquí desarrollado**. Las características de scan6 son:

1. Capacidad de cambiar las direcciones de enlace local o global de la interfaz que origina los mensajes (más no ofrece garantía de respuestas correctas si se hace).
2. Capacidad de analizar direcciones de enlace local o global. En el primero, puede estar en modo pasivo (registrando quien aparece) o en modo activo.
3. Puede fabricar segmentos TCP y UDP con puertos aleatorios y otros atributos de dichos encabezados.
4. Capacidad de clasificar los diferentes tipos de direcciones que obtuvo, y desplegar solo una por cada nodo.
5. Control de cantidad, tamaño y velocidad de transmisión (medido en bits por segundos) de cada *probe*.
6. Capacidad de indicar un máximo número de direcciones.
7. Capacidad de revisar constantemente dichas direcciones (y el tiempo a pasar entre cada revisión).
8. El mecanismo de *Low-Bytes* maneja rangos de X:X:X:X::0:0 a X:X:X:X::100:1500 para ser un máximo de 5,632 direcciones por cada prefijo otorgado.
9. Separa los mecanismos de OUI de forma similar a como nosotros lo hacemos.
10. Tiene capacidad similar a la técnicas de palabras al permitir recibir los últimos 64 bits (aunque no se refiere como tal a dicha técnica y no ofrece un diccionario base).

11. Capacidad de recibir prefijos IPv4.

12. Capacidad de operar con máquinas virtuales.

Las características 2, 4 y 7 son únicas de scan6, mientras que las demás son realizadas ya sea directamente por itsismx o bien ya estaban integradas en el mismo Nmap. El siguiente listado marca las principales diferencias de la implementación de Itsismx respecto a scan6. siendo Nmap el factor clave:

1. Nmap permite decidir el tipo de mensajes a utilizar para obtener respuesta del nodo. Puede ser simple ICMPv6 o basados en TCP o UDP.

2. Nmap permite controlar los tiempos entre los *probes* así como sus dimensiones y cantidad a diferentes dispositivos.

3. Nmap permite fabricar direcciones de origen arbitrarías tanto de IP como de la capa transporte.

4. Nmap permite controlar la transmisión de los mensajes con métricas más variables, tales como tamaño de grupos de nodos, tamaño del *probe*, tiempo de espera entra cada *probe*, etc.

5. El nivel de *verbosity* de Nmap (y el configurado en los guiones de Itsismx) permite realizar el pronostico y estimaciones cuando la exploración se lleva a cabo. Scan6 tiene una *verbosity* que puede llegar a indicar prefijos y rangos junto a nodos descubiertos, pero no es posible determinar en que porción de la exploración se encuentra.

6. Las técnicas de Itsismx permiten trabajar a partir de los prefijos dados, mientras que scan6 puede llegar a ignorarlos.

7. Itsismx permite controlar la cantidad de nodos a explorar por cada técnica disponible, esto mediante argumentos para cada uno. Scan6 en cambio toma libertad de acuerdo al prefijo original.

8. Scan6 permite buscar máquinas con la técnica de "mapeo" 4 a 6 en los formatos: X:X:X:a:b:c:d y X:X:X:X::a.b.c.d mientras que Itsismx solo realiza el segundo formato.

9. Scan6 maneja máquinas virtuales solo para VirtualBox y VMware (rango dinámico) y en ambos casos intenta descubrir todo el rango.

10. Nmap permite, mediante la salida de error indicar cuando recibe mensajes de rutas inalcanzables. Es importante para detectar cuando se ha provocado un DoS en los enrutadores. Scan6 carece de este indicador.

11. Itsismx provee toda su funcionalidad mediante 5 guiones NSE. Scan6 ejecuta todo mediante argumentos al programa.

La importancia de la generación de mensajes con TCP o UDP es para poder eliminar falsos negativos al poder evitar hacer uso de ICMPv6, tal como se mencionaba en el capítulo 2. **Nmap, al poseer por defecto estas cualidades, permitió enfocarnos exclusivamente en las técnicas a desarrollar**. **Así mismo permite realizar la exploración con mayor control contra escenarios de negación**.

La distinción número 7 es importante, ya que si se diese la dirección 2001:db8:c0ca:fea::/64 para buscar las direcciones bajas, Itsismx liberaría una exploración basada en los primeros 11 *bits*, mientras que scan6 serían 5,632 direcciones repartidas en los dos primeros segmentos. Con Itsismx bastaría mediante el argumento `itsismx-LowByt.nbits` para reducir o ampliar su rango, pero con Scan6 sería necesario modificar el prefijo. Por ejemplo, para buscar solo los primeros 8 bits el usuario debería de introducir 2001:db8:c0ca:fea::/120. Esto conlleva al siguiente punto débil, **en scan6 el prefijo no siempre es respetado, mientras que Itsismx valida que la técnica pueda ser empleada con el prefijo entregado**.

La sub-red 2001:db8:c0ca:fea::/64 será valida para realizar todas las técnicas de Itsismx y de Scan6, pero si quisiéramos usar la sub-red 2001:db8:c0ca:fea::/96 (que podría corresponder a un "mapeo" directo de IPv4 a IPv6) la técnica de *low-bytes* validaría con su rango por defecto. La técnica basada en SLAAC por el contrario, no se ejecutaría, pues el prefijo deja menos de 64 bits disponibles. la técnica basada en palabras, tomaría solo aquellas que ocuparan hasta 2 segmentos (c0ca, beef:loca, etc.). La técnica de "mapeo" operaría sin problema. En cambio, con Scan6 ignoraría el prefijo para cada una de las técnicas, y con ello ejecutaría SLAAC, las otras técnicas funcionarían igual por no tener inconvenientes con las dimensiones del prefijo.

La diferencia de como interpretar las sub-redes y sus sufijos puede no ser tan importante, pues cuando se desconoce la red objetivo y se provee el prefijo más grande posible (48 ó 64 usualmente), se podrían buscar todo los rangos posibles. Pero por otra parte, cuando ya se empieza a descubrir información de la red, mediante cualquier técnica, y se han confirmado las dimensiones de algunas sub-redes, es más eficiente que se ejecuten las técnicas adecuadas a ese prefijo, y bajo esta premis se da el comportamiento de Itsismx.

Una distinción importante, es la dificultad de controlar la agresividad de las pruebas, pues scan6 solo permite controlarlos mediante su tasa de transmisión medido en bits por segundo. Aunado a que no verifica los mensajes del enrutador, dejándonos en

riesgo de creer que se esta realizando adecuadamente, cuando en realidad los enrutadores están sufriendo la negación de forma constante. **Nmap, cuando detecta que no hay rutas disponibles detiene, el envío de los *probes*, mientras que scan6 mantiene constante su transmisión**.

Una distinción menor, pero importante, es el hecho que Scan6 no da indicadores de cuantos nodos ha explorado y cuanto tiempo le falta por acabar una exploración. Incluso, si se ejecuta sin *verbosity*, no desplegara nada hasta que termine. En cambio, Nmap lleva registro del avance e incluso, durante una ejecución sin *verbosity* puede desplegar dicho avance si el usuario presiona cualquier tecla.

Una característica de scan6, es que esta completamente especializado a la exploración. Tiene un control de memoria bastante eficiente del nodo anfitrión reduciendo el riesgo de agotarla durante las exploraciones grandes. Sin embargo, si se desea realizar auditorías los resultados deben ser exportados a otros programas, mientras que Nmap puede realizar dichas auditorías conforme descubra los nodos, con su respectivo incremento de uso de memoria y recursos.

En conclusión, **optar por usar Nmap en esta investigación fue la decisión más acertada, pues permitió concentrar recursos y tiempo, mucho menores a los invertidos en 6Tools, para poder desarrollar las técnicas aquí propuestas**, ofreciendo así una gran flexibilidad para cada una de ellas.

Capítulo 5

Conclusiones

Esta investigación se realiza con el objetivo de validar las diferentes ideas, conjeturas y mecanismos relacionados con una exploración en redes IPv6 de forma eficiente. **Como resultado se desarrollaron 5 guiones (*scripts*) denominados *Nmap Script Engine* (NSE) los cuales son ejecutados por la aplicación de código abierto Nmap**, una herramienta útil y versátil para realizar pruebas de auditoría de seguridad y descubrimiento de redes [7].

Nmap integra por defecto la capacidad de exploración de nodos en IPv4, Sin embargo, para IPv6 el soporte inicial en su versión 6.20 (que era la disponible al iniciar esta investigación) estaba restringida a exploraciones de nodos especificados por el usuario. Posteriormente, conforme esta investigación se desarrollaba, se liberó la versión 6.40 que añade la capacidad de una exploración de fuerza bruta, lo cual como se ha indicado anteriormente, no es practico. En cambio, las técnicas aquí mostradas permiten que Nmap realice la fase de exploración de nodos de forma más eficiente, para que los auditores puedan realizar sus actividades. Es decir, **el objetivo es ofrecer código reutilizable y funcional para la exploración de nodos IPv6**.

Esta investigación ofrece también la **capacidad de validar la existencia de una sub-red IPv6 mediante servidores DHCPv6** para ser utilizado en conjunto con otras técnicas, ofreciendo una capacidad modular capaz de darle al usuario la flexibilidad de decidir como realizar la exploración. Estas técnicas permiten la exploración de nodos mediante 3 formas:

- Configurados manualmente, sea mediante técnicas de asignación direcciones bajas, tales como 2001:db8::1, 2001:db8::ff, o mediante juegos de palabras fáciles de recordar, 2001:db8::c0ca:c0la,2001:db8::l0c0.

- Basadas en la capacidad de auto-configuración de los nodos mediante, el mecanismo EUI-64. Sin importar si es una máquina física o virtual.

- Basadas en una traducción directa del dominio de IPv4 a IPv6, por ejemplo 2001:db8::192.168.1.1, ::192.168.1.1, etc.

Cada una de las técnicas tiene su propio conjunto de argumentos para darle mayor flexibilidad, los cuales pueden ser hallados en el apéndice correspondiente. Es importante recalcar que, a pesar de su eficiencia, algunos de estos algoritmos trabajan con la exploración de 24 *bits*, es decir deben de calcular hasta 16,777,216 posibles direcciones, por lo tanto una exploración completa puede tomar más de 24 horas. Pero, es similar a cuando se realizan exploraciones de igual dimensiones en IPv4.

Para la realización de las pruebas se escogió replicar de forma básica una red de una compañía cualquiera, simulando diferentes sub-redes y enlaces de baja velocidad, a su vez, se decidió dar heterogeneidad a los nodos manejando tanto máquinas físicas como virtuales. Dichas máquinas utilizan diferentes sistemas operativos para poder cubrir el mayor rango posible de pruebas.

Durante la ejecución de las pruebas se pudo comprobar que existen limitantes nuevas en IPv6 respecto a IPv4. Las más importante están relacionadas al manejo de memoria en los nodos. Todos los dispositivos deben guardar un caché de los nodos vecinos, y como consecuencia es posible provocar un agotamiento de memoria en ellos al estar buscando todos los posibles nodos en una sub-red. Esto puede resultar **en negaciones de servicio en una o más partes de la la red durante las exploraciones**. La segunda problemática viene en el manejo de memoria del nodo que realiza la exploración, ya que la misma puede tan extensa, que si no se manejan de forma adecuada podría agotar toda la memoria RAM del nodo, o bien requerir mucha memoria para evitar este riesgo.

Los resultados cumplen los objetivos iniciales al ofrecer una respuesta adecuada con exploraciones pequeñas, siendo solamente la técnica basada en SLAAC la que se ve limitada, pues con una proporción de una exploración de 10 *bits*, solo busca 1024 posibles nodos de un total de más de 16 millones así que por probabilidad, no tendrá suerte en su ejecución, a menos que se incremente el rango de búsqueda o exploremos los 24 *bits*. Además, sufren del hecho que los nodos mantenidos en servidores virtuales de VMware no necesariamente requieren solo realizar un barrido de 8 *bits*, como se pensaba, pues los servidores pueden tomar su decisión de asignación de dirección MAC de forma dinámica basada en otros parámetros distintos la dirección IP del servidor.

5.1. Consideraciones para exploración de nodos

Toda las técnicas son eficientes, pero la eficiencia de cada uno se vera afectada de acuerdo a como este configurada la red IPv6 de la entidad a explorar. Es muy probable que los ***scripts* basados en *Low-Bytes* y palabras tengan la mayor ventaja para detectar dispositivos intermedios** como Gateways, debido a que es más fácil para un administrador de redes trabajar con esos formatos. Y otro beneficio de iniciar una exploración con estos *scripts*, es que por lo general, tendrán la menor cantidad de nodos a explorar y por lo tanto puede ser realizado en cuestión de segundos.

Posterior a estas dos técnicas en utilidad, estaría el guión basado en SLAAC, aunque es un poco contradictorio. Por un lado, *reduce la exploración de 64 bits a múltiplos de 24 bits*, sin embargo, este espacio de direcciones probablemente supere por mucho la cantidad de nodos existentes en una sub-red (pues tienden a lo mucho a ser unos miles) y debido al riesgo de la negación de servicio en la misma red, puede volverse casi imposible de realizar un barrido completo. Por consiguiente se tendrían que hacer barridos aleatorios cuya probabilidad de acierto seria bastante baja debido a ese espacio de más de 16 millones de nodos que representan esos 24 *bits*.

Si se sabe que se manejan servidores de VMware, SLAAC puede volverse muy útil, pues existe la posibilidad de bajar la exploración a múltiplos de 8 bits, sin embargo, si no se puede confirmar la dirección IPv4 de la máquina anfitriona del servidor VMware o si esta esta configurada con direcciones manuales e incluso si el mismo servidor no utiliza la dirección IPv4 para la asignación de la dirección MAC, nos veremos en la necesidad de explorar los 24 bits para poder descubrir las máquinas virtuales.

Los nodos con direcciones basadas en el formato EUI-64 no cambiaran los últimos 64 bits de dirección sin importar en que sub-redes se hallen, por lo mismo, conviene agregar dichos nodos al listado de diccionario, pues entonces se les podrá dar seguimiento sin gastar tantos recursos para encontrarlos. Esto puede resultar útil también para enrutadores si sus direcciones fueron configuradas con dicho formato, ya que seguramente en cada interfaz se repita la dirección MAC como se muestra en la figura 5.1. **La técnica de diccionario de palabras, puede ir incrementado su utilidad en aquellas redes cuyos nodos se vayan descubriendo, o bien para buscar nodos con direccione estáticas**.

Las 4 técnicas tienden a dejar de fuera la necesidad del servicio DHCPv6 *stateful* pues los nodos que utilicen direcciones bajas y/o mnemónicos, son configurados de forma manual (*stateful*), los nodos que utilizan el formato EUI-64 son por naturaleza *stateless* y no requieren el uso de un servidor DHCPv6, salvo para pedir algunos datos

```
R1#show ipv6 interface brief
FastEthernet0/0            [up/up]
    FE80::222:90FF:FEB1:4780
    2001:DB8:C0CA::C0A8:501
    2001:DB8:C0CA:CED0::1
FastEthernet0/1            [up/up]
    FE80::222:90FF:FEB1:4781
    2001:DB8:C0CA:AE00::1
Serial0/0/0                [up/up]
    FE80::222:90FF:FEB1:4780
    2001:DB8:C0CA:FE0::2
Serial0/0/1                [up/up]
    FE80::222:90FF:FEB1:4780
    2001:DB8:C0CA:EE00::2
Loopback0                  [up/up]
R1#
```

Figura 5.1: 3 de 4 Interfaces de un enrutador con misma dirección FE80 (configurado con EUI-64)

adicionales (servidor DHCPv6 *stateless*). Sin embargo, si la red utiliza un prefijo distinto al 64 para una o más redes, entonces los mecanismos de asignación solo pueden ser *stateful*, y es ahí donde el script *itsismx-dhcpv6* entra en acción. Ya que seguramente, dichas redes se configuren mediante DHCPv6.

El punto débil de la técnica de DHCPv6, es que el servidor regresa como respuesta una dirección sin prefijo, por lo tanto, si le proveemos al guión las sub-redes: 2001:db8:c0ca::/80,
2001:db8:c0ca:1::/80, y 2001:db8:2::/80 y recibimos 3 respuestas positivas, podría ser en realidad que exista la sub-red 2001:db8:c0ca::/70 y al momento de realizar barridos podríamos llegar a dejar fuera del rango 10 bits completos que se traducen a más de 1 millón de posibles nodos. No necesariamente es malo, podríamos enfocarnos primero en estos rangos y luego ir validando el extremo de la red con más solicitudes de dirección, o intentando obtener mensajes de ICMPv6 de ruta inalcanzable al enviar mensajes a nodos en los que creemos sean otras sub-redes contiguas. Por ejemplo, la dirección 2001:db8:c0ca:5::1 si provocase dicho mensaje de ICMPv6, significa que la sub-red 2001:db8:c0ca:5::/80 puede ser inválida.

En ultima instancia de prioridad queda el guión basado en "mapeo" 4 a 6, ya que

solo será útil si la entidad esta utilizando dicho esquema. Posiblemente, si la entidad lo utiliza también involucre un servidor DHCPv6. Si se llega a dar el caso que este implementado de este modo, se tendrán los espacios de exploración mas pequeños, pues seguramente serán dimensiones menores a 24 bits, incluso podrian ser solo de 8 *bits*.

Una ultima consideración, es que durante esta investigación se utilizaron prefijos de 64 para la mayoría de las sub-redes, salvo la parte de "mapeo", pero los guiones pueden recibir bloques de prefijos de diferentes dimensiones y los bits a usar para las exploraciones se ajustarán a estas dimensiones. Por lo mismo, los guiones basados en SLAAC y "mapeo" 4 a 6 requieren que el prefijo no sea menor a 64 y 96 respectivamente para ser ejecutados exitosamente. Si uno o más guiones no pueden operar con dicho prefijo simplemente desplegará un mensaje indicándolo.

5.2. Áreas de oportunidad

Como conclusión **las técnicas desarrolladas en *scripts* NSE para Nmap son funcionales y bastante estables**, sin embargo poseen diferentes aspectos que pueden ser mejorados para incrementar el rendimiento de estas técnicas. Tales como:

- Implementar las técnicas dentro del código fuente de Nmap, esto perfeccionaría aun más la paralización de las direcciones y el manejo de memoria.
- Paralizar el proceso de crear las direcciones. En escaneos grandes un tiempo considerable se pierde por ser serial en vez de paralizado.
- Implementar la capacidad de continuar el trabajo a partir de una dirección dada, particularmente cuando se realizan los ataques de fuerza bruta.
- Implementar técnicas para elegir direcciones aleatorias de formas más inteligentes. Particularmente para hacer más probable el obtener direcciones MAC validas.
- Reforzar el diccionario de direcciones basadas en palabras.
- Añadir mecanismos para extraer direcciones de privacidad de las ya descubiertas.

Los primeros dos aspectos se enfocan en poder reducir la carga de trabajo en los procesadores y eliminar tiempos muertos antes de iniciar el verdadero escaneo. Esto es importante debido a que Nmap ofrece para los *scripts* NSE funciones para paralización. Sin embargo, estas no están enfocadas a lo que se llamaría multi-hilos, si no para abrir diferentes *sockets* [7]. Si queremos mejorar el uso del procesador, ciertas porciones de

estos scripts deben ser escritos en C.

Como un elemento para incrementar la eficiencia de las exploraciones pequeñas, la idea de poder indicar rangos iniciales resulta ideal, particularmente para las técnicas basadas en SLAAC. Estas ultimas también se beneficiarían de una selección aleatoria más adecuada. Por ejemplo, es poco probable que un producto tenga una MAC como UU:UU:UU:UU:00:00:00:01 ó UU:UU:UU:UU:aa:bb:cc:dd, por lo tanto no deberían de ser ofrecidas .

Las direcciones de privacidad cumplen su objetivo inicial, **ser difíciles de descubrir**, sin embargo es muy factible que el nodo tenga otras direcciones además de las de privacidad, por lo tanto existe la posibilidad de darle seguimiento a un nodo una vez que se descubran algunas otras de sus direcciones. Probablemente, esta cuestión amerite todo un estudio a fondo de igual magnitud que esta investigación.

5.3. Recomendaciones de seguridad

La enumeración de nodos es parte integral para auditorías de seguridad, pero también lo son de los primeros pasos en los ataques en contra de las mismas entidades. Por lo tanto, poder protegerse de este tipos de exploraciones es importante.

El principal mecanismo para protegerse, el uso de las direcciones de privacidad, con un tiempo de vida relativamente corto. Sin embargo, este mecanismo implica una dificultad grande para servidores, pues los nodos deberán ser capaces de contactarlo en todo momento (esto probablemente mediante DNS). Los nodos y dispositivos intermedios se pueden beneficiar de las direcciones de privacidad (los Gateways cada que cambien su dirección avisarían a los *host* mediante NDP)

Si las direcciones de privacidad no están disponibles, se puede configurar servidores DHCPv6. Particularmente el servidor de Microsoft intenta asignar direcciones aleatorias a cada nodo para dificultar que se descubra el rango de dichas direcciones. Adicional a ello, los servidores DHCPv6 tienen dos tipos de manejo de las direcciones a asignar a los nodos. La primera es IA-NA (*IA-Nontemporary Address*) y consiste en asignar de forma permanente una dirección a cada nodo (identificado mediante su DUID), la segunda es IA-TA (*IA-Temporary Address*), la cual es equivalente a utilizar direcciones de privacidad si se manejan tiempos de vida cortos.

