Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Abschlussarbeit

im Studiengang Bachelor Informatik

Scannen von IPv6-Adressen in entfernten Subnetzen

von

Manuel Hachtkemper

Betreuer:	Prof. Dr. Martin Leischner
Zweitbetreuerin:	Prof. Dr. Kerstin Uhde
Eingereicht am:	16. September 2013

Erklärung

Manuel Hachtkemper
Paul-Langen-Str. 14a
53229 Bonn

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

.....	.....
Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

BGP	Border Gateway Protocol
DAD	Duplicate Address Detection
DHCPv6	Dynamic Host Configuration Protocol version 6
DNS	Domain Name System
EUI	Extended Unique Identifier
FQDN	Fully Qualified Domain Name
GPLv2	GNU General Public License version 2
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
ICMPv6	Internet Control Message Protocol version 6
IDS	Intrusion Detection System
IEEE	Institute of Electrical and Electronics Engineers
IEEE-SA	Institute of Electrical and Electronics Engineers Standards Association
IGMP	Internet Group Management Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
LIR	Local Internet Registry
MAC	Media Access Control
NAT	Network Address Translation
NIR	National Internet Registry
NSE	Nmap Scripting Engine
OUI	Organizational Unique Identifier
RFC	Request for Comments
RIPE NCC	Réseaux IP Européens Network Coordination Centre
RIR	Regional Internet Registry
SCTP	Stream Control Transmission Protocol
SLAAC	Stateless Address Autoconfiguration
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VPN	Virtual Private Network

1 Einleitung

1.1 Problemstellung

Mit der Einführung des Internet Protocol version 6 (IPv6) ist der Adressraum, im Vergleich zum Vorgänger Internet Protocol version 4 (IPv4), deutlich größer geworden. Während der gesamte Adressraum bei IPv4 32 Bit beträgt, ist dieser bei IPv6 auf 128 Bit gewachsen. Neben der allgemeinen Vergrößerung wurde auch eine Standardgröße von Subnetzen, zum Beispiel bei der automatischen Adressvergabe Stateless Address Autoconfiguration (SLAAC), mit 64 Bit eingeführt. Das bedeutet, dass sich potenziell 2^{64} Hosts in einem Subnetz befinden können (vgl. Chown 2008, S. 3 f.).
Da voraussichtlich die Anzahl von Hosts in einem Subnetz ähnlich bleiben wird, ist die Dichte von benutzten zu vorhandenen IPv6-Adressen deutlich geringer. Infolgedessen ist das von IPv4 bekannte Verfahren des Scannens von gesamten Subnetzen nicht mehr anwendbar, da dies, neben der Verursachung einer großen Menge Traffic, sehr lange dauern würde (vgl. Gont und Chown 2013, S. 3).
Request for Comments (RFC) 5157 und der darauf aufbauende Draft „Network Reconnaissance in IPv6 Network“ beschreiben andere Methoden, wie sowohl lokal als auch in entfernten Subnetzen IPv6-Adressen gefunden werden können. Es gibt seit neuestem eigenständige Implementierungen dieser Verfahren – allerdings sind bisher noch nicht für alle Verfahren Umsetzungen für den Netzwerkscanner Nmap vorhanden.

1.2 Motivation und Zielsetzung

Das Scannen von gesamten Subnetzen, zum Beispiel zur Inventarisierung von Hosts in einem Netzwerk, ist bei IPv4 weit verbreitet. Bei IPv6 ist dieses Verfahren aufgrund größerer Subnetze nicht mehr anwendbar.
Für lokale IPv6-Netze (in dem sich der Rechner befindet, mit dem gesucht werden soll) gibt es unter anderem die Möglichkeit, an die All-Nodes-Multicastadresse einen ICMPv6-Echo-Request zu schicken – mehr Informationen dazu unter Gont und Chown 2013, S. 15. Allerdings funktioniert dieses Verfahren nicht mehr, sobald das Paket über einen Router geleitet werden muss, also an ein anderes Subnetz adressiert ist. In größeren Netzwerken kann eine große Anzahl von Routern existieren, die verschiedene Gebäude oder Zweigstellen miteinander verbinden. Falls eine Inventarisierung in solchen Netzwerken anhand eines Scans stattfinden soll, müsste in jedem Subnetz ein Host zur Suche platziert sein beziehungsweise der Router eines Netzes benutzt werden. Allerdings ist dies nicht immer praktikabel, da eventuell kein Zugriff auf alle Router oder Lokalitäten besteht. Auch kann es sein, dass das Betriebssystem des Routers es nicht ermöglicht, (spezielle) ICMPv6-Multicastpakete