Tanto para los escenarios con direcciones de privacidad y servidores DHCPv6,

los dispositivos deben estar configurados para que no se asignen otras direcciones por defecto, particularmente las direcciones con el formato EUI-64. Esto puede ser una dificultad pues requiere ser configurado manualmente en cada dispositivo.

Otra alternativa para dificultar la exploración con nodos EUI-64 sería cambiar las direcciones MAC para que representen OUI distintos (por ejemplo, que las direcciones MAC de máquinas DELL utilicen rangos que pertenecen a HP), pero esto es un proceso manual y dependiendo del tipo de máquina sería la dificultad de realizarlo.

Las direcciones bajas se podrían cambiar, por ejemplo, en vez de estar en el ultimo segmento, cambiarlo a uno intermedio, sin embargo, debido a los pocos bits que ocupan, estos podrían ser hallados mediante mecanismos de diccionarios.

El RFC 5157 ofrece sugerencias similares a las aquí desplegadas [17], pero adicionales a ellas sugiere el uso de direcciones *Cryptographically Generated Address* (CGAs), las cuales son utilizadas cuando se implementa *Secure Neighbour Discovery* (SEND) [17, 39].

Por ultimo, si se quiere evitar este tipo de exploraciones, el administrador de red debe evitar (a como de lugar) el uso de direcciones basadas en IPv4 pues son las que garantizan las exploraciones de dimensiones más pequeñas. Incluso se puede realizar primero la exploración con IPv4 ya difícilmente provocaría una negación de servicio sobre la red.

Apéndice A

Nmap

A.1. Interfaz y cualidades de escaneo

Nmap es una herramienta originalmente creada para sistemas basados en GNU/Linux pero que actualmente incluye un excelente puerto a sistemas de operativos de la compañía de Microsoft, además de de los sistemas operativos de la compañía Apple. Por defecto, esta herramienta tiene una interfaz de linea de comandos, siendo sensible a minúsculas y mayúsculas, aunque posee una interfaz gráfica denominada Zenmap [7], la cual no será utilizada para esta investigación. Nmap responde a la siguiente sintaxis

```
nmap [ <tipo de escaneo>...] [ <Opciones>] { <especificaciones de objetivos>}
```

Un escaneo por defecto de Nmap, donde solo se añade los objetivos, evaluara los 1000 puertos más usados en redes TCP/IP. Los puertos que respondan a un *probe* son clasificados dentro de seis categorías que están desplegados en la tabla A.1 [6]. Para IPv4, Nmap ofrece una gran variedad de opciones para escanear múltiples nodos simultáneamente, para IPv6 también ofrece el mismo servicio pero es requerido introducir los nodos manualmente, ya sea en la línea de comando o mediante un archivo introducido como campo del argumento iL"[7, 6]. Además del escaneo por defecto, es posible realizar un escaneo agresivo donde se busca mucha más información del nodo, o bien uno a medida de lo que necesite el usuario [6].

Una característica importante de Nmap, tanto en IPv4 como IPv6 es que antes de intentar escanear un puerto, este intentará enviar un mensaje ICMP del tipo *echo request* para detectar si el nodo esta disponible. Pero se le puede indicar que realice las pruebas de conexión mediante los puertos 80 y 443, que son puertos de servicios [6]. En IP6 esta regla resulta en que previo a enviar cualquier *probe* a un nodo remoto, primero validara la disponibilidad de su Gateway y en caso de error imprimirá el mensaje.

Estado	Descripción
Abierto	Puerto que responde activamente a una conexión entrante.
Cerrado	Puerto que responde activamente a una conexión entrante pero no posee ningun servició en ejecución. Usualmente hallado en sistemas donde no hay un firewall implementado.
Filtrado	Puertos que están protegidos por un firewall o por algún mecanismo que previene conocer si el puerto esta abierto o cerrado.
Sin filtrar	Puerto que no esta filtrado pero Nmap no le es posible determinar si esta abierto o cerrado.
Abierto|Filtrado	Nmap cree que esta en una de esas dos concidicones pero no le es posible determinarlo.
Cerrado|Filtrado	Nmap cree que esta en una de esas dos concidicones pero no le es posible determinarlo.

Cuadro A.1: Estados de puertos en Nmap [6].

Para una mejor opción de escaneo de nodos remotos, considerando la posibilidad de que existan *Firewalls* o listas de acceso implementado, los siguientes argumentos de Nmap deben ser considerados [6]:

-6 Para que Nmap realice sus funciones en IPv6 es necesario indicarlo con dicho argumento. **IPv4 e IPv6 no pueden operar simultáneamente en una misma ejecución de Nmap**.

-Pn No ping previo a realizar el análisis.

-PS TCP SYN ping es un *probe* enfocado al protocolo TCP - por defecto al puerto 80 - que resulta útil cuando ICMP esta bloqueado.

-Pa TCP ACK ping es un *probe* enfocado al protocolo TCP - por defecto al puerto 80 - que resulta útil cuando ICMP esta bloqueado.

-PY TCP INIT ping es un *probe* enfocado al protocolo TCP - por defecto al puerto 80 - que resulta útil cuando ICMP esta bloqueado.

-R Nmap por defecto solo realiza un *Reverse DNS resolution* cuando detecta que el nodo esta en línea. Con este argumento lo forzamos a realizarlo primero. **Este argumento puede tener un impacto negativo en el rendimiento del escaneo** .

-n permite deshabilitar la traducción de DNS de los nodos descubiertos.

–**data-lenght** Anexa datos de tamaño variable al *probe* para intentar engañar *Firewalls* entrenados para reconocer paquetes de Nmap (que por defecto son tamaño fijo).

–**randomize-hosts** Permite revisar los *hosts* de forma aleatoria.

La enorme flexibilidad de como escanear nodos en Nmap es la razón principal por la que se escogió esta herramienta para el desarrollo de la investigación. Dentro de sus múltiples formas de escanear un nodo tenemos, entre otras, las siguientes opciones [6]:

-sS TCP SYN Scan, ejecuta pruebas a los mil puertos de TCP más usados mediante paquetes SYN. **Por defecto a usuarios con privilegio.**

-sT TCP connect Scan, ejecuta pruebas a los mil puertos de TCP más usados mediante paquetes SYN. **Por defecto a usuarios sin privilegio.**

-sU UDP Scan, ideal para buscar servicios corriendo sobre el protocolo UDP en vez de TCP (o simultáneamente).

-sN *TCP Null Scan*, se envía un paquete TCP mal formado para tratar de forzar una respuesta del nodo indicando esto.

-F Fast Scan , sol ose escanean los 100 puertos más utilizados comúnmente.

-p <Puerto> Permite especificar puertos, sea númericamente, por nombre o separando para TCP y/o UDP, puede aceptar el wildcard "*" con el cual se escanean todo los puertos posibles.

-O Detección del sistema operativo. Una característica especial de Nmap es que a partir de las respuestas obtenidas puede detectar el sistema operativo utilizado por el nodo.

-sV Detección de la versión del servicio. Permite intentar identificar al vendedor y versión del software que este operando un puerto detectado como abierto.

- -scan-delay El tiempo mínimo entre cada escaneo de puerto y nodo.

-T0 a -T5 Plantilla de tiempos. Que tan rápido queremos realizar las pruebas por cada puerto en cada *host*, siendo T0 el más lento y T5 el más rápido, el T3 es el que esta por defecto.

Como se puede apreciar en esta lista, la información que se obtenga dependerá directamente de los argumentos a utilizar. *Con toda esta capacidad, solo es requerido enfocarnos en las técnicas propuestas,* ya que Nmap se encargará de analizar los nodos dados por nosotros. Simplificando enormemente el trabajo involucrado para esta investigación.

A.2. Fases de Nmap

Cuando se ejecuta Nmap, este puede recibir múltiples argumentos de entrada, tales como direcciones especificas o rangos de estas, los puertos que se deben de trabajar e incluso los tiempos que debe de dedicar a cada uno, así como las restricciones que debe de cumplir mientras se este ejecutando el análisis, entre otros. Todos estos argumentos entraran en juego según la fase en la que se encuentre la ejecución. Dichas fases son las siguientes:

1. *Script-Pre-scanning*
2. *Target Enumeration*
3. *Host Discovery* (IPv4IPv6)
4. *Reverse-DNS Solution*
5. *Port Scanning*
6. *Version Detection*
7. *OS Detection*
8. *Traceroute*
9. *Script Scanning*
10. *Output*
11. *Script post-scanning*

No toda las etapas son necesarias de pasar, y el alcance de ellas puede estar controlado mediante argumentos dados al ejecutar Nmap. De hecho, un escaneo básico de Nmap por lo general omite la fase 1,8,9 y 11. Las cuales se utilizarán cuando el usuario ejecute ciertos comandos. Por ejemplo, observe la siguiente secuencia de ejecución de Nmap:

```
     >nmap 192.168.1.254 ..
Starting Nmap 6.25 ( http://nmap.org ) at 2013-07-17 10:58
Nmap scan report for dsldevice.lan (192.168.1.254)
Host is up (0.0012s latency).
Not shown: 999 filtered ports
PORT STATE SERVICE
1723/tcp open pptp
MAC Address: 00:26:44:AE:E9:AD (Thomson Telecom Belgium)

Nmap done: 1 IP address (1 host up) scanned in 5.44 seconds
```

En ella, se han ejecutado las fases de enumeración (2), descubrimiento de nodos (3), Reves al DNS (4), escaneo de puerto (5) y salida (10). Ahora observe esta secuencia cuando añadimos otras fases mediante el argumento.

```
>nmap -A 192.168.1.254

Starting Nmap 6.25 ( http://nmap.org ) at 2013-07-17 11:11 Central Daylight
Time (Mexico)
Nmap scan report for dsldevice.lan (192.168.1.254)
Host is up (0.0010s latency).
Not shown: 999 filtered ports
PORT STATE SERVICE VERSION
1723/tcp open pptp THOMSON (Firmware: 1)
MAC Address: 00:26:44:AE:E9:AD (Thomson Telecom Belgium)
Warning: OSScan results may be unreliable because we could not find at least
1 open and 1 closed port
Aggressive OS guesses: Thomson ST 585 or ST 536i ADSL modem (98%),
No exact OS matches for host (test conditions non-ideal).
Network Distance: 1 hop
Service Info: Host: SpeedTouch

TRACEROUTE
HOP RTT ADDRESS
1 1.02 ms dsldevice.lan (192.168.1.254)

OS and Service detection performed. Please report any incorrect results at
http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 61.81 seconds
```

`--min-hostgroup ; --max-hostgroup`	Ajustar el tamaño del grupo para los sondeos paralelos.
`--min-parallelism <numsondas>; --max-parallelism <numsondas>`	Ajustar el número de sondas enviadas en paralelo.
`--min-rtt-timeout , --max-rtt-timeout , --initial-rtt-timeout`	Ajustar expiración de sondas.
`--max-retries`	Especifica el número máximo de sondas de puertos que se retransmiten.
`--host-timeout`	Abandona equipos objetivo lentos.
`--scan-delay ; --max-scan-delay`	Ajusta la demora entre sondas.
`-T Paranoid|Sneaky| Polite|Normal| Aggressive|Insane`	Fija una plantilla de tiempos. Permiten que el usuario especifique cuan agresivo quiere ser, al mismo tiempo que deja que sea Nmap el que escoja los valores exactos de tiempos.

Cuadro A.2: Opciones de control de tiempo y rendimiento. [7]

El argumento -A es un argumento especial debido a que agrupa los argumentos básicos para activar las fases de detección de versión (6) , detección del S.O. (7) , trazo de ruta (8) y guión NSE (9). De la fase 9 carga 106 guiones pre-existentes que son relativamente seguros de ejecutar - difícilmente causaran daños a los nodos objetivos - y por lo mismo una salida puede variar de un tipo de nodo a otro. Es posible utilizar los argumentos *-dd* y *-v* para obtener más información sobre cuales son.

A.3. Control de tiempo y rendimiento

Un sondeo por omisión (`nmap <nombre-de-sistema>`) de cualquier sistema en una red local tarda un quinto de segundo [7]. Sin embargo, cuando empezamos a añadir miles de nodos a la lista, y las técnicas empleadas empiezan a ser avanzadas estos tiempos pueden cambiar. La tabla A.2 muestra los argumentos que podemos darle a Nmap para controlarlo.

A.4. Características de un guión NSE

Nmap Scripting Engine (NSE) es una de las cualidades más poderosas y flexibles de Nmap. Permite a los usuarios crear guiones (*scripts*), basados en el lenguaje de programación Lua, para automatizar una gran variedad de tareas en la red [7, **?**]. Los *scripts* son ejecutados en sus respectivas fases y la instalación por defecto posee ya una gran variedad de diferentes *scripts* pero es factible crear nuevos guiones para cubrir necesidades especificas, particularmente ejecución de vulnerabilidades.

Debido a esta de diversidad de usos, los *scripts* están catalogados en una o más categorías. Dichas categorías son: *auth, broadcast, default. discovery, dos, exploit, external, fuzzer, intrusive, malware, safe, version* y *vuln*. Para propósitos de esta investigación los *scripts* desarrollados estarán en la categoría de *discovery* pues su objetivo es descubrir nodos en línea.

Todos los *scripts* están conformados por los siguientes campos:

Descripción del *script*. Usualmente contiene también un ejemplo básico de su ejecución.

Categorías a las que pertenece el *script*.

Autor Datos relacionados al autor incluyendo forma de contactarlo.

Licencia bajo la cual se libera el *script*. Por defecto es una basada en GPL v3.0 [7].

Dependencias (Opcional) Indica si el *script* debe ejecutarse después de otros.

Reglas Permiten indicar si el script se ejecuta en una de las 3 fases que le corresponde (*pre-scanning, host/port scanning, post-scanning)*.

Acción El cuerpo principal del *script* el cual puede estar dividido en funciones y utilizar variables de ambiente para definir en cual fase se halla.

Cuando se manejan los *script* NSE es posible pasarle argumentos, así que vale la pena diferenciar los argumentos que se le pasan a Nmap de los argumentos que se le pasan a los *scripts*, en el texto cuando sea requerido, los primeros se dirán llanamente argumentos y los segundos como argumentos NSE. Dichos argumentos de NSE son los siguientes:

`-sC`

Ejecuta todos los *scripts* de la categoría *default*. Es equivalente a `--script=default`

`--script <filename>|<category>|<directory>|<expression> [,...]`

Indica la ejecución de uno o más *scripts*. Permite el uso de expresiones regulares y expresiones booleanas. Por ejemplo: `itsismx*`

`--script-args <n1>=<v1>,<n2>={<n3>=<v3>},<n4>={<v4>,<v5>}`

Propiamente permite el paso de uno o múltiples argumentos, estos por defectos los consideran texto, pero se pueden mandar en forma de tablas también [**?**].

`--script-args-file <filename>`

Es posible pasar todos los argumentos en un único archivo de texto. Ideal cuando son muchos parámetros.

`--script-help <filename>|<category>|<directory>|<expression>[,...]`

Despliega información de ayuda de los *scripts*. Específicamente la descripción.

`--script-trace`

Imprime toda la comunicación de interior al exterior y viceversa que realizan los *scripts*. Particularmente útil para ver que es lo que se transmite durante su ejecución.

`--script-updatedb`

Nmap utiliza una pequeña BD para administrar la ejecución de los *scripts*. Por lo mismo, cuando se instala un nuevo conjunto de ellos es requerido ejecutar este argumento para que actualice su BD y los incluya.

Apéndice B

Configuración de los enrutadores

B.1. Router 1

```
!
hostname R1
!
ipv6 unicast-routing
!
!
interface Loopback0
 ip address 172.16.0.1 255.255.0.0
!
interface FastEthernet0/0
  description  Subnet Servidor VMware ESXi
 ip address 192.168.5.1 255.255.255.0
 duplex auto
 speed auto
 ipv6 address 2001:DB8:C0CA:CED0::1/64
 ipv6 eigrp 10
 no shutdown
 !
interface FastEthernet0/1
 description  Subnet Servidor DHCP
 ip address 192.168.6.1 255.255.255.0
 duplex auto
 speed auto
 ipv6 address 2001:DB8:C0CA:AE00::1/64
 ipv6 eigrp 10
 no shutdown
 !
```

```
interface Serial0/0/0
 ip address 192.168.3.2 255.255.255.0
 ipv6 address 2001:DB8:C0CA:FE0::2/64
 ipv6 eigrp 10
 clock rate 64000
 no shutdown
 !
interface Serial0/0/1
 ip address 192.168.4.2 255.255.255.0
 ipv6 address 2001:DB8:C0CA:EE00::2/64
 ipv6 eigrp 10
 clock rate 64000
 no shutdown
 !
ipv6 router eigrp 10
 no shutdown
!
router eigrp 10
network 192.168.0.0 0.0.255.255
!
end
!
```

B.2. Router 2

```
hostname R2
!
ipv6 unicast-routing
interface Loopback0
 ip address 172.16.0.2 255.255.0.0
!
interface FastEthernet0/0
 description sub-net A
 ip address 192.168.1.1 255.255.255.0
 ipv6 address 2001:DB8:C0CA:6001::1/64
 ipv6 address 2001:DB8:C0CA:FEA::1/64
 !ipv6 address 2001:DB8:C0CA:FEA::/64 eui-64
 ipv6 eigrp 10
 no shutdown
```

```
 !
interface FastEthernet0/1
 description Red del atacante
 ip address 192.168.2.1 255.255.255.0
 duplex auto
 speed auto
 ipv6 address 2001:DB8:C0CA:FEB::1/64
 ipv6 eigrp 10
 no shutdown
 !
interface Serial0/0/0
 ip address 192.168.3.1 255.255.255.0
 ipv6 address 2001:DB8:C0CA:FE0::1/64
 ipv6 eigrp 10
  no shutdown
!
ipv6 router eigrp 10
 no shutdown
!
router eigrp 10
network 192.168.0.0 0.0.255.255
!
end
!
```

B.3. Router 3

```
hostname R3
ipv6 unicast-routing
!
!
interface Loopback0
 ip address 172.16.0.3 255.255.0.0
!
interface FastEthernet0/0
  description  Subnet C
 ip address 192.168.7.1 255.255.255.0
 duplex auto
 speed auto
```

```
 ipv6 address 2001:DB8:C0CA:6003::1/64
 ipv6 address 2001:DB8:C0CA:EE01::1/64
 ipv6 eigrp 10
 no shutdown
 !
interface Serial0/0/0
 ip address 192.168.4.1 255.255.255.0
 ipv6 address 2001:DB8:C0CA:E001::1/64
 ipv6 eigrp 10
  no shutdown
!
ipv6 router eigrp 10
 no shutdown
 exit
!
router eigrp 10
network 192.168.0.0 0.0.255.255
!
!
end
!
```

Apéndice C

Códigos fuentes

C.1. Argumentos a los scripts

Todo el conjunto de herramientas desarrolladas en está investigación incluyen una librería Lua base, un diccionario de palabras y 5 *scripts* NSE. Cada uno de ellos tiene a su disposición diferentes tipos de argumentos para ampliar sus capacidades. Algunos de estos argumentos son globales para definir tipos de operaciones o bien para proporcionar las redes a explorar. Es fácil identificar a cual pertenecen, pues los argumentos globales inician con `args itsismx-...` mientras que los específicos a cada *script* se dividen en dos partes y la primera indica cual es el *script*. Por ejemplo, el argumento `itsismx-LowByt.nbits` pertenece al script `itsismx-Lowbyt.nse` ya que la parte a la izquierda del punto indica su nombre, mientras que el resto indica el nombre del argumento.

Estos argumentos por lo general pueden recibir datos como *string*, algunos de ellos solo aceptan ciertos valores, y cuando se describan indicaran esos vales entre comillas, ej "number". Por el contrario, si indica que el tipo de dato es *string /Tabla* señala que espera una entrada de algún tipo, por ejemplo como una dirección IPv6 con prefijo o bien se puede proporcionar un listado de ese mismo tipo. El listado debe estar en sintaxis de Lua. Es decir, si se permite direcciones IPv6 sería `{2001:db8:c0ca:fea::/64,2001:db8:c0ca:loca::/64,2001:db8:c0ca::1}` . En algunos casos el colocar el argumento es más que suficiente para que entre en operación, para esos casos no se desplegara valor alguno.

Se tiene un total de 20 posibles argumentos a introducir durante la ejecución de los *scripts*, se puede apreciar un resumen en la tabla C.1, dichos argumentos son:

1. `itsismx-subnet` Proporciona un listado de redes IPv6 a explorar para cada *script* que se ejecute. Sintaxis: X:X:X:X::/YY

2. `itsismx-IPv6ExMechanism` Decide como calcular direcciones IPv6, basadas en operaciones de números de 32 bits u operaciones de *string*. Tiene efecto en el rendimiento y por defecto son números.

3. `itsismx-SaveMemory` Solo impacta en los mecanismos de SLAAC y Map4to6. Si se activa se hará un uso más controlado de memoria, pero por lo mismo el reporte final no incluirá información de estos *script*.

4. `itsismx-dhcpv6.subnets` Un listado de sub-redes tentativas a confirmar su existencia mediante servidores DHCPv6. Sintaxis: X:X:X:X::/YY o un listado de las mismas.

5. `itsismx-dhcpv6.TimeToBeg` Representa un número de 16 bits y sirve para hacer creer al servidor DHCPv6 que el nodo fantasma lleva más tiempo solicitando la dirección. Por defecto, este valor se considera como cero.

6. `itsismx-dhcpv6.Company` Número Hexadecimal de 6 caracteres. Representa una OUI para camuflar al cliente fantasma. Por defecto se hace pasar por un equipo DELL (OUI 24B6FD).

7. `itsismx-dhcpv6.utime` Número que representa micro-segundos para indicar espera máxima entre el envió de cada solicitud al servidor DHCPv6. Por defecto, esperamos un tiempo aleatorio de hasta 200 microsegundos.

8. `itsismx-LowByt.OverrideLock` Por defecto, el *script* solo permite barridos de hasta 24 bits. Si se da este argumento el *script* permitirá barridos mayores.

9. `itsismx-LowByt.nbits` Indica la cantidad de bits a barrer. Si no se proporciona se hará un barrido de 8 bits.

10. `itsismx-wordis.nsegments` Número cuyo rango es 1 - 4. Representa que tan grande son las palabras a buscar, dividas en segmentos de 16 bits cada uno. Por defecto se buscan todas las palabras en el diccionario.

11. `itsismx-wordis.fillright` Indica en que orientación se forman las palabras (Ej: 2001:db8::C0CA ó 2001:db8:C0CA::). Por defecto se utiliza el llenado en los bits menos significativos (izquierda).

12. `itsismx-Map4t6.IPv4Hosts` Un listado de direcciones IPv4 y/o sub-redes IPv4 a convertir a IPv6. Puede recibir entradas mezcladas como: A.B.C.D ó A.B.C.D/YY.

13. `itsismx-slaac.vendors` Permite nombres de compañías (DELL, IBM, Toshiba, etc.) y/u OUI especificas. Por defecto agrega una OUI de DELL pero es altamente recomendable que se de una entrada distinta.

14. `itsismx-slaac.nbits` Representa cantidad de bits a barrer. El rango es de 1 a 24. Según el mecanismo de computo es como se interpreta. Puede ser barrido de esos bits o bien 2 ^ nbits muestras aleatorias. Por defecto son

15. `itsismx-slaac.compute` Indica como realizar el muestreo. Bruto significa un barrido de fuerza bruta. Si la cantidad de bits es 23 o 24 será automáticamente bruto. Por defecto es aleatorio.

16. `itsismx-slaac.vms` Permite indicar si se desea buscar algún tipo de máquina virtual en especifico. Solamente el caso de VMware ofrece argumentos adicionales. Toda las demás son tratadas de forma similar a máquinas físicas reales.

    a) (vacio) si no se introduce ningún valor, añadirá a la lista de trabajo las VM de VMware, Virtual Box, Paralalles, Virtual PC y QEMU VMs.

    b) `W` añadirá a la lista de trabajo las VM de VMware.

    c) `wS` añadirá a la lista de trabajo las VM de VMware cuya configuración de dirección MAC es estática/manual.

    d) `wD` añadirá a la lista de trabajo las VM de VMware cuya configuración de dirección MAC es dinámica.

    e) `P` añadirá a la lista de trabajo las VM de Parallels Virtuozzo y Dekstop VMs.

    f) `pV` añadirá a la lista de trabajo las VM de Parallels Virtuozzo.

    g) `pD` añadirá a la lista de trabajo las VM de Dekstop VMs.

    h) `pVpD` Equivalente a P

    i) `V` añadirá a la lista de trabajo las VM de Oracle Virtual Box.

    j) `M` añadirá a la lista de trabajo las VM de Microsoft Virtual PC VMs.

    k) `L` añadirá a la lista de trabajo las VM de Linux QEMU.

    l) `WPVML` Equivalente a la opción por defecto.

17. `itsismx-slaac.vms-nbits` Si se desea buscar máquinas VMware con asignación de dirección MAC dinámica se usa un barrido de 1-16 bits en lugar del 1 - 24. Este argumento controla este nuevo rango que por defecto es 2. (hablamos de 255* 4 direcciones).

18. `itsismx-slaac.vmipv4` Al argumento anterior es posible ofrecer directamente direcciones de *host* IPv4 tentativas.

19. `itsismx-slaac.knownbits` Alternativo a los dos argumentos anteriores, es posible proveer directamente parte de los 16 bits tentativos reduciendo el número de muestras a buscar.

#	Nombre	Tipo de dato	Uso
1	`args itsismx-subnet`	String |Tabla	Preferente.
2	`args itsismx-IPv6ExMechanism`	"number" |"string"	Opcional.
3	`itsismx-SaveMemory`		Opcional.
4	`itsismx-dhcpv6.subnets`	String |Tabla	Preferente.
5	`itsismx-dhcpv6.TimeToBeg`	Número	Opcional.
6	`itsismx-dhcpv6.Company`	String	Opcional.
7	`itsismx-dhcpv6.utime`	Número	Opcional.
8	`itsismx-LowByt.OverrideLock`		Opcional.
9	`itsismx-LowByt.nbits`	Número	Opcional.
10	`itsismx-wordis.nsegments`	Número	Opcional.
11	`itsismx-wordis.fillright`		Opcional.
12	`itsismx-Map4t6.IPv4Hosts`	String |Tabla	Mandatario.
13	`itsismx-slaac.vendors`	String |Tabla	Preferente.
14	`itsismx-slaac.nbits`	Número	Opcional.
15	`itsismx-slaac.compute`	"random" |"brute"	Opcional.
16	`itsismx-slaac.vms`	Múltiples (Ver más abajo)	Opcional.
17	`itsismx-slaac.vms-nbits`	Número	Opcional.
18	`itsismx-slaac.vmipv4`	String |Tabla	Opcional.
19	`itsismx-slaac.knownbits`	String	Opcional
20	`newtargets`		**Mandatorio**

Cuadro C.1: Total de argumentos para el conjunto de herramientas desarrolladas.

20. `newtargets` Permite añadir los nodos a las siguientes fases de Nmap.

Por lo menos, el argumento `itsismx-dhcpv6.subnets` o `itsismx-dhcpv6.subnets` es requerido para poder realizar la ejecución de los *scripts*. El primero se debe de realizar cuando se utiliza el *script* para DHCPv6 con el cual se pasan las sub-redes tentativas a buscar en los demás *scripts*. Solo en caso de obtener respuesta positiva para una o más de las sub-redes, estas serán pasadas a los demás *scripts*. En cambio, el segundo argumento es universal y todos los *scripts* enfocados a descubrir nodos, toman las sub-redes proveidas por dicho argumento.

C.2. Códigos fuentes

A continuación se despliegan los códigos fuentes. Los códigos fuentes se hallan disponibles en el repositorio https://code.google.com/p/itsis-mx/index.html.

C.2.1. itsismx.lua

```
---
-- This library implements common tools for all the  NSE scripts
-- of ITSISMX.
-- The library contains the following classes:
--

--@author Raul Fuentes <ra.fuentes.sam@gmail.com>
--@copyright Same as Nmap--See http://nmap.org/book/man-legal.html

--
-- Version 0.2
--
-- Created 01/03/2013 - v0.1 - created by Raul Fuentes <ra.fuentes.sam@gmail.com>
local ipOps = require "ipOps"
local nmap = require "nmap"
local stdnse = require "stdnse"
local table = require "table"
local strict = require "strict"
local math = require "math"
_ENV = stdnse.module("itsismx", stdnse.seeall)

--[[ Tools for  brute sweep of IPv6 Addrss

 There are two  differents methods:
 1 - Strings  representing bits
 2- 4 numbers of 32 bits

 Originally think to use class for those but  have problem
 creating independently instances of the class (because LUA it-  not
 a oriented object language)  so at the end choice the use the classical
 aproach of functions.

--]]

-- Return a number from a binary value reprsent on string
-- Curiosity: "Mordidas" is a VERY BAD Translation of bits
-- @args  A string of 32 characters that area binary (0 | 1 )
--  @returns Numeric value of those 32 bits
local Bits32_BinToNumber = function ( Mordidas )
 local iValor,iPos = 0,0
 --print("Longitud " .. #Mordidas .. " de " .. Mordidas)

 if string.len(Mordidas) ~= 32 then -- We only work with 32 bits
  return iValor
 end

 --There is something odd how Nmap use Dword for IP address of 32
 --bits making me  choice a more "pure" way to do it
 Mordidas:reverse() --More easy to my head
 for iPos=1, 32 ,  1 do
  --print("\t\t\t" .. #Mordidas:sub(iPos,iPos) .. " " ..Mordidas:sub(iPos,iPos))
  iValor = iValor + tonumber( Mordidas:sub(iPos,iPos) ) * math.pow(2,32 - iPos)
  --print ( "\t\t" .. Mordidas:sub(iPos,iPos) .. " * 2^" .. 32 - iPos )
 end
 --print ("\t BinToNumber: ", iValor)
 return iValor
end

--- Return binary value (on strings) of a number.
```

```
-- @args  Numeric value of 32 bits (Actually more than that)
--  @returns A string of 32 characters that area binary (0 | 1 )
--    The first 32 bits of the args given.
local Bits32_NumberToBin= function ( Decimal )
  local Mordidas,iPos = "",0

  --Lua 5.2 give us a easy time doing this!
  for iPos=31,0, -1 do
   Mordidas = Mordidas ..  tostring ( bit32.extract (Decimal, iPos, 1 ) )
  end

  --print ("\t NumberToBin: ", Mordidas )
  return Mordidas
end

--- We sum our number of 128 bits, against another number of 128 bits  using
--  special structure.
-- @args NumberA A table having 4 variables
-- @args NumberX A table having 4 variables
--  @returns A table having 4 variables
--     If there is a error or Overflow will return all bits set to one
local Bits32_suma128 = function ( NumberA, NumberX)

 --print(NumberA.ParteAltaH ..  NumberA.ParteAltaL .. NumberA.ParteBajaH  .. NumberA.ParteBajaL)
 --print(NumberX.ParteAltaH ..  NumberX.ParteAltaL .. NumberX.ParteBajaH  .. NumberX.ParteBajaL)

 -- We sum the parts
 NumberA.ParteAltaH = NumberA.ParteAltaH + NumberX.ParteAltaH
 NumberA.ParteAltaL = NumberA.ParteAltaL + NumberX.ParteAltaL
 NumberA.ParteBajaH = NumberA.ParteBajaH + NumberX.ParteBajaH
 NumberA.ParteBajaL = NumberA.ParteBajaL + NumberX.ParteBajaL


 --We validate  Zero Overflow!
 if (NumberA.ParteBajaL >= math.pow(2,32) ) then
  NumberA.ParteBajaL = 0
  NumberA.ParteBajaH = NumberA.ParteBajaH + 1
  if (NumberA.ParteBajaH >= math.pow(2,32) ) then
   NumberA.ParteBajaH = 0
   NumberA.ParteAltaL = NumberA.ParteAltaL + 1
   if (NumberA.ParteAltaL >= math.pow(2,32) ) then
    NumberA.ParteAltaL = 0;
    NumberA.ParteAltaH = NumberA.ParteAltaH + 1
    if (NumberA.ParteAltaH >= math.pow(2,32) ) then
     --SCREW THIS!  Overflow
     NumberA.ParteAltaH = 0xFFFFFFFF
     NumberA.ParteAltaL = 0xFFFFFFFF
     NumberA.ParteBajaH = 0xFFFFFFFF
     NumberA.ParteBajaL = 0xFFFFFFFF
    end
   end
  end
 end

 --print(NumberA.ParteAltaH ..  NumberA.ParteAltaL .. NumberA.ParteBajaH  .. NumberA.ParteBajaL)

 return NumberA
end

---
```

```
-- 1-bit Binary adder with Carry.
-- Actually it.s pseudo-bit because got strings and
-- return strings (Onlye 1 and 0) . It's slower
-- than using real numbers but it's a good option for
-- IPv6.
-- @args   A A String of 1 bits
-- @args   B A String of 1 bits
-- @args   Cin A String of 1 bit - Carry In
-- @return String A string of 1 bits
-- @return String A string of 1 bit ( Carry out)
local Sumador =  function (A, B, Cin )
 --Bless coercion
 local S, Cout = 0,0
 S = bit32.bxor( A, B, Cin )
 Cout =  bit32.bor (bit32.band(A, B),bit32.band(Cin ,bit32.bor( A, B ) ))

 return  tostring( S),  tostring(Cout)

 end

---
-- 8-bits Binary adder with Carry.
-- Actually it.s pseudo-bit because got strings and
-- return strings (Onlye 1 and 0) . It's slower
-- than using real numbers but it's a good option for
-- IPv6.
local Sumador8bits = function ( A, B, Cin)

 local  S, Cout, Caux = {}, "0","0"
 local  aA, aB = "", ""
 local Sstring = ""

 -- DANGER: No leer al reves el numero
 -- Bless "Class" mechanism of Lua
 aA = A:sub(8,8)
 aB = B:sub(8,8)

 S[8], Caux = Sumador(aA, aB, Cin)
 Caux = tostring(Caux)
 --print("A[8]: " .. aA .. " B[8]: " .. aB .. " S[8]: " .. S[8] .. " C: " .. Caux )
 for I = 7,2,-1 do
  aA = A:sub(I,I)
  aB = B:sub(I,I)
  S[I],Caux = Sumador(aA, aB, Caux)

  Caux = tostring(Caux)
 end

 aA = A:sub(1,1)
 aB = B:sub(1,1)

 S[1],Cout = Sumador(aA, aB, Caux)

 for  I= 1,8 do
  Sstring = Sstring .. tostring( S[I])
 end

 return Sstring,  tostring(Cout)
end
```

```
---
-- 16-bits Binary adder with Carry.
-- Actually it.s pseudo-bit because got strings and
-- return strings (Onlye 1 and 0) . It's slower
-- than using real numbers but it's a good option for
-- IPv6.
-- @args A A String of 16 bits
-- @args B A String of 16 bits
-- @args Cin A String of 1 bit - Carry In
-- @return String A string of 16 bits
-- @return String A string of 1 bit ( Carry out)
local Sumador16bits = function ( A, B, Cin)

 local  S, Cout, Caux = {}, "0","0"
 local  aAH,aAL, aBH,aBL = "", "","", ""
 local Sstring = ""

 -- Bless "Class" mechanism of Lua
 aAH = A:sub(1,8)
 aAL = A:sub(9,16)
 aBH = B:sub(1,8)
 aBL = B:sub(9,16)

 S[2], Caux = Sumador8bits(aAL, aBL, Cin)
 Caux = tostring(Caux)

 S[1], Cout= Sumador8bits( aAH,  aBH,  Caux)

 Sstring = S[1] .. S[2]

 return Sstring,  tostring(Cout)
end

-- Sumador binario de 32 bits (String) con acarreo
-- 32-bits Binary adder with Carry.
-- Actually it.s pseudo-bit because got strings and
-- return strings (Onlye 1 and 0) . It's slower
-- than using real numbers but it's a good option for
-- IPv6.
-- Useful for IPv4 (Thought Nmap can do it better)
-- @args A A String of 32 bits
-- @args B A String of 32 bits
-- @args Cin A String of 1 bit - Carry In
-- @return String A string of 32 bits
-- @return String A string of 1 bit ( Carry out)
local Sumador32bits = function ( A, B, Cin)

 local  S, Cout, Caux = {}, "0","0"
 local  aAH,aAL, aBH,aBL = "", "","", ""
 local Sstring = ""

 -- We divide on blocks of 16
 aAH = A:sub(1,16)
 aAL = A:sub(17,32)
 aBH = B:sub(1,16)
 aBL = B:sub(17,32)

 S[2], Caux = Sumador16bits(aAL, aBL, Cin)
 Caux = tostring(Caux)
```

```
 S[1], Cout= Sumador16bits( aAH,  aBH,  Caux)

 Sstring = S[1] .. S[2]

 return Sstring,  tostring(Cout)
end

-- Sumador binario de 64 bits (String) con acarreo
-- Useful for when using the Node Portion of IPv6
-- @args A A String of 64 bits
-- @args B A String of 64 bits
-- @args Cin A String of 1 bit - Carry In
-- @return String A string of 64 bits
-- @return String A string of 1 bit ( Carry out)
local Sumador64bits = function ( A, B, Cin)

 local  S, Cout, Caux = {}, "0","0"
 local  aAH,aAL, aBH,aBL = "", "","", ""
 local Sstring = ""

 -- We divide on blocks of 16
 aAH = A:sub(1,32)
 aAL = A:sub(33,64)
 aBH = B:sub(1,32)
 aBL = B:sub(33,64)

 S[2], Caux = Sumador32bits(aAL, aBL, Cin)
 Caux = tostring(Caux)

 S[1], Cout= Sumador32bits( aAH,  aBH,  Caux)

 Sstring = S[1] .. S[2]

 return Sstring,  tostring(Cout)
end

---
-- 128-bits Binary adder with Carry.
-- Actually it.s pseudo-bit because got strings and
-- return strings (Onlye 1 and 0) . It's slower
-- than using real numbers but it's a good option for
-- IPv6.
-- Useful for when using the Node Portion of IPv6
-- @args A A String of 128 bits
-- @args B A String of 128 bits
-- @args Cin A String of 1 bit - Carry In
-- @return String A string of 128 bits
-- @return String A string of 1 bit ( Carry out)
local Sumador128bits = function ( A, B, Cin)

 local  S, Cout, Caux = {}, "0","0"
 local  aAH,aAL, aBH,aBL = "", "","", ""
 local Sstring = ""

 -- We divide on blocks of 16
 aAH = A:sub(1,64)
 aAL = A:sub(65,128)
 aBH = B:sub(1,64)
 aBL = B:sub(65,128)
```

```
 S[2], Caux = Sumador64bits(aAL, aBL, Cin)
 Caux = tostring(Caux)

 S[1], Cout= Sumador64bits( aAH,  aBH,  Caux)

 Sstring = S[1] .. S[2]
 return Sstring,  tostring(Cout)
end


--- This function will always return the next inmediatly  IPv6
-- address. This work only with String format.
-- @args IPv6Address A String IPv6 address  X:X:X:X:X:X:X:X
-- @args Prefix Optional Prefix. If it-s provided the function
--    will check to do sum with lesser bits (64, 32, 16 or 8)
-- @returns String 128 bits of a IPv6 Address
local GetNext_AddressIPv6_String = function(IPv6Address, Prefix)

 local UNO
 local Next

 UNO = ipOps.ip_to_bin("::1")

 if (not  Prefix )  then -- nil?
  Next = Sumador128bits( IPv6Address, UNO , "0")
 elseif ( Prefix > 120) then
  Next = Sumador8bits( IPv6Address:sub(121,128), UNO:sub(121,128) , "0")
  Next = IPv6Address:sub(1,120) .. Next
 elseif (  Prefix > 112) then
  Next = Sumador16bits( IPv6Address:sub(113,128), UNO:sub(113,128) , "0")
  Next = IPv6Address:sub(1,112) .. Next
 elseif (  Prefix > 96) then
  Next = Sumador32bits( IPv6Address:sub(97,128), UNO:sub(97,128) , "0")
  Next = IPv6Address:sub(1,96) .. Next

 elseif (  Prefix > 64) then
  Next = Sumador64bits( IPv6Address:sub(65,128), UNO:sub(65,128) , "0")
  Next = IPv6Address:sub(1,64) .. Next
 else -- Wasn't need the Prefix but anyway...
  Next = Sumador128bits( IPv6Address, UNO , "0")
 end

 return Next
end

--- This function will always return the next inmediatly  IPv6
-- address. This work only with a structure of 4 numbers.
-- @args IPv6Address A String IPv6 address  X:X:X:X:X:X:X:X
-- @args Prefix Optional Prefix. If it-s provided the function
--    will check to do sum with lesser bits (64, 32, 16 or 8)
-- @returns String 128 bits of a IPv6 Address
local GetNext_AddressIPv6_4Structure = function(IPv6Address, Prefix)
 local UNO
 local Next, Current

 Current =  { ParteAltaH = 0, ParteAltaL = 0, ParteBajaH = 0, ParteBajaL = 0 }
 UNO =   { ParteAltaH = 0, ParteAltaL = 0, ParteBajaH = 0, ParteBajaL = 1 }

 Current.ParteAltaH =  Bits32_BinToNumber( IPv6Address:sub(1,32)  )
 Current.ParteAltaL =  Bits32_BinToNumber( IPv6Address:sub(33,64)  )
```

```
 Current.ParteBajaH =  Bits32_BinToNumber( IPv6Address:sub(65,96)  )
 Current.ParteBajaL =  Bits32_BinToNumber( IPv6Address:sub(97,128)  )

 -- Now we add those numbers and make Casting (abusing a little of Lua)
 Next = Bits32_suma128(Current,UNO )
 Next = Bits32_NumberToBin( Next.ParteAltaH) ..
     Bits32_NumberToBin( Next.ParteAltaL) ..
     Bits32_NumberToBin( Next.ParteBajaH) ..
     Bits32_NumberToBin( Next.ParteBajaL)

 return Next
end

--[[ Global Functions

 Those are the global function that can be called by any script.
--]]

---
-- This function will always return the next inmediatly  IPv6
-- address.
-- We work with 2 very differents aproachs: Strings or Numbers
-- the first make booleans operation with strings and the second
-- make math with 4 separed numbers.
-- Note: By default use the 128 bits for adding but if the
-- the prefix its big can be a waste, that is why  there is a option
-- for reduce the number of bits to sum (String case only).
-- @args IPv6Address A String IPv6 address  X:X:X:X:X:X:X:X
-- @args (Optional) Prefix. If it-s provided the function
--    will check to do sum with lesser bits (64, 32, 16 or 8)
--    but only work if we are using "String"
-- @args  IPv6_Mech_Operator A string which represent the mechanis
--    for calculating the next IPv6 Address. Values:
--    string  - (Default) use pseudo binary operations
--    number - Divide the IPv6 in 4 numbers of 32 bits
--        (Mathematical operations)
-- @return String Formated full IPv6 X:X:X:X:X:X:X:X)
 GetNext_AddressIPv6 = function(IPv6Address, Prefix, IPv6_Mech_Operator)

 local Next = "::"
 -- 64 prefix left 64 bits to search
 -- 96 Prefix left 32 bits to search
 -- 112 prefix left 16 bits to search
 -- 120 prefix left  8 bits to search

 --First... Which mechanism?
 IPv6Address = ipOps.ip_to_bin(IPv6Address)
 if  IPv6_Mech_Operator == nil then
  --Next = GetNext_AddressIPv6_String(IPv6Address, Prefix)
  Next = GetNext_AddressIPv6_4Structure(IPv6Address, Prefix)
 elseif IPv6_Mech_Operator:lower(IPv6_Mech_Operator) == "string"  then --  We create two specials tables

  Next = GetNext_AddressIPv6_String(IPv6Address, Prefix)

 elseif  IPv6_Mech_Operator:lower(IPv6_Mech_Operator) == "number" then

  Next = GetNext_AddressIPv6_4Structure(IPv6Address, Prefix)
 end -- For the moment are the only cases, if something come wrong Next is a invalid IPv6 address
 return ipOps.bin_to_ip(Next)
end
```

```
---
--  This function will always return the next inmediatly  IPv4 address.
--   We use only Dword operations for calculating so, there is no more options for this.
GetNext_AddressIPv4 = function (IPv4ddress)

 local Next, aux, Octetos
 local d,c,b,a
 local IPN = ipOps.todword( IPv4ddress ) + 1

 aux = ipOps.fromdword (IPN)
 -- Oddly Nmap 6.25 change the octets order instead of  A.B.C.D return
 -- D.C.B.A
  Octetos = ipOps.get_parts_as_number(aux)
 if Octetos then d,c,b,a = table.unpack( Octetos ) end
 Next = a .. "." .. b .. "." .. c .. "."  .. d
 return Next
end

---
-- Receive X:X:X:X::/YY and return two separated fields:
-- IPv6 ADdress and Prefix.
--  The lesser prefix that return it's 48 because before that
-- is IANA Field.
-- @args IPv6PRefix A String IPv6 address with Prefix: X:X:X:X::/YY
-- @return String Formated full IPv6 ( X:X:X:X:: )
-- @return Number Prefix number (0-128)
 function Extract_IPv6_Add_Prefix(IPv6PRefix)
 local Campos = {}
 local Dirre6, Prefijo --= "", 0

 Campos = stdnse.strsplit("/",IPv6PRefix )

 Dirre6 = Campos[1]
 Prefijo = tonumber( Campos[2] )

 if  Prefijo < 48 then
  Prefijo = 48
 end

 return Dirre6, Prefijo
end

---
-- This function will initialize the global registry nmap.registry.itsismx
-- As this is global will check if was already called by one previous script
-- if not, will create it. In both cases  a sub entry will be generated too.
  Registro_Global_Inicializar =  function ( Registro )

 local Global = nmap.registry.itsismx

 if Global  == nil  then --The first script to run initialice all the register

  Global = {}
  Global[Registro] = {}

  nmap.registry.itsismx = Global

 elseif  Global[Registro] == nil then --- WE MUST BE CAREFUL Don't overwritte other registry
  nmap.registry.itsismx[Registro]  = {}
```

```
 end
end

---
-- Convert Decimal number to Hexadecimal
-- This piece of code was taken from:
-- http://snipplr.com/view/13086/
-- @args Number  A Lua number format
-- @return String  String representing Hexadecimal value
function DecToHex(Number)
    local hexstr = '0123456789abcdef'
    local s = ''
    while Number > 0 do
        local mod = math.fmod(Number, 16)
        s = string.sub(hexstr, mod+1, mod+1) .. s
        Number = math.floor(Number / 16)
    end
    if s == '' then s = '0' end
    return s
end

---
-- Confirm ir a given String is a oui or not.
-- the OUI are 6 hexadecimal characters.
-- @args  OUI  String representing the potential OUI
-- @return Boolean TRUE if a OUI valid format, otherwise false
Is_Valid_OUI = function (  OUI )

 --Robust and simple
 if OUI == nil then
  return false
 elseif type(OUI )  ~= "string" then
  return false
 elseif #OUI ~= 6 then
  return false
 end

 local hexstr = '0123456789abcdef'
 local Index, Caracter = 1, ""

 --Now begin the process
 for Index = 1, 6 do
  Caracter = OUI:sub(Index,Index)
  if  hexstr:find(Caracter) == nil then
   return false
  end
 end

 return true
end

---
-- Get a binary number represent with strings. Will check than only have
-- zeros and ones.
-- @args Bits String representing a binary value.
-- @return Boolean TRUE if is a valid binary number, otherwise false.
Is_Binary = function ( Bits )
 local i
  for i = 1 , #Bits  do
```

```
  if Bits:sub(i,i) ~= "0" and Bits:sub(i ,i) ~= "1" then
   return false
  end
  end
 return true
end


---
-- This function will do NOTHING but kill time. LUA do not provide something
-- similar.  This only is useful IF WE NEED TO WAIT or KILL TIME for avoid
-- detections and only if we are on pre-scanning or post-scannig due nmap
-- provide more powerfuls tools for the other two phases
waitUtime = function ( micrseconds )
  local start = stdnse.clock_us ()
  repeat until stdnse.clock_us () > start + micrseconds
end

---
-- For some part of the script, I need to work bits separted before be able to use
-- the functions of ipOPs library. So, i need be able to convert from hex to binarie.
--  NOTE: THIS NEED WORK
-- @args  Number  String  representing hexadecimal number.
-- @return  String  String representing binary number (Nil if there is a error)
HextToBin = function( Number )

 local Bits , hex, index = ""
 Number = Number:lower()
 for index = 1, #Number do
  hex = Number:sub(index,index)

  --Not the best... I need to change later ...
  if hex == "0" then
   Bits = Bits .. "0000"
  elseif hex == "1" then
   Bits = Bits .. "0001"
  elseif hex == "2" then
   Bits = Bits .. "0010"
  elseif hex == "3" then
   Bits = Bits .. "0011"
  elseif hex == "4" then
   Bits = Bits .. "0100"
  elseif hex == "5" then
   Bits = Bits .. "0101"
  elseif hex == "6" then
   Bits = Bits .. "0110"
  elseif hex == "7" then
   Bits = Bits .. "0111"
  elseif hex == "8" then
   Bits = Bits .. "1000"
  elseif hex == "9" then
   Bits = Bits .. "1001"
  elseif hex == "a" then
   Bits = Bits .. "1010"
  elseif hex == "b" then
   Bits = Bits .. "1011"
  elseif hex == "c" then
   Bits = Bits .. "1100"
  elseif hex == "d" then
   Bits = Bits .. "1101"
```

```
  elseif hex == "e" then
   Bits = Bits .. "1110"
  elseif hex == "f" then
   Bits = Bits .. "1111"
  elseif  hex == "." or hex == ":" then --Nothing bad happens
  -- return nil
  else
   --return nil
  end
 end
 return Bits
end


---
-- This will get any valid IPv6 address and will expand it WITH all the bytes
-- returning a string of bytes
-- @args  IPv6_Address String  representing IPv6 Address
-- @return  String  String representy the 16 bytes of the IPv6 Address
Expand_Bytes_IPv6_Address = function ( IPv6_Address )

 local Segmentos, HexSeg, linkAdd = {}
 local S1, S2, S3, S4, S5, S6, S7, S8
 Segmentos = ipOps.get_parts_as_number(IPv6_Address)
 S1, S2, S3, S4, S5, S6, S7, S8 = table.unpack( Segmentos )
 linkAdd = ""

 HexSeg = DecToHex(S1)
 while #HexSeg < 4 do HexSeg = "0" .. HexSeg end
 linkAdd = linkAdd .. HexSeg

 HexSeg = DecToHex(S2)
 while #HexSeg < 4 do HexSeg = "0" .. HexSeg end
 linkAdd = linkAdd .. HexSeg

 HexSeg = DecToHex(S3)
 while #HexSeg < 4 do HexSeg = "0" .. HexSeg end
 linkAdd = linkAdd .. HexSeg

 HexSeg = DecToHex(S4)
 while #HexSeg < 4 do HexSeg = "0" .. HexSeg end
 linkAdd = linkAdd .. HexSeg

 HexSeg = DecToHex(S5)
 while #HexSeg < 4 do HexSeg = "0" .. HexSeg end
 linkAdd = linkAdd .. HexSeg

 HexSeg = DecToHex(S6)
 while #HexSeg < 4 do HexSeg = "0" .. HexSeg end
 linkAdd = linkAdd .. HexSeg

 HexSeg = DecToHex(S7)
 while #HexSeg < 4 do HexSeg = "0" .. HexSeg end
 linkAdd = linkAdd .. HexSeg

 HexSeg = DecToHex(S8)
 while #HexSeg < 4 do HexSeg = "0" .. HexSeg end
 linkAdd = linkAdd .. HexSeg

 return linkAdd
```

```
end


return _ENV;
```

C.2.2. itsismx-words-known

```
# $Id: itsismx-words-known 2013-04-26 rfuentess $ por ahora hecho a manita
# For now the words come from "common" password on dictionary attacks
# and from other texts.
# The way to look this file is
# Number of segments |  Word |  word in binary
# This is for give a easy time  choicing which one to pick and give a easy time
# assembling the IPv6 address on binary.
# Some basic dictionariesÑ
# http://nedbatchelder.com/text/hexwords.html
#
1 c0ca 1100000011001010
1 10ca 0001000011001010
1  dead 1101111010101101
1 bee 0000101111101110
1 beef 1011111011101111
2 cafebabe 11001010111111101011101010111110
2 deadbeef 11011110101011011011111011101111
```

C.2.3. itsismx-DHCPv6.nse

```
local bin = require "bin"
local nmap = require "nmap"
local packet = require "packet"
local stdnse = require "stdnse"
local itsismx = require "itsismx"
local ipOps = require "ipOps"

description = [[
  The objective is work as "fake" relay agent for DHCPv6. We are going to generate "valids" request for a
  host, but we are going to spoofing the Sub-network. With this, the server  will send us a valid option or
  nothing (if the subnet is wrong or if there is  ACL or IPsec).

  The objective is not a DoS against the server neither retrive host info but simply confirm if a subset of
  sub-networks exist at all. Will generate one single relay-forwarder message (RFC 3315 20.1.2 p. 59)
  with a good HOP_COUNT_LIMIT (Spoofed) and a host request-message (Spoofed with random DUID) and we are going
  to wait for a Relay-reply message (20.3 p. 60) if we got answer, (ToDo) we send another relay-forwarder
  message with a host declined-message for don't be evil with the server.

  ACL on the server, ACL on the router,  IPsec between relays agent and server  can kill this technique.
  However almost all the RFC 3315 is more cautious with host poisoning than this type of idea.
]]
```

```
---
-- @usage
-- nmap -6 --script itsismx-dhcpv6 --script-args
--
-- @output


-- @args itsismx-dhcpv6.subnets
--
-- @args itsismx-dhcpv6.TimeToBeg
--
--@args itsismx-dhcpv6.Company
--
--@args itsismx-dhcpv6.utime


-- Version 1.0
--  Update 28/09/2013 - V1.0 First functional IA-NA mechanish finished.
-- Update 19/09/2013 - V0.7 Finished tranmsision
-- Update 04/06/2013 - V0.5 Produce the messages to spoof.
--  Created 27/05/2013 - v0.1 - created by Ing. Raul Fuentes <ra.fuentess.sam+nmap@gmail.com>
--

author = "Raul Armando Fuentes Samaniego"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"broadcast", "safe"}

Generar_DUID = function ( )

 local DUID = "0001" -- DUID-LLT begin with the constant 0x0001
 local Hardware = "0001"
 local stime = nmap.clock()
 local LinkAdd, nRand, Mac

 if ( stime > 946684800) then -- Ok, maybe on 17 years this is going to be fatal error
 stime = stime - 946684800 -- For now, we only substract 30 years (more or less)
 end

 stime = stdnse.tohex (stime )
 while  #stime < 8 do  stime = "0" .. stime end

 local Ghost = stdnse.get_script_args( "itsismx-dhcpv6.Company" )

       if Ghost ~= nil then
    if itsismx.Is_Valid_OUI(Ghost) then
      LinkAdd = "FE80000000000000" .. Ghost  .. "FFFE"
      Mac = Ghost
    else
       LinkAdd = "FE80000000000000" .. "24B6FD"  .. "FFFE"
       Mac = "24B6FD"
       stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 "DUID-LLT ERROR " .. " was provided a INVALID OUI value and was ignored. " )
    end
 else
   Mac = "24B6FD"
   LinkAdd = "FE80000000000000" .. "26B6FD"  .. "FFFE"
 end
```

```
 nRand = itsismx.DecToHex( math.random( 16777216 ) )-- 2^24
 while  #nRand < 6 do  nRand = "0" .. nRand end

 LinkAdd = LinkAdd .. nRand
 Mac = Mac .. nRand

 DUID = DUID .. Hardware .. stime .. Mac
 stdnse.print_verbose(2, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
    "DUID-LLT: " .. " New DUID: " ..   DUID  )

 return DUID , LinkAdd
end

local Generar_Option_Relay = function ( Mensaje)

 local Option_Code , Option_Len = "0009"
 Option_Len = itsismx.DecToHex( #Mensaje/2 )
 while #Option_Len < 4 do Option_Len = "0" .. Option_Len end

 return Option_Code .. Option_Len .. Mensaje
end

local Generar_Option_ClientID =  function ()
 local ClientID
 local Option_Code , Option_Len, DUID, LinkAdd = "0001","0000",
 Generar_DUID()

 Option_Len =  itsismx.DecToHex( #DUID / 2 )
 while #Option_Len < 4 do Option_Len = "0" .. Option_Len end

 stdnse.print_verbose(4, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   ".Solicit.Elapsed Time: " ..
   " \n\t --[[Option]]-Code: " ..   Option_Code ..
   " \n\t Option Lenght: " ..   Option_Len ..
   " \n\t DUID: " ..   DUID  )

 ClientID = Option_Code .. Option_Len .. DUID

 return ClientID, DUID , LinkAdd
end


local Generar_IA_Option = function()
  local Op_IAADDR, option_len, Ipv6Add , preferred, valid, options
  Op_IAADDR = "0005"

  preferred, valid = "00000000", "00000000"
  Ipv6Add =   "20010db8c0ca000000000000c0a8010b"
  options = ""
   option_len = itsismx.DecToHex( 24 + #options/2)
   while #option_len < 4 do option_len = "0" .. option_len end

 return Op_IAADDR .. option_len .. Ipv6Add .. preferred .. valid .. options
end

local Generar_Option_IA_TA = function()
 local IA_TA
 local Option_Code , Option_Len, IAID, Options = "0004", 4, 0, ""
 IAID = "f"
 while #IAID < 8 do IAID = "0" .. IAID  end
```

```
 Options = Generar_IA_Option()
 Option_Len = itsismx.DecToHex (4 + #Options/2)
 while #Option_Len < 4 do Option_Len = "0" .. Option_Len  end
 IA_TA = Option_Code .. Option_Len .. IAID .. Options

 stdnse.print_verbose(4, SCRIPT_NAME ..
    ".Solicit.Elapsed Time: " ..
    " \n\t Option-Code: " ..   Option_Len ..
    " \n\t Option Lenght: " ..   Option_Len ..
    " \n\t IAID: " ..   #IAID ..
    " \n\t Options: " ..   Options)
 return IA_TA , IAID
end

local Generar_Option_IA_NA = function()
    local IA_NA, Option_Code , Option_Len, IAID, T1, T2
    local IA_NA_Options
 Option_Code,T1, T2 = "0003", "00000000", "00000000"

 IAID="f" --Seem wide-dhcpv6-server need this field to be F
 while #IAID < 8 do IAID = "0" .. IAID  end

 IA_NA_Options = ""

 Option_Len = itsismx.DecToHex (12 + #IA_NA_Options/2)
 while #Option_Len < 4 do Option_Len = "0" .. Option_Len  end

 IA_NA = Option_Code .. Option_Len .. IAID .. T1 .. T2 .. IA_NA_Options
 return IA_NA , IAID
end
local Generar_Option_Elapsed_Time = function()

 local option_code, option_len, elapsed = "0008" , "0002", "0000"

 local TimetoBeg = stdnse.get_script_args( "itsismx-dhcpv6.TimeToBeg" )

 if TimetoBeg ~= nil  then
     elapsed = itsismx.DecToHex(TimetoBeg )
     while #elapsed < 4 do elapsed = "0" .. elapsed end
 end

 stdnse.print_verbose(4, SCRIPT_NAME ..
    ".Solicit.Elapsed Time: " ..
    " \n\t Option-Code: " ..   option_code ..
    " \n\t Option Lenght: " ..   option_len ..
    " \n\t Time elapsed: " ..   elapsed  )

 return option_code .. option_len .. elapsed
end



local Generar_Option_Request = function()

  local Option_Oro, Option_Len, req_option_code1 = "0006", "0002"
  req_option_code1 = "0018"
  return Option_Oro .. Option_Len .. req_option_code1

end
```

```
local Spoof_Host_Solicit = function ()

 local Solicit, Error = "01", nil
 local TransactionID, DUID, IAID, LinkAdd = 0
 local ClientID, IA_TA, Time, Option_Request
 local IA_NA
 local Host = { "DUID", "Type", "IAID", "LinkAdd"}

 local Na_or_Ta = stdnse.get_script_args( "itsismx-dhcpv6.IA_NA" )
 Na_or_Ta = true

 local Bool_Option_Req = stdnse.get_script_args(
 "itsismx-dhcpv6.Option_Request" )

 TransactionID = itsismx.DecToHex( math.random( 16777216 ) ) -- 2^24
 while #TransactionID < 6 do TransactionID = "0" .. TransactionID end

 ClientID, DUID, LinkAdd = Generar_Option_ClientID()

 if Na_or_Ta  == nil then
     IA_TA, IAID = Generar_Option_IA_TA ()
 else
     IA_NA, IAID = Generar_Option_IA_NA ()
 end

 Time = Generar_Option_Elapsed_Time()

 if Bool_Option_Req ~= nil then
     Option_Request = Generar_Option_Request()
 end

 stdnse.print_verbose(3, SCRIPT_NAME ..
 ".Solicit: " .. " New SOLICIT Message. ID: " .. TransactionID )
 stdnse.print_verbose(3, SCRIPT_NAME ..
    ".Solicit: " .. " Client ID: " ..   ClientID  )

 if Na_or_Ta == nil then
     stdnse.print_verbose(3, SCRIPT_NAME ..
    ".Solicit: " .. " IA-TA : " ..   IA_TA  )

 else
     stdnse.print_verbose(3, SCRIPT_NAME ..
    ".Solicit: " .. " IA-NA : " ..   IA_NA  )
 end

 stdnse.print_verbose(3, SCRIPT_NAME ..
    ".Solicit: " .. " Time: " ..   Time  )

 if Bool_Option_Req ~= nil then
     stdnse.print_verbose(3, SCRIPT_NAME ..
    ".Solicit: " .. " Option Request: " ..   Option_Request  )
 end

 Host.DUID = DUID
 Host.Type = "temporary" -- For this version we're  using only IA_TA
 Host.IAID = IAID
 Host.LinkAdd = LinkAdd

 stdnse.print_verbose(2, SCRIPT_NAME ..
 ".Solicit: " .. " (G)Host - Link-Address: " .. Host.LinkAdd .. --
```

```
    " type of request: " .. Host.Type  ..
    "\n DUID: " ..  Host.DUID .. "\n IAID: "  .. Host.IAID )


 if Na_or_Ta == nil then
 Solicit = Solicit .. TransactionID .. ClientID .. IA_TA .. Time
 else
 Solicit = Solicit .. TransactionID .. ClientID .. IA_NA .. Time
 end

 if Bool_Option_Req ~= nil then
     Solicit = Solicit .. Option_Request
 end

 return Solicit, Host,  Error
end

local Spoof_Relay_Forwarder = function ( Source, SOLICIT , Subnet )



 local msg_type, hopcount, linkAdd, peerAdd, Options
 local Relay, sError, sUnicast, Address, Prefix

 msg_type = "0C" -- msg-type is 12 ( 0x0C)
 hopcount = "00" -- Though this option could be other values some server don-t
 accept a fake number
 linkAdd = "20010db8c0ca00000000000000000001" -- THIS IS WHAT WE WANT !!!!
 peerAdd  = Source
 Options = Generar_Option_Relay( SOLICIT )

 if Subnet == nil then
   linkAdd = "20010db8c0ca00000000000000000001"
 elseif #Subnet == 0 then -- empty or nil is bad (But we need to confirm first is not
 nil)
   linkAdd = "20010db8c0ca00000000000000000001"
 else --We assume is IPv6 Address and we need to convert to Hexadecimal value

     Address, Prefix = itsismx.Extract_IPv6_Add_Prefix(Subnet)
 sUnicast, sError = ipOps.get_last_ip (Address, Prefix) --We use the last IPv6
 add because is alway valid

     if sUnicast == nil then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 "\n\t Relay Forward: " .. " The subnet provided (" ..
 stdnse.string_or_blank(Subnet) ..
      ")  was bad formed and throw the next error: " ..  sError )
  return "", sError
      else


  linkAdd = itsismx.Expand_Bytes_IPv6_Address(sUnicast)

     end
 end

 stdnse.print_verbose(3, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
    "\n\t Relay Forward: " .. " msg_type: " ..   msg_type ..
    "\n\t hopcount: " .. msg_type  ..
    "\n\t linkAdd: " .. linkAdd  ..
```

```
   "\n\t peerAdd: " .. peerAdd  ..
   "\n\t Options: " ..  Options   )

 Relay = msg_type .. hopcount .. linkAdd ..  peerAdd .. Options
 return Relay, nil
end


 local Verify_Relay_Reply = function ( PeerAddress, Relay_Reply , Subnet )

      local Longitud, Adv_Msg = #Relay_Reply-(49+38)+1 , ""
      local hex_pos, hex_dhcp_data, Campos
      local Candidata, bBool, sBool

 hex_pos , hex_dhcp_data = bin.unpack("H".. tostring(Longitud), Relay_Reply
 ,49+38 )

      if ( hex_dhcp_data:sub(1,2) ~= "02") then
 return false, "It's not a Solicit message"
      end
 local Na_or_Ta = stdnse.get_script_args( "itsismx-dhcpv6.IA_NA" )
   Na_or_Ta = true
      local offset=0
      if (Na_or_Ta ) then  -- IA-NA
     offset= 7+2+4
 Campos = hex_dhcp_data:sub(offset) -- We extract until "Option-len"

     Longitud = Campos:sub(1,4)

 stdnse.print_verbose(3, "Client ID Option length: " .. tonumber(Longitud,16)
 .. " bytes")

     offset= 7+2+4+4+ tonumber(Longitud,16)*2 + 4
 Campos = hex_dhcp_data:sub(offset) -- We extract until "Option-len"
     Longitud = Campos:sub(1,4)

 stdnse.print_verbose(3, "Server ID Option length: " .. tonumber(Longitud,16)
 .. " bytes")

      offset = offset + tonumber(Longitud,16)*2 + 4
      Campos = hex_dhcp_data:sub(offset)

     offset = offset + 4 + 4 + 4 + 8 + 8 +4
     Campos = hex_dhcp_data:sub(offset)

     if ( Campos:sub(1,4) ~= "0005") then
 return false, "We are waiting for a IA-NA but got another type of answer: " ..
 Campos:sub(1,4)
     end

     offset = offset + 4 + 4
     Campos = hex_dhcp_data:sub(offset)
     Candidata = Campos:sub(1,32)
  return true, Campos:sub(1,32)

      else
  stdnse.print_verbose(1, "IA-TA is not yet implemented.!")
      end
```

```
   bBool, sBool = ipOps.ip_in_range(Candidata, Subnet )
   if bBool then
   return bBool, nil
   else
   return bBool, sBool
   end

end
prerule = function()
  if ( not(nmap.is_privileged()) ) then
 stdnse.print_verbose("%s with lack of privileges (and we need GNU/Linux).",
 SCRIPT_NAME)
  return false
 end

  if ( not(nmap.address_family() == "inet6") ) then
 stdnse.print_verbose("%s Need to be executed for IPv6.", SCRIPT_NAME)
  return false
 end

   if ( nmap.get_interface () == nil ) then
 stdnse.print_verbose(" The NSE Script need to work with a Ethernet Interface,
 Please use the argument -e <Interface>. " )
      return false
   end

 local iface, err = nmap.get_interface_info( nmap.get_interface () )

    if ( err ~=nil) then
 stdnse.print_verbose (1, "dhcv6 can't be initialized due the next ERROR: " .. err
 )
 return false;
    elseif iface.link ~= "ethernet"  then
 stdnse.print_verbose (1, " The NSE Script need to work with a Ethernet Interface,
 Please use the argument -e <Interface> to select it. " )
 return false
    end

  return true
end

 local Transmision_Recepcion = function ( IPv6src, IPv6dst , Prtsrc, Prtdst ,
 Mensaje, Interface , Subnet)

 local Bytes --, ToTransmit
 Bytes = bin.pack("H" , "0000000000000000" .. Mensaje ) -- those extra bits are for
 being overwritten

 local Interfaz, err = nmap.get_interface_info( Interface )
 local Bool, Tcpdumpfilter

 local dnet = nmap.new_dnet()
 local pcap = nmap.new_socket()


 local src_mac = packet.mactobin("00:D0:BB:00:7d:01") -- (Spoofed) Cisco
 device!
 local dst_mac = packet.mactobin("33:33:00:01:00:02")

 local src_ip6 = bin.pack("H", itsismx.Expand_Bytes_IPv6_Address( IPv6src ) )
```

```
 local dst_ip6 = packet.ip6tobin(IPv6dst)


 dnet:ethernet_open( Interfaz.device)
 Tcpdumpfilter = "ip6 dst " .. IPv6src .. " and udp src port 547 and udp dst port 547"

 stdnse.print_verbose(5, "\t The DCPdump filter: \t" .. Tcpdumpfilter )
 pcap:pcap_open(Interfaz.device, 1500, true, Tcpdumpfilter)

 local Spoofed = packet.Packet:new()
 Spoofed:build_ipv6_packet(src_ip6,dst_ip6,17, Bytes ,3,0,0)
 Bool = Spoofed:ip6_parse(false)

 Spoofed:udp_parse(false) --Now the UDP ...

 Spoofed:udp_set_sport(Prtsrc)
 Spoofed:udp_set_dport(Prtdst)
 Spoofed:udp_set_length(#Bytes)
 Spoofed.ip_p = 17 -- Seem that udp_count_checksum() wasn't update for IPv6...
 Spoofed:udp_count_checksum()

 Spoofed:count_ipv6_pseudoheader_cksum()

 local probe = packet.Frame:new()
 probe.mac_dst = dst_mac
 probe.mac_src = src_mac
 probe.ether_type = string.char(0x86, 0xdd)
 probe.buf = Spoofed.buf

 dnet:ethernet_send(dst_mac .. src_mac .. string.char(0x86, 0xdd) .. Spoofed.buf)

 local status, plen, l2_data, l3_data, time , hex_pos , hex_l3_data
  status, plen, l2_data, l3_data, time = pcap:pcap_receive()

 if (status==nil) then -- No packet captured (Timeout)
 stdnse.print_verbose(3, " The subnet " .. Subnet .. " seem not be avaliable" )
   Bool = false
 elseif status==true then -- We got a packet ... time to work with it
  Bool , err = Verify_Relay_Reply(src_ip6, l3_data,  Subnet )

  if Bool == false then
 stdnse.print_verbose(1, " The subnet " .. Subnet .. " got this error: " .. err)
  else
 stdnse.print_verbose(3, " The subnet " .. Subnet .. " is Online")
  end

 end

 pcap:pcap_close()
 dnet:ip_close()
 dnet:ethernet_close()
 return Bool
end

local Extaer_Subredes = function(Subnet)

  local Auxiliar = {}
  local  Contador, Valor, Aux
 local Net, Bits, Total, Dirre, Prefijo, NewPrefix, Binario, NewNet, mensaje
```

```
    if type(Subnet) == "table" then  --  {X:X:X:X::/YY, B, T}  ??

     Net, Bits, Total = Subnet[1] , Subnet[2], Subnet[3]
     Dirre, Prefijo = itsismx.Extract_IPv6_Add_Prefix(Net)
     NewPrefix = Prefijo + Bits

     Binario, mensaje = ipOps.ip_to_bin(Dirre)

 if Binario ~= nil then -- We proceed to save the entry to the list.
  for Contador = 1, Total do
 Valor = stdnse.tobinary(Contador) -- There is a very low risk of overflow with
 this tactic...
    while #Valor < tonumber(Bits) do Valor = "0" .. Valor end

 NewNet = Binario:sub(1,Prefijo) .. Valor .. Binario:sub(NewPrefix+1 , 128)
    NewNet = ipOps.bin_to_ip(NewNet)
    table.insert(Auxiliar, NewNet .. "/" .. NewPrefix)
  end

     else    -- Error, so we escape this entry
 stdnse.print_verbose(3, SCRIPT_NAME .. "\t\t The next provided subnet has
 wrong syntax: " ..
   Subnet ..   mensaje)
     end

   else
        table.insert(Auxiliar, Subnet) -- X:X:X:X::/YY
   end

  return Auxiliar
end


local Listado_Subredes = function ()
    local TotalNets, Aux = {} , {}
 local Subredes = stdnse.get_script_args( "itsismx-dhcpv6.subnets" )


    local index, campo, Subnets
    local interface_name
    if Subredes ~= nil then
 if type(Subredes) ==  "table" then

     for index, campo in ipairs(Subredes) do
       Aux = Extaer_Subredes(campo)
 for _, Subnets in ipairs(Aux) do table.insert(TotalNets,Subnets ) end

     end
 else
     Aux = Extaer_Subredes(Subredes)
 for _, Subnets in ipairs(Aux) do table.insert(TotalNets,Subnets ) end

 end

 else -- We need provided at least one valid sub-net (Future works will

 stdnse.print_verbose(1, SCRIPT_NAME .. " ERROR: Need to provided at least one "
 ..
 " single subnet to test. Use the argument itsismx-dhcpv6.subnets " )
```

```
    end
    return TotalNets
end
action = function()

 local tOutput = stdnse.output_table()
 local bExito , bRecorrido = false, false
 local tSalida =  { Subnets={}, Error=""}
 local microseconds = stdnse.get_script_args( "itsismx-dhcpv6.utime" )
 local Boolean_IPv6Address = stdnse.get_script_args(
 "itsismx-dhcpv6.Spoofed_IPv6Address" )
 local Spoofed_IPv6Address

 if microseconds == nil then
     microseconds = 200
 else
     microseconds = tonumber( microseconds )
 end

 tOutput.Subnets = {}
 itsismx.Registro_Global_Inicializar("dhcpv6") -- We prepare our work!

 local Mensaje, Host, Error, Relay
 local UserSubnets, Index, Subnet

 UserSubnets = Listado_Subredes()

 for Index, Subnet in ipairs(UserSubnets) do
     math.randomseed ( nmap.clock_ms() )
 Mensaje, Host, Error = Spoof_Host_Solicit() -- Each subnet a different host
 Relay = Spoof_Relay_Forwarder ( Host["LinkAdd"] , Mensaje , Subnet )

     if  Boolean_IPv6Address == nil  then
   local iface, err = nmap.get_interface_info(nmap.get_interface ())

   if ( err ~=nil  ) then
    tSalida.Error = err
   else
    Spoofed_IPv6Address = iface.address
   end

     else
    Spoofed_IPv6Address = Host.LinkAdd
     end

 bRecorrido = Transmision_Recepcion( Spoofed_IPv6Address , "FF02::1:2",
 546,547, Relay, nmap.get_interface () , Subnet )

    bExito = bExito or bRecorrido
    if bRecorrido then
   table.insert(tSalida.Subnets,Subnet)
    end

    itsismx.waitUtime( math.random(microseconds) )

 end

 if (bExito) then
  nmap.registry.itsismx.PrefixesKnown = tSalida.Subnets
 stdnse.print_verbose(1, SCRIPT_NAME .. " Were added " .. #tSalida.Subnets ..
```

```
      " subnets to scan!"   )
 else
 itsismx.Registro_Global_Inicializar("PrefixesKnown") -- We prepare our
 work!
  nmap.registry.itsismx.PrefixesKnown = tSalida.Subnets
 stdnse.print_verbose(1, SCRIPT_NAME .. " Not sub-net were added to the scan
 list!" )
 end
 return stdnse.format_output(bExito, tOutput);

end
```

C.2.4. itsismx-words

```
local ipOps = require "ipOps"
local nmap = require "nmap"
local stdnse = require "stdnse"
local target = require "target"
local itsismx = require "itsismx"
local datafiles = require "datafiles"
local bin = require "bin"
local table = require "table"
local math = require "math"


description=[[
  This is the most simple and easier of all the scripts.
  The objective is to do a discover based on dictionary.

  For each prefix we discover, we are going to check
  against known hex-words.  ( EX. 2001:db8:c0ca::beef )

  P.d. The dictionary still need more entries for this
  script become very useful.
]]

---
-- @usage
-- nmap -6 --script itsismx-slaac --script-args newtargets
--
-- @output
-- Pre-scan script results:
-- | itsismx-wordis:
-- |_  itsismx-wordis.prerule:  Were added 4 nodes to the host scan phase

-- Host script results:
-- | itsismx-wordis:
-- | Host online - IPv6 address SLAAC
-- |_  2001:db8:c0ca::dead

-- @args newtargets      MANDATORY
-- @args itsismx-wordis.nsegments (Optional)
-- @args itsismx-wordis.fillright (Optional)

-- @args itsismx-subnet  (Optional)

-- Version 1.0
--  Created 29/04/2013 - v0.1 - created by Ing. Raul Fuentes <ra.fuentess.sam@gmail.com>
--
```

```
author = "Raul Fuentes"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"broadcast", "safe"}

dependencies = {"itsismx-dhcpv6"}

 local CrearRangoHosts = function (Direccion, Prefijo, TablaPalabras,
 User_Segs, User_Right )

 local IPv6Bin, Error   = ipOps.ip_to_bin (Direccion )
 local Candidatos, sError = {} , ""
 local Indice, Palabras, MaxRangoSegmentos, Filler, Host
 if IPv6Bin == nil then --Niagaras!
  return false, {}, Error
 end

 if (User_Segs == nil  ) then
  MaxRangoSegmentos  = math.ceil( (128 - Prefijo)/16 )
  User_Segs = false
 else
  MaxRangoSegmentos  = tonumber(User_Segs)
 end

 stdnse.print_verbose(3, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": Will be calculted " .. #TablaPalabras .. " hosts for the subnet: " .. Direccion ..
 "/" .. Prefijo )

 for  Indice , Palabras in ipairs(TablaPalabras ) do

 if ((tonumber( Palabras.Segmento) <= MaxRangoSegmentos ) and User_Segs ==
 false ) or
 (User_Segs and (tonumber( Palabras.Segmento) == MaxRangoSegmentos ) )then
   Filler = ""
   while (Prefijo + #Filler + #Palabras.Binario  ) < 128 do
    Filler = "0" ..Filler
   end

   if (User_Right ~= nil ) then
    Host = IPv6Bin:sub(1, Prefijo)  .. Palabras.Binario .. Filler
   else
    Host = IPv6Bin:sub(1, Prefijo) .. Filler .. Palabras.Binario
   end

   Host, Error = ipOps.bin_to_ip(Host)
   if Host == nil then -- Something is very wrong but we don-t stop
    sError = sError .. "\n" .. Error
   else
    table.insert(Candidatos , Host )
   end
  end
 end
 return true, Candidatos, sError
end
local LeerArchivo = function ( RangoSegmentos )
 local bBoolean, Archivo =
 datafiles.parse_file("nselib/itsismx-words-known",{ "^[^#]+%d" })
 local index, reg, token
 local Candidatos = {}
 local Registro  = { ["Segmento"]=0, ["Binario"]="0"}
```

```
 local sMatch = {}

 if bBoolean ~= true  then
  return nil , Archivo
 end

 for index, reg in pairs(Archivo) do
  sMatch = {}
  Registro  = { ["Segmento"]=0, ["Binario"]="0"}
  for token in  reg:gmatch("%w+" ) do
   sMatch[#sMatch+1] = token
  end
  Registro.Segmento = sMatch[1]
  Registro.Binario = sMatch[3]
  table.insert( Candidatos, Registro )
 end
 return Candidatos, ""
end

local Prescanning = function ()
 local bSalida, tSalida = false , { Nodos={}, Error=""}
 local IPv6PRefijoUsuario = stdnse.get_script_args("itsismx-subnet")
 local IPv6PRefijoScripts  = nmap.registry.itsismx.PrefixesKnown
 local  TablaPalabras, sError, IPv6refijosTotales  = {}, "",{}
 local PrefixAux, Prefijo,  Direccion
 local Hosts, Nodo, Indice = {}
 local User_Segs, User_Right =
 stdnse.get_script_args("itsismx-wordis.nsegments","itsismx-wordis.fillright" )
 stdnse.print_verbose(2, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   ": Begining the Pre-scanning work... "    )

  TablaPalabras = LeerArchivo()
 if TablaPalabras == nil then
  tSalida.Error = sError
  return bSalida, tSalidas
 end

 if IPv6PRefijoUsuario == nil and IPv6PRefijoScripts == nil then
 tSalida.Error = "There is not IPv6 subnets to try to scan!. You can run a script for
 discovering or adding your own" ..
       "  with the arg: itsismx-subnet."
  return bSalida, tSalida
 end

 if IPv6PRefijoScripts ~= nil then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": Number of Prefixes Known from other sources: " .. #IPv6PRefijoScripts )
  for _ , PrefixAux in ipairs(IPv6PRefijoScripts) do
   table.insert(IPv6refijosTotales,PrefixAux )
  end
 end

 if IPv6PRefijoUsuario ~= nil then
  if type(IPv6PRefijoUsuario) ==  "string" then
   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
    ": Number of Prefixes Known from other sources: 1 "     )
   table.insert(IPv6refijosTotales,IPv6PRefijoUsuario )
  elseif type(IPv6PRefijoUsuario) ==  "table" then
   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": Number of Prefixes Known from other sources: " .. #IPv6PRefijoUsuario )
```

```
 for _ , PrefixAux in ipairs(IPv6PRefijoUsuario) do -- This is healthy for my
 mind...
    table.insert(IPv6refijosTotales,PrefixAux )
   end
  end
 end

 for _, PrefixAux in ipairs(IPv6refijosTotales) do
   Direccion, Prefijo = itsismx.Extract_IPv6_Add_Prefix(PrefixAux)
 bSalida, Hosts, sError = CrearRangoHosts (Direccion, Prefijo, TablaPalabras,
 User_Segs, User_Right )

   if bSalida ~= true then
   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": There was a error for the prefix: " .. PrefixAux .. " Message:" .. sError )
   end

 if sError ~= "" then -- Not all the error are fatal for the script.
   tSalida.Error = tSalida.Error .. "\n" .. sError
   end

   for Indice, Nodo in ipairs(Hosts) do
   table.insert( tSalida.Nodos, Nodo)
   end
 end
 return true, tSalida
end

local Hostscanning = function( host)
 local tSalida = { Nodos=nil, Error=""}
 local aux
 stdnse.print_verbose(4, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   ": Begining the Host-scanning results... " )
 if nmap.registry.itsismx == nil then
 tSalida.Error = "You must first initialize the global register Itsismx (There is
 a global function for that!)"
  return false, tSalida
 end
 aux = nmap.registry.itsismx.wordis
 if aux == nil then
 tSalida.Error = "The global register Itsismx wasn't initialzed correctly
 (There is a global function for that!)"
  return false, tSalida
 end
 aux[#aux +1] = host.ip
 nmap.registry.itsismx.wordis = aux

 tSalida.Nodos = host.ip  -- This rule ALWAY IS ONE ELEMENT!
 return true, tSalida
end

prerule = function()
   if ( not(nmap.address_family() == "inet6") ) then
 stdnse.print_verbose("%s Need to be executed for IPv6.", SCRIPT_NAME)
  return false
 end

 if ( stdnse.get_script_args('newtargets')==nil ) then
 stdnse.print_verbose(1, "%s Will only work on pre-scanning. The argument
 newtargets is needed for the host-scanning to work.", SCRIPT_NAME)
```

```
 end
 return true
end

hostrule = function(host)
 local Totales, Objetivo, bMatch, sMatch = nmap.registry.wordis_PreHost

 if Totales == nil  then return false end
 for _, Objetivo in pairs( Totales ) do
  bMatch, sMatch = ipOps.compare_ip(host.ip, "eq", Objetivo)
  if bMatch == nil then
   stdnse.print_verbose(1, "\t hostrule  had a error with " ..
        host.ip .. "\n Error:" .. sMatch )
  elseif bMatch then
   return true
  end
 end

 return false
end

action = function(host)

 local tOutput = {}
 tOutput = stdnse.output_table()
 local bExito = false
 local tSalida =  { Nodos={}, Error=""}
 local  bHostsPre, sHostsPre
 local Nodes = {} -- Is a Auxiliar

 itsismx.Registro_Global_Inicializar("wordis") -- Prepare everything!

 if ( SCRIPT_TYPE== "prerule" ) then

  bExito , tSalida = Prescanning()

  tOutput.warning = tSalida.Error

  if bExito then
   for _,  sHostsPre in ipairs(tSalida.Nodos) do
    bHostsPre, sTarget = target.add(sHostsPre)
    if bHostsPre then --We add it!
     table.insert(Nodes, sHostsPre)
    else -- Bad luck
     tOutput.warning = tOutput.warning .. " \n" .. sTarget
    end
   end

   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " Temptative address based on SLAAC added to the scan:" .. #tSalida.Nodos ..
 "\n Succesful address based on SLAAC added to the scan:" .. #Nodes )

   nmap.registry.wordis_PreHost = Nodes
 table.insert(tOutput, SCRIPT_NAME .. "." .. SCRIPT_TYPE .. ": Were added " ..
 #Nodes ..
       " nodes to the host scan phase" )

 end
 end
```

```
 if SCRIPT_TYPE== "hostrule" then
  bExito , tSalida = Hostscanning(host)
   tOutput.warning = tSalida.Error

   if ( bExito ~= true) then
   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
        " Error: " .. tSalida.Error)
   end

   tOutput.name = "Host online - IPv6 address SLAAC"
 table.insert(tOutput,tSalida.Nodos) --This will be alway one single host.
 end

 return stdnse.format_output(bExito, tOutput);

end
```

C.2.5. itsismx-map4to6.nse

```
local ipOps = require "ipOps"
local nmap = require "nmap"
local stdnse = require "stdnse"
local table = require "table"
local target = require "target"
local itsismx = require "itsismx"
local tab = require "tab"

description=[[
  This script SHOULD run a typical Nmap discovery of hosts  on IPv4   but for each host up will
  try to map the address to IPv6 (IPv6 subnets provided by user or by other script ).

  HOWEVER, There is a problem with nmap 6.25 architecture:  OR only  IPv4   OR only  IPv6
  by execution. This mean we can't check first for IPv4 address to be up so, the
  user must provided the IPv4 hosts   to check. We have two way to do it: The User provide
  IPv4 hosts address   or provide IPv4 Subnet Address ( X.X.X.X/YY ).

  The script run at pre-scanning phase and script phase (The first for create tentative
  4to6 address and the second for put the living nodes on the list of discovered nodes).

]]

---
-- @usage
-- nmap -6 --script itsismx-Map4to6 --script-args newtargets,itsismx-Map4t6.IPv4Hosts=X.X.X.X
--
-- @output
-- Pre-scan script results:
-- | itsismx-map4to6:
-- |_  itsismx-map4to6.prerule:  Were added 18 nodes to the host scan phase

-- Host script results:
-- | itsismx-map4to6:
-- | Host online - Mapped IPv4 to IPv6
-- |_  2001:db8:c0ca:1::9d9f:64e1

-- nmap.registry.itsismx.Map4t6
-- nmap.registry.map6t4_PreHost
-- @args itsismx-Map4t6.IPv4Hosts
-- @args itsismx-SaveMemory (Optional)
```

```
-- @args itsismx-subnet
-- @args newtargets  MANDATORY

--
-- Version 1.0
-- Update 18/10/2013 - V 1.0.1 - Added  SaveMemory option
-- Update  29/03/2013 - V 1.0 -  Functional script
--  Created 28/03/2013 - v0.1  - created by Ing. Raul Fuentes <ra.fuentess.sam+nmap@gmail.com>
--

author = "Raul Armando Fuentes Samaniego"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"broadcast", "safe"}

dependencies = {"itsismx-dhcpv6"}

 local From_4_to_6 = function (IPv6_Network, IPv6_Prefix, IPv4SHosts )
 local sError, Listado = nil, 0

 local _ , Host -- _ Can give problem
 local sBin6, sBin4, tTabla = nil, nil, {}
 local SaveMemory, bool ,err = stdnse.get_script_args( "itsismx-SaveMemory"
 )

 if IPv6_Prefix > 96 then
 return nil , " The IPv6 subnet: " .. IPv6_Network .. "/" .. IPv6_Prefix ..
      " can't support a direct Mapping 4 to 6."
 end
 sBin6,sError = ipOps.ip_to_bin(IPv6_Network) -- We don't left dangerous
 operation
 if sBin6 == nil then
  return nil, sError
 end

 if type(IPv4SHosts) ==  "table" then
  tTabla = IPv4SHosts
 else
  table.insert(tTabla, IPv4SHosts)
 end

 stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ".Map4to6: " .. " Total IPv4 objects to analyze: " .. #tTabla ..
    " for IPv6 subnet: " ..IPv6_Network .. "/" .. IPv6_Prefix   )

 for _ ,  Host in ipairs(tTabla) do
   stdnse.print_verbose(2, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
      ".Map4to6: " .. " IPv4 Object: " ..   Host  )

   sBin4, sError = ipOps.ip_to_bin(Host) -- We have two options ..
   if ( sBin4 == nil ) then

    local  IPv4_First, IPv4_Last, IPv4_Next, SErr2, IPAux
    IPv4_First, IPv4_Last, SErr2 = ipOps.get_ips_from_range( Host )

 if (IPv4_First ~= nil and IPv4_Last ~= nil ) then -- Sweep Subnet, we must do

     IPv4_Next = itsismx.GetNext_AddressIPv4(IPv4_First)
     while  ipOps.ip_in_range( IPv4_Next ,Host)  do
      if ipOps.compare_ip (IPv4_Next, "lt", IPv4_Last ) then
```

```
       IPAux = sBin6:sub(1,96) ..   ipOps.ip_to_bin(IPv4_Next)
       IPAux = ipOps.bin_to_ip(IPAux)
 stdnse.print_verbose(5, SCRIPT_NAME .. "." .. SCRIPT_TYPE .. ".Map4to6: " .. "
 \t IPv6 address: " .. IPAux )


       bool ,err =  target.add(IPAux )
       if ( (bool) and not(SaveMemory) ) then
        table.insert( nmap.registry.map6t4_PreHost, IPAux )

       elseif not(bool) then
 stdnse.print_verbose(6, SCRIPT_NAME .. "." .. SCRIPT_TYPE .. ": Had been a error
 adding the node " .. IPAux .. " which is: " .. err )
       end
      end
      IPv4_Next = itsismx.GetNext_AddressIPv4(IPv4_Next)
     end

    else -- This entry of host IS WRONG! WRONG!
 return #nmap.registry.map6t4_PreHost, "At least one Host/Subnet was wrong
 passed: " .. Host
    end

   else  -- Format: X.X.X.X
    Host =  sBin6:sub(1,96) ..  sBin4
    Host = ipOps.bin_to_ip(Host)
    stdnse.print_verbose(5, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
         ".Map4to6: " .. " \t IPv6 address: " ..   Host  )


    bool ,err =  target.add(Host )
    if ( (bool) and not(SaveMemory) ) then
     table.insert( nmap.registry.map6t4_PreHost, Host )
    elseif not(bool) then
 stdnse.print_verbose(6, SCRIPT_NAME .. "." .. SCRIPT_TYPE .. ": Had been a error
 adding the node " .. Host .. " which is: " .. err )
    end

   end
 end

 return #nmap.registry.map6t4_PreHost
end

local Prescanning = function ()

 local bSalida = false
 local tSalida = { Nodos=0, Error=""}
 local IPv6_Subnet , IPv6_Add, IPv6_Prefix
 local IPv6Host, sError, Grantotal = nil, nil, 0
 local IPv4Subnets, IPv6User =
 stdnse.get_script_args("itsismx-Map4t6.IPv4Hosts", "itsismx-subnet" )
 local IPv6Knowns = nmap.registry.itsismx.PrefixesKnown
 local iIndex


 stdnse.print_verbose(2, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   ": Begining the Pre-scanning work... "    )
```

```
if  IPv4Subnets == nil then
tSalida["Error"] = "There is not IPv4 subnets to scan!. You must provide it using
the argument: itsismx.Map4t6.IPv4Nets "
 return bSalida, tSalida
end

if IPv6User == nil and IPv6Knowns == nil then
tSalida["Error"] = "There is not IPv6 subnets to try to scan!. You can run a script
for discovering or adding your own" ..
      "  with the arg: itsismx.PrefixesKnown."
 return bSalida, tSalida
end

if IPv6Knowns ~= nil then

 stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" Number of Subnets Known from other sources: " .. #IPv6Knowns )

for _ , IPv6_Subnet in ipairs(IPv6Knowns) do --We need to extract the data
IPv6_Add, IPv6_Prefix = itsismx.Extract_IPv6_Add_Prefix(IPv6_Subnet) -- We
break the data

IPv6Host, sError = From_4_to_6(IPv6_Add, IPv6_Prefix,IPv4Subnets )
  if ( sError ~= nil) then
stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE .. " ERROR: One IPv6
subnet wasnt translate")
   tSalida["Error"] = tSalida["Error"] .. "\n" ..  sError
  end
  if IPv6Host then    -- We need to concatenate the new nodes
   Grantotal = Grantotal + IPv6Host
  end
 end
end

if IPv6User ~= nil then  -- We got tww options with this.
 if   type(IPv6User) ==  "string" then

  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   " Number of Subnets  provided by the user:  1"     )

IPv6_Add, IPv6_Prefix = itsismx.Extract_IPv6_Add_Prefix(IPv6User)
IPv6Host, sError = From_4_to_6(IPv6_Add, IPv6_Prefix,IPv4Subnets )

  if ( sError ~= nil) then
stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE .. " ERROR: One IPv6
subnet wasnt translate")
   tSalida["Error"] = tSalida["Error"] .. "\n" ..  sError
  end
  if IPv6Host then    -- We need to concatenate the new nodes
   Grantotal = Grantotal + IPv6Host
  end
 elseif type(IPv6User) ==  "table" then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   " Number of Subnets provided by the user: " .. #IPv6User    )

for _ , IPv6_Subnet in ipairs(IPv6User) do --We need to extract the data
IPv6_Add, IPv6_Prefix = itsismx.Extract_IPv6_Add_Prefix(IPv6_Subnet) -- We
break the data
IPv6Host, sError = From_4_to_6(IPv6_Add, IPv6_Prefix,IPv4Subnets )
   if ( sError ~= nil) then
```

```
 stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE .. " ERROR: One IPv6
 subnet wasnt translate")
     tSalida["Error"] = tSalida["Error"] .. "\n" ..  sError
    end

    if IPv6Host then    -- We need to concatenate the new nodes
      Grantotal = Grantotal + IPv6Host
    end
   end
  else
 stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE .. " WARNING: The
 itsismx-subnet was ignored because wrong values")
  end


 end

 tSalida.Nodos = Grantotal
 return true, tSalida -- We got to this point everything was fine (Or don't crash)!
end

local Hostscanning = function (host)

 local tSalida = { Nodos=nil, Error=""}
 local aux

 stdnse.print_verbose(3, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   ": Begining the Host-scanning results... "    )

 if nmap.registry.itsismx == nil then
 tSalida.Error = "You must first initialize the global register Itsismx (There is
 a global function for that)"
  return false, tSalida
 end
 aux = nmap.registry.itsismx.Map4t6
 if aux == nil then
 tSalida.Error = "The global register Itsismx wasn't initialzed correctly
 (There is a global function for that)"
  return false, tSalida
 end

 aux[#aux +1] = host.ip
 nmap.registry.itsismx.Map4t6 = aux

 tSalida.Nodos = host.ip

 return true, tSalida

end

prerule = function()
 if ( not(nmap.address_family() == "inet6") ) then
 stdnse.print_verbose("%s Need to be executed for IPv6.", SCRIPT_NAME)
  return false
 end

 if ( stdnse.get_script_args('newtargets')==nil ) then
 stdnse.print_verbose(1, "%s Will only work on pre-scanning. The argument
 newtargets is needed for the host-scanning to work.", SCRIPT_NAME)
 end
```

```
 return true
end
hostrule = function(host)

 local Totales, Objetivo, bMatch, sMatch = nmap.registry.map6t4_PreHost

 if Totales == nil  then return false end

 for _, Objetivo in pairs( Totales ) do
  bMatch, sMatch = ipOps.compare_ip(host.ip, "eq", Objetivo)
  if bMatch == nil then
   stdnse.print_verbose(1, "\t hostrule  had a error with " ..
        host.ip .. "\n Error:" .. sMatch )
  elseif bMatch then
   return true
  end
 end
 return false
end


action = function(host)
 local tOutput = stdnse.output_table()
 local bExito = false
 local tSalida =  { Nodos={}, Error=""}
 local  sHostsPre, bTarget, sTarget
 local Nodes = {} -- Is a Auxiliar
 tOutput.Nodes = {}
 itsismx.Registro_Global_Inicializar("Map4t6") -- We prepare our work!

 if ( SCRIPT_TYPE== "prerule" ) then
  nmap.registry.map6t4_PreHost = {}
  bExito , tSalida = Prescanning()
  tOutput.warning = tSalida.Error

  if bExito then

   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": Succesful Mapped IPv4 to IPv6 added to the scan:" .. tSalida.Nodos )

   if tSalida.Error ~= "" then
       stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
        ".Warnings:  " ..  tSalida.Error )
   end

  else
   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": Was unable to add nodes to the scan list due this error: " .. tSalida.Error )

  end

 end

 if ( SCRIPT_TYPE== "hostrule" ) then

   bExito , tSalida = Hostscanning(host)
   tOutput.warning = tSalida.Error

   if ( bExito ~= true) then
```

```
   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
        " Error: " .. tSalida.Error)
   end

   tOutput.name = "Host online - Mapped IPv4 to IPv6"
  table.insert(tOutput,tSalida.Nodos)

 end

 return stdnse.format_output(bExito, tOutput);

end
```

C.2.6. itsismx-slaac.nse

```
local ipOps = require "ipOps"
local nmap = require "nmap"
local stdnse = require "stdnse"
local target = require "target"
local itsismx = require "itsismx"
local datafiles = require "datafiles"
local bin = require "bin"
local table = require "table"
local math = require "math"

description=[[

  This script is very simple, will run a brute-force attack for discovering
  all the posible hosts  using a stateless SLAAC configuration.  Remember
  this mean the first  64 bits are for the subnet then   next 64 bits
  are like this <High Mac> FF FE <Low Mac>

  By default will search 4,096 random hosts for one particular MAC vendor
  but will have arguments for run a full scan of  16,777,216.00  host
  by each vendor provided ( Only for the INSANE! ).

  BE CAREFUL , remember some vendors have more than one single OUI
]]

---
-- @usage
-- nmap -6 --script itsismx-slaac --script-args newtargets
--
-- @output

--
-- @args newtargets  MANDATORY Need for the host-scaning to succes
-- @args itsismx-slaac.vendors
-- @args itsismx-slaac.vms  (Optional)
-- @args itsismx-slaac.nbits  (Optional)
-- @args itsismx-slaac.vms-nbits (Optional)
-- @args itsismx-slaac.compute  (Optional)
-- @args itsismx-slaac.vmipv4  (Optional)
-- @args itsismx-slaac.knownbits   (Optional)
-- @args itsismx-SaveMemory

-- @args itsismx-subnet
-- @args itsismx-IPv6ExMechanism
```

```
-- Version 2.5
-- Updated 01/11/2013 - v2.5 - Fixed bugs and errors, the VM had a new argument.
-- Updated 11/10/2013 - v2.1  - A strong change for a more friendly use of memory.
-- Updated 25/04/2013 - v1.3 - First version at full power! (and minor corrections)
-- Updated 24/04/2013 - v1.0
--  Created 10/04/2013 - v0.1 - created by Ing. Raul Fuentes <ra.fuentess.sam+nmap@gmail.com>
--

author = "Raul Fuentes"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"broadcast", "safe"}

dependencies = {"itsismx-dhcpv6"}

local Brute_Range = function( IPv6Base, nBits )
 local TheLast, TheNext, err
 local  Prefix  = 0
 local IPv6ExMechanism = stdnse.get_script_args( "itsismx-IPv6ExMechanism"
 )
 local SaveMemory = stdnse.get_script_args( "itsismx-SaveMemory" )

 if nBits == nil then
  Prefix =  128 -11
 elseif tonumber(nBits) > 2 and  tonumber(nBits) <= 24 then
  Prefix =  128 - nBits
 else       -- Something wrong, we must be careful
  return nil
 end

 TheNext, TheLast, err = ipOps.get_ips_from_range(IPv6Base .. "00:0/" ..
 Prefix)

 stdnse.print_verbose(3, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
     ".Vendors.Address: " .. " will be calculated 2 ^ " ..
     nBits  .. " hosts using brute values for: " .. IPv6Base)
 repeat
  stdnse.print_verbose(5, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ".Brute-Mechanis: Added IPv6 address " .. TheNext .. " to the host scanning list...")

  bool ,err =  target.add(TheNext )
  if ( (bool) and not(SaveMemory) ) then
   table.insert( nmap.registry.slaac_PreHost, TheNext )
  elseif not(bool) then
    stdnse.print_verbose(6, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ".Brute-mechanism: Had been a error adding the node " .. TheNext .. " which is: " ..err )
  end

 TheNext = itsismx.GetNext_AddressIPv6(TheNext,Prefix, IPv6ExMechanism)

 until not ipOps.ip_in_range(TheNext, IPv6Base .. "00:0/" .. Prefix)
 return #nmap.registry.slaac_PreHost
end


local Random_Range = function ( IPv6Base, nBits )
 local  MaxNodos =  10
 local bool ,err
 local SaveMemory = stdnse.get_script_args( "itsismx-SaveMemory" )

 if nBits == nil then
```

```
 nBits = 11
 MaxNodos =  math.pow(2, nBits)
elseif tonumber(nBits) > 1 and  tonumber(nBits) <= 24 then
 MaxNodos =  math.pow(2, nBits)
else       -- Something wrong, we must be careful
 return nil
end

local iAux, iIndex, _, iValor, bUnico, hHost
local  Hosts, Numeros = {}, {}

stdnse.print_verbose(3, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
    ".Vendors.Address: " .. " will be calculated 2 ^ " ..
    nBits  .. " hosts using random values for: " .. IPv6Base)

for iIndex = 1, MaxNodos do
iAux = math.random( 16777216 ) --Remember this a C/C++ Random, isn-t better than
that!

 bUnico = true
 for _ , iValor  in ipairs(Numeros ) do
  if iValor == iAux then
   bUnico= false
   break
  end
 end

 if bUnico ~= true then
  iIndex = iIndex - 1 -- We don0t leave until got a new  Value...
 else

  table.insert(Numeros, iAux)
  hHost = itsismx.DecToHex(iAux)

  while #hHost < 6 do
   hHost = "0" .. hHost
  end
  hHost = hHost:sub(1,2) .. ":" .. hHost:sub(3,6)
  stdnse.print_verbose(5, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
".Random-mechanism: Adding IPv6 address " .. IPv6Base .. hHost .. " to the host
scanning list..." )

  bool ,err =  target.add(IPv6Base .. hHost )
  if ( (bool) and not(SaveMemory) ) then
   table.insert( nmap.registry.slaac_PreHost, IPv6Base .. hHost )
  elseif not(bool) then
   stdnse.print_verbose(6, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
".Random-mechanism: Had been a error adding the node " .. IPv6Base .. hHost .. "
which is: " .. err )
  end
 end
end
return #nmap.registry.slaac_PreHost
end

local Vmware_Range_000C29WellKnown = function( IPv6Base, sHexadecimal ,
IPv4Candidatos, IPv6ExMechanism )

local  _, Candidato, sIPv4L, Segmentos,sError
local sIPv4Ldot3, sIPv4Ldot4, sIPv6P120
```

```
local IPv6Prefix, TheNext, TheLast
local hosts = {}
local bool ,err
local SaveMemory = stdnse.get_script_args( "itsismx-SaveMemory" )
for _, Candidato in ipairs(IPv4Candidatos) do

 Segmentos,sError = ipOps.get_parts_as_number(Candidato)
 if ( sError ~= nil) then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" VMware(dynamic): ERROR with one or more IPv4 provided by the user:" ..
     Posible .. ". Error Message: " .. sError)
  return nil
 end

sIPv4Ldot3, sIPv4Ldot4 = itsismx.DecToHex(Segmentos[3]) ,
itsismx.DecToHex(Segmentos[4])

 while #sIPv4Ldot3 < 2 do sIPv4Ldot3 = "0" .. sIPv4Ldot3 end
 while #sIPv4Ldot4 < 2 do sIPv4Ldot4 = "0" .. sIPv4Ldot4 end
 sIPv4L  = sIPv4Ldot3 .. sIPv4Ldot4

 Segmentos,sError = ipOps.get_parts_as_number(IPv6Base)
 if ( sError ~= nil) then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" VMware(dynamic): ERROR with" .. IPv6Base .. "/64. Error Message: " .. sError)
  return nil
 end

IPv6Prefix = itsismx.DecToHex(Segmentos[1]) .. ":" ..
itsismx.DecToHex(Segmentos[2]) .. ":" ..
itsismx.DecToHex(Segmentos[3]) .. ":" .. itsismx.DecToHex(Segmentos[4]) ..
":" ..
sHexadecimal .. "FF:FE" .. sIPv4L:sub(1,2) .. ":" .. sIPv4L:sub(3,4) .. "00"

TheNext, TheLast, sError = ipOps.get_ips_from_range(IPv6Prefix .. "/120")
 if ( sError ~= nil) then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
     " VMware(dynamic): ERROR with (Variable IPv6Prefix)" ..
     IPv6Prefix .. "/120. Error Message: " .. sError)
  return nil
 end

 stdnse.print_verbose(4, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" VMware(dynamic): Will be add 256 targets to the scan list: " .. IPv6Prefix ..
"/120")

 repeat
  bool ,err =  target.add(TheNext )
  if ( (bool) and not(SaveMemory) ) then
   table.insert( nmap.registry.slaac_PreHost, TheNext )
  elseif not(bool) then
    stdnse.print_verbose(6, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" VMware(dynamic with well-known IP):: Had been a error adding the node " ..
TheNext .. " which is: " .. err )
  end

TheNext = itsismx.GetNext_AddressIPv6(TheNext,120, IPv6ExMechanism)
 until not ipOps.ip_in_range(TheNext, IPv6Prefix .. "/120" )
end
return  #nmap.registry.slaac_PreHost
```

```
end

local Vmware_Range_000C29 = function ( IPv6Base, nBits, Metodo )
 local hosts, sError = 0 , nil
 local IPv4Candidatos, Num_Aleatorios,iC, iAux = {},{},0
 local IPv4Argumentos, BitsKnown =
 stdnse.get_script_args("itsismx-slaac.vmipv4","itsismx-slaac.knownbits")
 local IPv6ExMechanism = stdnse.get_script_args( "itsismx-IPv6ExMechanism" )
 local TotalMuestras, Wellknown, RangoAleatorio
 local Segmentos,sError, Candidato,IPv6Prefix,IPv6Candidato
 local bool ,err
 local SaveMemory = stdnse.get_script_args( "itsismx-SaveMemory" )

 local sHexadecimal = "020C:29" -- This is the high 24 bits
 if IPv4Argumentos ~= nil then
  if type(IPv4Argumentos) ==  "string" then
   stdnse.print_verbose(2, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
    " VMware (Dynamic): The user provided  1  IPv4 address."     )
   table.insert(IPv4Candidatos,IPv4Argumentos )
  elseif type(IPv4Argumentos) ==  "table" then
   stdnse.print_verbose(2, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " VMware (Dynamic): The user provided " .. #IPv4Argumentos .. " IPv4 address." )
 for _ , PrefixAux in ipairs(IPv4Argumentos) do -- This is healthy for my mind...
    table.insert(IPv4Candidatos,PrefixAux )
   end
  end  -- To this point the first way don-t need to do more...

 return Vmware_Range_000C29WellKnown ( IPv6Base, sHexadecimal ,
 IPv4Candidatos, IPv6ExMechanism )
 end

 local nBits = stdnse.get_script_args("itsismx-slaac.vms-nbits")
 if nBits == nil then
  nBits = 2
 elseif tonumber(nBits) > 16 then -- As this is a special case, this can happens.
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " VMware (Dynamic): The args nbits was trunked to 16 for compute this part. " )
  nBits = 16
 elseif tonumber(nBits) < 1 then
  nbits = 1
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " VMware (Dynamic): The args nbits was set to 1 for compute this part. " )
 end

 if BitsKnown == nil then
  Wellknown = 0
 elseif itsismx.Is_Binary (BitsKnown) then
  Wellknown = #BitsKnown
 else -- The user provided something very important, wrong, SO WE STOP.
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " VMware (Dynamic): The user provided a wrong binary value: " .. BitsKnown )
  return nil
 end


 if  nBits + Wellknown >  16 then --  Houston we have a problem...
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " VMware (Dynamic): There is incongruity with knownbits and/or nbits" ..
 " because are bigger than 16: " .. nBits .. " + " .. Wellknown .. " > 16" )
  return nil
```

```
end

TotalMuestras = math.pow(2, nBits)    -- How many samples to do
RangoAleatorio = math.pow(2, 16 - Wellknown) -- How big the random number to
compute


if Metodo == nil then
 if RangoAleatorio <=  math.floor( 0.5 *  TotalMuestras) then
  Metodo = "brute"

 else
  Metodo = "random"
 end
end

while  iC < TotalMuestras  do
 if Metodo == "brute" then
  table.insert(IPv4Candidatos, iC)
  iC = iC + 1
elseif Metodo == "random" then -- We are going to use a extra table
  iAux = math.random( RangoAleatorio )
  bBool = true
  for _ , Canditato in ipairs(IPv4Candidatos) do
   if Canditato == iAux then
     bBool = false
     break
   end
  end
  if bBool then
   table.insert(IPv4Candidatos, iAux)
   iC = iC + 1
  end
 end
end

Segmentos,sError = ipOps.get_parts_as_number(IPv6Base)
 if ( sError ~= nil) then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" VMware(dynamic): ERROR with" .. IPv6Base .. "/64. Error Message: " .. sError)
  return nil
 end

IPv6Prefix = itsismx.DecToHex(Segmentos[1]) .. ":" ..
itsismx.DecToHex(Segmentos[2]) .. ":" ..
itsismx.DecToHex(Segmentos[3]) .. ":" .. itsismx.DecToHex(Segmentos[4]) ..
":" .. sHexadecimal .. "FF:FE"
IPv6Prefix, sError = ipOps.ip_to_bin(IPv6Prefix.."00:0000" ) -- FE::/104 ~=
FE00::/112
if sError ~= nil then
 stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" VMware(dynamic): ERROR with" .. IPv6Prefix.."00:0000" .. " " .. sError)
end

for _ , Candidato in ipairs( IPv4Candidatos ) do
  iAux = itsismx.DecToHex(Candidato)
  iAux = itsismx.HextToBin(iAux)

  if  16 - Wellknown < #iAux  then --too big?
   while   16 - Wellknown < #iAux  do iAux = iAux:sub(2) end
```

```
   elseif 16 - Wellknown > #iAux then -- too small?
    while   16 - Wellknown > #iAux  do iAux = 0 .. iAux end
   end

   if  Wellknown == 0  then
    IPv6Candidato = IPv6Prefix:sub(1,104) .. iAux .. "00000000"
   else
 IPv6Candidato = IPv6Prefix:sub(1,104) .. BitsKnown .. iAux .. "00000000"
   end

   IPv6Candidato = ipOps.bin_to_ip( IPv6Candidato )

 TheNext, TheLast, sError = ipOps.get_ips_from_range(IPv6Candidato .. "/120")
   if ( sError ~= nil) then
    stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
       " VMware(dynamic): ERROR with (Variable IPv6Candidato)" ..
       IPv6Candidato .. "/120. Error Message: " .. sError)
    return nil
   end

   stdnse.print_verbose(4, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " VMware(dynamic): Will be add 256 targets to the scan list: " .. IPv6Candidato .. "/120")

   repeat

    stdnse.print_verbose(5, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " VMware(dynamic): Added IPv6 address " .. TheNext .. " to the host scanning list...")

    bool ,err =  target.add(TheNext )
    if ( (bool) and not(SaveMemory) ) then
     table.insert( nmap.registry.slaac_PreHost, TheNext )

    elseif not(bool) then
      stdnse.print_verbose(6, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " VMware(dynamic):: Had been a error adding the node " .. TheNext .. " which is: " ..err )
    end
 TheNext = itsismx.GetNext_AddressIPv6(TheNext,120, IPv6ExMechanism)
    until not ipOps.ip_in_range(TheNext, IPv6Candidato .. "/120" )
 end
 return #nmap.registry.slaac_PreHost
end

local Vmware_Range_005056 = function ( IPv6Base, nBits, Metodo )
 local hosts, sError,IPv6Segmentos = nil , nil, nil

 IPv6Segmentos,sError = ipOps.get_parts_as_number(IPv6Base)

 if sError ~= nil then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
      " VMware(static): Error: " .. sError )
  return nil, sError
 end

 IPv6Base = itsismx.DecToHex(IPv6Segmentos[1]) .. ":" ..
 itsismx.DecToHex(IPv6Segmentos[2]) .. ":" ..
 itsismx.DecToHex(IPv6Segmentos[3]) .. ":" ..
 itsismx.DecToHex(IPv6Segmentos[4]) .. ":" .. "0250:56ff:fe"
 if tonumber(nBits) > 18 then --24-6
  nBits = 18
  Metodo = "brute"
```

```
 sError = " \n VMware Static MAC: Was need to reduce the bits to 18."
 elseif tonumber(nBits) > 16 then -- IF we are going to search for 25% of host... then
 brute force.
  Metodo = "brute" -- Probably is more efficient than random
 end

 if Metodo == nil then
  hosts = Random_Range(IPv6Base,nBits )
 elseif Metodo == "random" then
  hosts = Random_Range(IPv6Base,nBits )
 elseif Metodo == "brute" then
  hosts = Brute_Range(IPv6Base,nBits )
 else -- ERROR!
 return nil, "ERROR: The compute mechanism is incorrect: " .. Metodo
 end

 return hosts , sError
end

local getSlaacCandidates = function ( IPv6Prefix , HighPart )

 local hosts, sError, aux =nil, ""
 local _, OUI, hexadecimal, bitsAlto
 local Metodo, NumBits = stdnse.get_script_args("itsismx-slaac.compute",
 "itsismx-slaac.nbits")
 local IPv6Base, IPv6Segmentos
 local FinalHost = 0

 if NumBits == nil then
   NumBits = 11
 elseif tonumber(NumBits) < 1 then
  NumBits = 1
  sError = "Was add a very small value to nbits. Was fixed to 1"
 elseif tonumber(NumBits) > 24 then
  NumBits = 24
  Metodo = "brute"
  sError = "Was add a very high value to nbits. Was fixed to 24"
 elseif tonumber(NumBits) > 21 and tonumber(NumBits) <= 24  then
  Metodo = "brute"
 end

 for _ , OUI in ipairs(HighPart) do

 math.randomseed ( nmap.clock_ms() ) -- We are going to use Random values, so
 Seed!
 if #OUI == 6 then -- Our clasic case! (And some Virtual mahcines cases too)

   hexadecimal = tonumber(OUI,16)
 hexadecimal = bit32.replace( hexadecimal , 2,16,2) -- This or AND
 bitsAlto = itsismx.DecToHex( hexadecimal) -- This ignore the high part...
   while #bitsAlto < 6 do
    bitsAlto = "0" .. bitsAlto
   end

   IPv6Base, sError = ipOps.expand_ip(IPv6Prefix)
 if ( sError ~= nil ) then -- Weak point if the IPv6 address is bad formed
    return nil, sError
   end
   IPv6Segmentos = ipOps.get_parts_as_number(IPv6Base)
 IPv6Base = itsismx.DecToHex(IPv6Segmentos[1]) .. ":" ..
```

```
itsismx.DecToHex(IPv6Segmentos[2]) .. ":" ..
itsismx.DecToHex(IPv6Segmentos[3]) .. ":" ..
itsismx.DecToHex(IPv6Segmentos[4]) .. ":" ..
       bitsAlto:sub(1,4) .. ":" .. bitsAlto:sub(5,6) .. "ff:fe"

  if Metodo == nil then
   hosts = Random_Range(IPv6Base,NumBits )
   Metodo = "random"
  elseif Metodo == "random" then
   hosts = Random_Range(IPv6Base,NumBits )
  elseif Metodo == "brute" then
   hosts = Brute_Range(IPv6Base,NumBits )
  else -- ERROR!
return nil, "ERROR: The compute mechanism is incorrect: " .. Metodo
  end

  if hosts == nil then
sError = "\n There was a error with the Prefix: " .. IPv6Prefix ..
     " or maybe with the OUI: " ..  HighPart ..
     " you can use -dddd for more information"
  end

 elseif (OUI == "VMware-Alls" ) then  --VMware Cases!

  hosts, sError = Vmware_Range_000C29(IPv6Prefix,NumBits, Metodo  )

  if hosts == nil then
sError = " \n The compute of VMware 00:0C:29:WW:TT:UU had a error for the prefix " ..

IPv6Prefix .. " you can use -d for find the error (probably human)." .. sError
   hosts = 0
  end

  math.randomseed ( nmap.clock_ms() ) -- We update the Seed again.
  aux  = Vmware_Range_005056 (IPv6Prefix,NumBits, Metodo )

  if hosts == nil then
sError = " The compute of VMware 00:50:56:XX:YY:ZZ had a error for the prefix " ..
IPv6Prefix .. " you can use -d for find the error (probably human)." .. sError
  else
   host = aux + hosts
  end


elseif (OUI == "VMware-Static" ) then --VMware Static MAC address assignation.
  hosts = Vmware_Range_005056 (IPv6Prefix,NumBits, Metodo )
  if hosts == nil then
sError = " The compute of VMware 00:50:56:XX:YY:ZZ had a error for the prefix " ..
IPv6Prefix .. " you can use -dddd for find the error (probably human)."
  end
elseif (OUI == "VMware-Dynamic" ) then --VMware Dynamic MAC address
assignation.
  hosts = Vmware_Range_000C29 (IPv6Prefix,NumBits, Metodo )
  if hosts == nil then
sError = " The compute of VMware 00:0C:29:WW:TT:UU had a error for the prefix " ..
IPv6Prefix .. " you can use -dddd for find the error (probably human)."
  end
 end

 if (hosts ~= nil) then FinalHost = FinalHost + hosts end
```

```
 end

 if sError == nil then
  sError = ""
 end
 return FinalHost, sError
end

local getMacPrefix = function ( Vendedores, MacList   )

 local sLista, hLista = {},{}
 local hMac, sID, _, sUserMac

 if type(Vendedores) == "string" then -- This only make the search easy...
  table.insert(sLista, Vendedores)
 elseif type(Vendedores) ==  "table" then
  sLista = Vendedores
 else
  return nil
 end

 for _, sUserMac in pairs ( sLista)  do
  sUserMac = sUserMac:lower()

  if itsismx.Is_Valid_OUI(sUserMac ) then
    table.insert(hLista,sUserMac )
    stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": " .. " Was added the OUI " .. sUserMac .. " provided by the user. " )

  else  -- Name of a companie
   for hMac, sID in pairs( MacList ) do
    sID = sID:lower()

    if sID:find(sUserMac) ~= nil then
     table.insert(hLista,hMac )
     stdnse.print_verbose(2, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ".Vendors: " .. " Adding " .. hMac .. " OUI for the vendor: " .. sUserMac ..
      " ( " .. sID .. " )")
    end
   end

   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": " .. " Were added " .. #hLista .. " OUI for the vendor: " .. sUserMac )
  end
 end

 return hLista
end

local Prescanning = function ()

 local MacList, PrefixAux, _
 local bSalida, tSalida = false , { Nodos={}, Error=""}
 local MacUsers,IPv6User,VM =
 stdnse.get_script_args("itsismx-slaac.vendors","itsismx-subnet","itsismx-slaac.vms")
 local IPv6Knowns = nmap.registry.itsismx.PrefixesKnown
 local PrefixHigh, IPv6Total = {}, {}
 local IPv6_Add, IPv6_Prefix

 stdnse.print_verbose(2, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
```

```
  ": Begining the Pre-scanning work... "    )

bSalida, MacList = datafiles.parse_mac_prefixes ()

if not bSalida then
 tSalida.Error = " The Mac Prefixes file wasn't find!"
 return bSalida, tSalida
end

if (MacUsers == nil ) then
 if  VM == nil then
  PrefixHigh = getMacPrefix( "002170",MacList  )
 end
else
 PrefixHigh = getMacPrefix( MacUsers,MacList  )
end

local sVM = " Was added the next VM plataforms to the search: "
if  VM ~= nil then

if type(VM) == "number" then -- Default case (Strange, but if the argument is empty
is a number)
  table.insert(PrefixHigh, "VMware-Alls")
  table.insert(PrefixHigh, "001C42")
  table.insert(PrefixHigh, "001851")
  table.insert(PrefixHigh, "080027")
  table.insert(PrefixHigh, "525400")

sVM = sVM .. " VMware VMs, Virtual Box VMs, Parallels Virtuozzo & Desktop VMs,
Microsoft Virtual PC VMs, Linux QUEMU VMs"
 else --WPVML

  if VM:find("W") then  --VMware case
table.insert(PrefixHigh, "VMware-Alls") -- will work in other part.
   sVM = sVM .. " VMware VMs ,"
  elseif VM:find("wS") then  --VMware Static/Manual assignation
table.insert(PrefixHigh, "VMware-Static") -- will work in other part.
   sVM = sVM .. " VMware VMs (Manual) ,"
  elseif VM:find("wD") then  --VMware Static/Manual assignation
table.insert(PrefixHigh, "VMware-Dynamic") -- will work in other part.
   sVM = sVM .. " VMware VMs (Dynamic) ,"
  end

  if VM:find("P") then  --Parallels  case
   table.insert(PrefixHigh, "001C42")
   table.insert(PrefixHigh, "001851")
   sVM = sVM .. " Parallels Virtuozzo & Desktop VMs ,"
  elseif VM:find("pV") then
   table.insert(PrefixHigh, "001851") --Parallels  Virtuozzo
   sVM = sVM .. " Parallels Virtuozzo VMs ,"
  elseif VM:find("pD") then
   table.insert(PrefixHigh, "001C42") --Parallels  Desktop
   sVM = sVM .. " Parallels Desktop VMs ,"
  end

  if VM:find("V") then  -- Virtual-Box case
   table.insert(PrefixHigh, "080027")
   sVM = sVM .. " Virtual Box VMs ,"
  end
```

```
  if VM:find("M") then  --Virtual PC case
   table.insert(PrefixHigh, "VMware-Alls")
   sVM = sVM .. " Microsoft Virtual PC VMs ,"
  end

  if VM:find("L") then  --QEMU case
   table.insert(PrefixHigh, "525400")
   sVM = sVM .. " Linux QUEMU VMs"
  end


 end

 stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
  ": "  .. sVM )
end

bSalida = false
if IPv6User == nil and IPv6Knowns == nil then
tSalida.Error = "There is not IPv6 subnets to try to scan!. You can run a script for
discovering or adding your own" ..
      "  with the arg: itsismx-subnet."
 return bSalida, tSalida
end

if IPv6Knowns ~= nil then
 stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" Number of Prefixes Known from other sources: " .. #IPv6Knowns )
 for _ , PrefixAux in ipairs(IPv6Knowns) do
  table.insert(IPv6Total,PrefixAux )
 end
end

if IPv6User ~= nil then
 if type(IPv6User) ==  "string" then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   " Number of Prefixes Known from other sources: 1 "     )
  table.insert(IPv6Total,IPv6User )
 elseif type(IPv6User) ==  "table" then
  stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
" Number of Prefixes Known from other sources: " .. #IPv6User )
for _ , PrefixAux in ipairs(IPv6User) do -- This is healthy for my mind...
   table.insert(IPv6Total,PrefixAux )
  end
 end

end

for  _, PrefixAux in ipairs(IPv6Total) do

IPv6_Add, IPv6_Prefix = itsismx.Extract_IPv6_Add_Prefix(PrefixAux)
 if( IPv6_Prefix ~= 64) then
tSalida.Error = tSalida.Error .. "\n" .. PrefixAux .. " Must have a prefix of 64
(Was omited)"
 else

tSalida.Nodos,tSalida.Error = getSlaacCandidates ( IPv6_Add , PrefixHigh )

 end
end
```

```
 return true, tSalida
end

local Hostscanning = function( host)
 local tSalida = { Nodos=nil, Error=""}
 local aux

 stdnse.print_verbose(4, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
   ": Begining the Host-scanning results... " )

 if nmap.registry.itsismx == nil then
 tSalida.Error = "You must first initialize the global register Itsismx (There is
 a global function for that!)"
  return false, tSalida
 end

 aux = nmap.registry.itsismx.sbkmac
 if aux == nil then
 tSalida.Error = "The global register Itsismx wasn't initialized correctly
 (There is a global function for that!)"
  return false, tSalida
 end

 aux[#aux +1] = host.ip
 nmap.registry.itsismx.sbkmac = aux

 tSalida.Nodos = host.ip  -- This rule ALWAY IS ONE ELEMENT!

 return true, tSalida
end
prerule = function()
   if ( not(nmap.address_family() == "inet6") ) then
 stdnse.print_verbose("%s Need to be executed for IPv6.", SCRIPT_NAME)
  return false
 end

 if ( stdnse.get_script_args('newtargets')==nil ) then
 stdnse.print_verbose(1, "%s Will only work on pre-scanning. The argument
 newtargets is needed for the host-scanning to work.", SCRIPT_NAME)
 end

 return true
end
hostrule = function(host)
 local Totales, Objetivo, bMatch, sMatch = nmap.registry.slaac_PreHost

 if Totales == nil  then return false end

 for _, Objetivo in pairs( Totales ) do

  bMatch, sMatch = ipOps.compare_ip(host.ip, "eq", Objetivo)
  if bMatch == nil then
   stdnse.print_verbose(1, "\t hostrule  had a error with " ..
        host.ip .. "\n Error:" .. sMatch )
  elseif bMatch then
   return true
  end
 end
```

```
 return false
end


action = function ( host )

 local tOutput = {}
 tOutput = stdnse.output_table()
 local bExito = false
 local tSalida =  { Nodos={}, Error=""}
 local  bHostsPre, sHostsPre
 local Nodes = {} -- Is a Auxiliar
 itsismx.Registro_Global_Inicializar("sbkmac") -- Prepare everything!
 local SaveMemory = stdnse.get_script_args( "itsismx-SaveMemory" )

 if ( SCRIPT_TYPE== "prerule" ) then

  nmap.registry.slaac_PreHost = {}
  bExito , tSalida = Prescanning()
  tOutput.warning = tSalida.Error

  if bExito then

   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 " Finished. Were added : " .. tSalida.Nodos .. " to the scan phase." )

   if SaveMemory then
    stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": Warning: Was given the option to save memory, therefore the final report will "
 ..  " be not correct")
   else
   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
 ": Was unable to add nodes to the scan list due this error: " .. tSalida.Error )

   end

   if tSalida.Error ~= "" then
       stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
      ".Warnings:  " ..  tSalida.Error )
   end

  end


 elseif ( SCRIPT_TYPE== "hostrule" )  then
   bExito , tSalida = Hostscanning(host)
   tOutput.warning = tSalida.Error

   if ( bExito ~= true) then
   stdnse.print_verbose(1, SCRIPT_NAME .. "." .. SCRIPT_TYPE ..
        " Error: " .. tSalida.Error)
   end

   tOutput.name = "Host online - IPv6 address SLAAC"
 table.insert(tOutput,tSalida.Nodos) --This will be alway one single host.
 end

 return stdnse.format_output(bExito, tOutput);
end
```

C.2.7. itsismx-report.nse

```
--local bin = require "bin"
--local nmap = require "nmap"
--local packet = require "packet"
local stdnse = require "stdnse"
local itsismx = require "itsismx"
local ipOps = require "ipOps"

description = [[
  A general overview of the previews 6 scripts will be displayed.  Only work on
  the last phase of Nmap and work only with the final variables made by each one
  of the scripts.
]]

---
-- @usage
-- nmap -6 --script  itsismx-report -v
--
-- @output
-- Post-scan script results:
-- | itsismx-report:
-- |   Subnets:
-- |     No Subnets were discovered using this series of script.
-- |   Hosts:
-- |      SLAAC       : Discovered 3 nodes online which are 17.647058823529% Of total nodes discovered.
-- |      MAP 6 to 4  : Discovered 8 nodes online which is 47.058823529412 Of total nodes discovered.
-- |      Low Bytes   : Discovered 5 nodes online which is 29.411764705882 Of total nodes discovered.
-- |_     Words        : Discovered 1 nodes online which is 5.8823529411765 Of total nodes discovered.
--

-- Version 1.1
-- Updated 01/11/2013 - V1.1 - Majors updates for a best report.
--  Created 28/09/2013 - v0.1 - created by Ing. Raul Fuentes <ra.fuentess.sam+nmap@gmail.com>
--

author = "Raul Fuentes"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"broadcast", "safe"}

Agregar_Nodo_Descubierto = function( Original,  Nodo, Tactic)

 local Host ={IPv6="", Fuentes={}}
 local iIndex = 1

 for iIndex , Host in ipairs(Original) do

  if ipOps.compare_ip(Nodo, "eq", Host.IPv6) then
   table.insert ( Original[iIndex].Fuentes, Tactic )
   return Original, true
  end
 end

 table.insert(Original, {IPv6=Nodo, Fuentes={Tactic}} )

 return Original, false
end

postrule  = function()

 if ( not(nmap.address_family() == "inet6") ) then
```

```
 stdnse.print_verbose("%s Need to be executed for IPv6.", SCRIPT_NAME)
  return false
 end

 local Global = nmap.registry.itsismx

 if ( Global == nil) then
 stdnse.print_verbose("%s Need to be executed after the other itsismx scripts.
 \n or there wasn't reports. ", SCRIPT_NAME)
  return false
 end

 return true
end

action = function()

 local lSlaac = nmap.registry.itsismx.sbkmac
 local lMap64 = nmap.registry.itsismx.Map4t6
 local lLwByt = nmap.registry.itsismx.LowByt
 local lWords = nmap.registry.itsismx.wordis
 local lDhcp6 = nmap.registry.itsismx.PrefixesKnown

 local Hosts = { {IPv6="", Fuentes={}} } -- Fuentes will have the list of
 local Aux , bAux, bBoolean= {IPv6="", Fuentes={}}, false, false
 local Total , SubRedes, Address, Repetidos, Tactica = 0 , 0 ;
 local tOutput = stdnse.output_table()
 if lDhcp6 ~= nil then
  SubRedes = SubRedes + #lDhcp6
 else
  lDhcp6 = {}
 end

 if lSlaac ~= nil then
  Total = Total + #lSlaac
  for _ , Address in ipairs(lSlaac) do
 Hosts, bAux = Agregar_Nodo_Descubierto( Hosts , Address, "SLAAC")
  bBoolean = bBoolean or bAux
  end
 else
  lSlaac = {}
 end

 if lMap64 ~= nil then
  Total = Total + #lMap64
  for _ , Address in ipairs(lMap64) do
 Hosts, bAux = Agregar_Nodo_Descubierto( Hosts , Address, "Mapt6to4")
  bBoolean = bBoolean or bAux
  end
 else
  lMap64 = {}
 end
 if lLwByt ~= nil then
  Total = Total + #lLwByt
  for _ , Address in ipairs(lLwByt) do
 Hosts, bAux = Agregar_Nodo_Descubierto( Hosts , Address, "Low-Bytes")
  bBoolean = bBoolean or bAux
  end
 else
  lLwByt = {}
```

```
end

if lWords ~= nil then
 Total = Total + #lWords
 for _ , Address in ipairs(lWords) do
Hosts, bAux = Agregar_Nodo_Descubierto( Hosts , Address, "Words")
 bBoolean = bBoolean or bAux
 end
else
 lWords = {}
end

tOutput.Subnets={}
tOutput.Hosts={}
tOutput.Repeats={}

local Auxiliar = { ["Technique"]="",["List"]={}}
if SubRedes == 0  then
table.insert(tOutput.Subnets, "No Subnets were discovered using this series
of script. ")
else
 if #lDhcp6 ~= 0 then
table.insert( tOutput.Subnets, #lDhcp6 .. " Were confirmed to exits using the
spoofing technique with DHCPv6" )
 end

end

if Total == 0 then
table.insert(tOutput.Hosts, "No Hosts were discovered using this series of
script. ")
else

 if #lSlaac ~= 0 then
  Auxiliar = { ["Technique"]="",["List"]={}}
Auxiliar.Technique = "SLAAC - Discovered " .. #lSlaac .. " nodes online which are "
.. #lSlaac/Total * 100 .. "% Of total nodes discovered."
   if nmap.verbosity >= 1 then
for _ , Address in ipairs(lSlaac) do table.insert(Auxiliar.List, Address) end
   end
   table.insert(tOutput.Hosts, Auxiliar)
 end


 if #lMap64 ~= 0 then
  Auxiliar = { ["Technique"]="",["List"]={}}
Auxiliar.Technique = " Map6to4 - Discovered " .. #lMap64 .. " nodes online which is
" .. #lMap64/Total * 100 .. " Of total nodes discovered."
  if nmap.verbosity() >= 1 then
for _ , Address in ipairs(lMap64) do table.insert(Auxiliar.List, Address) end
   end
   table.insert(tOutput.Hosts, Auxiliar)
 end

 if #lLwByt ~= 0 then
  Auxiliar = { ["Technique"]="",["List"]={}}
Auxiliar.Technique = " Low Bytes - Discovered " .. #lLwByt .. " nodes online which
is " .. #lLwByt/Total * 100 .. " Of total nodes discovered."
  if nmap.verbosity() >= 1 then
for _ , Address in ipairs(lLwByt) do table.insert(Auxiliar.List, Address) end
```

```
    end
    table.insert(tOutput.Hosts, Auxiliar)
  end

  if #lWords ~= 0 then
   Auxiliar = { ["Technique"]="",["List"]={}}
 Auxiliar.Technique = " Words - Discovered " .. #lWords .. " nodes online which is "
 .. #lWords/Total * 100 .. " Of total nodes discovered."

   if nmap.verbosity() >= 1 then
 for _ , Address in ipairs(lWords) do table.insert(Auxiliar.List, Address) end
    end
    table.insert(tOutput.Hosts, Auxiliar)

  end
 end

 if {bBoolean} then
  for _,  Aux in ipairs(Hosts) do
   Repetidos = ""
   if #Aux.Fuentes > 1 then
 for Tactica in Aux.Fuentes do Repetidos = Repetidos .. " " .. Tactica end
 table.insert( tOutput.Repeats, "The node " .. Aux.IPv6 .. " was found by the
 differents mechanis: " .. Repetidos)
   end
  end
 end
 return tOutput
end
```

Bibliografía

[1] C. Stöcker and J. Horchert, "Mapping the internet: A hacker's secret internet census," *SPIEGEL Online International*, marzo 2013, accedido el: 18/06/2013. [Online]. Available: http://www.spiegel.de/international/world/hacker-measures-the-internet-illegally-with-carna-botnet-a-890413.html

[2] P. Shukla, "Compromised devices of the carna botnet," Presentación en línea, Abril 2013, accedido el: 19/06/2013. [Online]. Available: http://docs.google.com/file/d/0BxMgdZPXsSLBQWRIZDB0cTdGU2c/edit?pli=1

[3] "Ipv6 address autoconfiguration," Microsoft corporation, Tech. Rep., Agosto 2010, accedido el: 18/06/2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa917171.aspx

[4] D. Malone, "Observations of ipv6 addresses." assive and Active Measurement Conference, Abril 2008, accedido el: 18/06/2013. [Online]. Available: http://www.maths.tcd.ie/~dwmalone/p/addr-pam08.pdf

[5] F. Gont and T. Chown, *Network Reconnaissance in IPv6 Networks - draft-ietf-opsec-ipv6-host-scanning-01*, Internet Engineering Task Force, Abril 2013. [Online]. Available: http://tools.ietf.org/html/rfc1122

[6] N. Marsh, *Nmap Cookbook: The Fat-free Guide to Network Scanning*, ser. YBP ORDER. CreateSpace Independent Publishing Platform, 2010. [Online]. Available: http://books.google.com.mx/books?id=U4H3QwAACAAJ

[7] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*, 1st ed. Nmap Project, enero 2009.

[8] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet Security; Repelling the Wily Hacker*, 2nd ed. Reading, MA: Addison-Wesley, 2003. [Online]. Available: http://www.wilyhacker.com/

[9] R. Zakon, *RFC 2235 - HobbesÍnternet Timeline*, Internet Engineering Task Force, octubre 1997. [Online]. Available: http://tools.ietf.org/html/rfc2235

[10] I. S. I. U. of Southern California, *RFC 791 - INTERNET PROTOCOL*, Internet Engineering Task Force, septiembre 1981. [Online]. Available: http://tools.ietf.org/html/rfc791

[11] J. Mogul and J. Postel, *RFC 950 - Internet Standard Subnetting Procedure*, Internet Engineering Task Force, agosto 1985. [Online]. Available: http://tools.ietf.org/html/rfc950

[12] V. Fuller, T. Li, J. Yu, and K. Varadhan, *RFC 1519 - Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*, Internet Engineering Task Force, septiembre 1993. [Online]. Available: http://tools.ietf.org/html/rfc1519

[13] P. Srisuresh and K. Egevang, *RFC 3022 -Traditional IP Network Address Translator (Traditional NAT)*, Internet Engineering Task Force, enero 2001. [Online]. Available: http://tools.ietf.org/html/rfc3022

[14] C. Aoun and E. Davies, *RFC 4966 - Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status*, Internet Engineering Task Force, julio 2007. [Online]. Available: http://tools.ietf.org/html/rfc4966

[15] S. Bradner and A. Mankin, *RFC 1752 - The Recommendation for the IP Next Generation Protocol*, Internet Engineering Task Force, enero 1995. [Online]. Available: http://tools.ietf.org/html/rfc1752

[16] S. Deering and R. Hinden, *RFC 2460 - Internet Protocol, Version 6 (IPv6) Specification*, Internet Engineering Task Force, diciembre 1998. [Online]. Available: http://tools.ietf.org/html/rfc2460

[17] T. Chown, *RFC 5157 - IPv6 Implications for Network Scanning*, Internet Engineering Task Force, marzo 2008. [Online]. Available: http://tools.ietf.org/html/rfc5157

[18] IANA, *Internet Protocol Version 6 Address Space*, Febrero 2013, accedido el: 18/06/2013. [Online]. Available: http://www.iana.org/assignments/ipv6-address-space/ipv6-address-space.xml

[19] R. Hinden and S. Deering, *RFC 4291 - IP Version 6 Addressing Architecture*, Internet Engineering Task Force, febrero 2006. [Online]. Available: http://tools.ietf.org/html/rfc4291

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*, Internet Engineering Task Force, junio 1999. [Online]. Available: http://tools.ietf.org/html/rfc2616

[21] J. Postel, *RFC 777 - Internet Control Message Protocol*, Internet Engineering Task Force, abril 1981. [Online]. Available: http://tools.ietf.org/html/rfc777

[22] F. Gont, *RFC 6633 - Deprecation of ICMP Source Quench Messages*, Internet Engineering Task Force, mayo 2012. [Online]. Available: http://tools.ietf.org/html/rfc6633

[23] F. Gont and C. Pignataro, *RFC 6918 - Formally Deprecating Some ICMPv4 Message Types*, Internet Engineering Task Force, abril 2013. [Online]. Available: http://tools.ietf.org/html/rfc6918

[24] D. C. Plummer, *RFC 826 - An Ethernet Address Resolution Protocol*, Internet Engineering Task Force, noviembre 1982. [Online]. Available: http://tools.ietf.org/html/rfc826

[25] A. Conta, S. Deering, and M. Gupta, *RFC 4443 - Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*, Internet Engineering Task Force, marzo 2006. [Online]. Available: http://tools.ietf.org/html/rfc4443

[26] T. Simonite, "What happened when one man pinged the whole internet," artículo electrónico, Abril 2013, accedido el: 18/06/2013. [Online]. Available: http://www.technologyreview.com/contributor/tom-simonite/

[27] H. Moore, "Security flaws in universal plug and play," enero 2013, accedido el: 18/06/2013. [Online]. Available: https://community.rapid7.com/docs/DOC-2150

[28] J. P. García-Moran, Y. Fernandez Hansen, A. Ochoa Martin, and A. A. Ramos Varón, *Hacking y Seguridad en Internet*, ser. Edición 2011 year=2011, publisher=rama.

[29] Fyodor, "Compromised devices of the carna botnet," Lista de correo electrónico, Marzo 2013, accedido el: 19/06/2013. [Online]. Available: http://seclists.org/nmap-dev/2013/q1/371

[30] IAB and IESG, *RFC 1881 - IPv6 Address Allocation Management*, Internet Engineering Task Force, diciembre 1995. [Online]. Available: http://tools.ietf.org/html/rfc1881

[31] R. Hinden, S. Deering, and E. Nordmark, *RFC 3587 - IPv6 Global Unicast Address Format*, Internet Engineering Task Force, agosto 2003. [Online]. Available: http://tools.ietf.org/html/rfc3587

[32] IAB and IESG, *RFC 3177 - IAB/IESG Recommendations on IPv6 Address Allocations to Sites*, Internet Engineering Task Force, septiembre 2001. [Online]. Available: http://tools.ietf.org/html/rfc3177

[33] T. Narten, G. Huston, and L. Roberts, *RFC 6177 - IPv6 Address Assignment to End Sites*, Internet Engineering Task Force, marzo 2011. [Online]. Available: http://tools.ietf.org/html/rfc6177

[34] S. Thomson, T. Narten, and T. Jinmei, *RFC 4862 - IPv6 Stateless Address Autoconfiguration*, Internet Engineering Task Force, septiembre 2007. [Online]. Available: http://tools.ietf.org/html/rfc4862

[35] T. Narten, R. Draves, and S. Krishnan, *RFC 4941 - Privacy Extensions for Stateless Address Autoconfiguration in IPv6*, Internet Engineering Task Force, septiembre 2007. [Online]. Available: http://tools.ietf.org/html/rfc4941

[36] "Guidelines for 64-bit global identifier (eui-64$^{\text{TM}}$)," IEE Standards Association, Tech. Rep., Noviembre 2012, accedido el: 18/06/2013. [Online]. Available: http://standards.ieee.org/develop/regauth/tut/eui64.pdf

[37] T. Narten, E. Nordmark, W. Simpson, and H. Soliman, *RFC 4861 - Neighbor Discovery for IP version 6 (IPv6)*, Internet Engineering Task Force, septiembre 2007. [Online]. Available: http://tools.ietf.org/html/rfc4861

[38] E. R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney, *RFC 3315 - Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*, Internet Engineering Task Force, julio 2003. [Online]. Available: http://tools.ietf.org/html/rfc3315

[39] J. Akko, J. Kempf, B. Zill, and P. Nikander, *RFC 3971 - SEcure Neighbor Discovery (SEND)*, Internet Engineering Task Force, marzo 2005. [Online]. Available: http://tools.ietf.org/html/rfc3971

[40] P. Nikander, J. Jempf, and E. Nordmark, *RFC 3756 - IPv6 Neighbor Discovery (ND) Trust Models and Threats*, Internet Engineering Task Force, mayo 2004. [Online]. Available: http://tools.ietf.org/html/rfc3756

[41] "Ipv6 rfcs and internet drafts (windows ce 5.0)," Microsoft corporation, Tech. Rep., Noviembre 2006, accedido el: 18/06/2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa450089.aspx

[42] T. Narten and R. Draves, *RFC 3041 - Privacy Extensions for Stateless Address Autoconfiguration in IPv6*, Internet Engineering Task Force, enero 2001. [Online]. Available: http://tools.ietf.org/html/rfc3041

[43] "Desktop operating system market share," Netmarketshare, Tech. Rep., 2013, accedido el: 28/06/2013. [Online]. Available: http://www.netmarketshare.com/operating-system-market-share.aspx

[44] C. Huitema, *RFC 4380 - Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs)*, Internet Engineering Task Force, febrero 2006. [Online]. Available: http://tools.ietf.org/html/rfc4380

[45] van Hauser, "Thc-ipv6-attack-toolkit," The Hacker Choice, Tech. Rep., mayo 2013, accedido el: 28/06/2013. [Online]. Available: http://www.thc.org/thc-ipv6/

[46] F. Gont, "6tools," Security/Robustness Assessment of IPv6 Neighbor Discovery Implementations, Tech. Rep., Noviembre 2012, accedido el: 03/03/2013. [Online]. Available: http://www.si6networks.com/tools/ipv6toolkit/si6networks-ipv6-nd-assessment.pdf

[47] D. Garcia and R. Sanchez, "Topera." 2da Conferencias Navaja Negra, Diciembre 2012, accedido el: 10/07/2013. [Online]. Available: http://toperaproject.github.io/topera/

[48] *Network Reconnaissance in IPv6 Networks - draft-ietf-opsec-ipv6-host-scanning-01*, Free Software Foundation, junio 2005. [Online]. Available: http://unixhelp.ed.ac.uk/CGI/man-cgi?awk

[49] C. Sanders, *Practical Packet Analysis: Using Wireshark to Solve Real-world Network Problems*, ser. No Starch Press Series. No Starch Press, 2007.

[50] S. Groat, M. Dunlop, R. Marchany, and J. Tront, "What dhcpv6 says about you," in *Internet Security (WorldCIS), 2011 World Congress on*, 2011, pp. 146–151, accedido el: 03/03/2013.

[51] *vSphere 4.1 - ESXi Installable and vCenter Server - ESXi Configuration Guide - Networking - Advanced Networking - MAC Address*, VMWare, accedido el: 28/06/2013. [Online]. Available: http://pubs.vmware.com/vsphere-4-esxi-installable-vcenter/index.jsp

[52] suzsuz, "Wide-dhcpv6," Repositorio de código fuente, diciembre 2005, accedido el: 28/06/2013. [Online]. Available: http://sourceforge.net/projects/wide-dhcpv6/

[53] "Windows seerver - dhcpv server," Microsoft, Tech. Rep., 2008, accedido el: 28/06/2013. [Online]. Available: http://technet.microsoft.com/en-us/windowsserver/dd448608.aspx

[54] I. Gashinksy, J. Jaeggli, and W. Kumari, *RFC 6583 - Operational Neighbor Discovery Problems*, Internet Engineering Task Force, marzo 2012. [Online]. Available: http://tools.ietf.org/html/rfc6583