zu versenden, woraufhin Suchverfahren für entfernte Netze eine Alternative sein könnten. Deswegen ist das Ziel dieser Bachelorarbeit, die remote IPv6-Adresssuche weitgehend darzustellen und ausgewählte, bisher nicht implementierte Ansätze zur Verkleinerung des Suchraumes für die Benutzung mit Nmap umzusetzen. Die entstandenen Implementierungen werden nach Fertigstellung unter freier Lizenz, der GNU General Public License version 2 (GPLv2), veröffentlicht.

1.3 Methodik

In dieser Arbeit wird nach einer Literaturrecherche auf notwendige Grundlagen eingegangen. Zu diesen zählen verschiedene Adresstypen von globalen IPv6-Unicastadressen. Darauf aufbauend werden diverse in der Literatur vorgestellte Verfahren zur Suche dieser Adressen in entfernten IPv6-Subnetzen dargelegt. Eine Teilmenge davon sind die in „Network Reconnaissance in IPv6 Network“ Appendix 3 (vgl. Gont und Chown 2013, S. 34) aufgelisteten Methoden:

- Bereichsscan
- Suche nach SLAAC-Identifiern
- Suche nach eingebetteten IPv4-Adressen
- Suche nach Dienst-Port-Adressen
- Suche nach wortreichen Adressen

Jedes Verfahren wird kurz kritisch behandelt. Dazu zählt, ob es genau definiert und somit überhaupt in dieser Form umsetzbar ist oder Anpassungen für den Einsatz erforderlich sind – insofern sinnvoll, wird eine Implementierung umgesetzt.
Als Grundlage für die Implementierung (ohne weitere Evaluierung oder Vergleich mit anderen Programmen) dient der Netzwerkscanner Nmap, der diese Verfahren bisher nicht unterstützt. Die Gründe für die Wahl dieses Programmes sind vielfältig:

- Das Ziel der Entwicklung von Nmap war es, diverse Portscanner aus einem fragmentierten Bereich in einem Programm mit einheitlicher Oberfläche zu vereinigen (vgl. Lyon 2009, S. 19).
- Nmap ist eine freie Software (GPLv2) und somit potenziell veränder- oder erweiterbar. Die Erweiterbarkeit ist außerdem durch ein Pluginsystem, der Nmap Scripting Engine, gegeben (vgl. Lyon 2009, S. 313 ff.).
- Die Software ist auf diversen Betriebssystemen nutzbar, unter anderem GNU/Linux, Mac OS X und Windows (vgl. Lyon 2009, S. 69 ff.).

Vor der Implementierung wird zunächst evaluiert, in welcher Form dies geschehen soll – in der Nmap Scripting Engine, direkt im Nmap-Quelltext oder als eigenständiges Programm.
Als Abnahmekriterium der entstandenen Software wird lediglich geprüft, ob sie im

Normalfall fehlerfrei funktioniert und dass etwaige Eingaben die gewünschten Ausgaben (zur Verwendung mit Nmap) erzeugen können.
Anhand der theoretischen Minimallaufzeit und Messungen mit Nmap wird danach beurteilt, ob die Verfahren derzeit in der Praxis durchführbar sind.
Zuletzt werden verschiedene Methoden zur Abwehr von Adresssuchen kurz aufgezeigt, da auch Angreifer die Methoden zur Adresssuche verwenden können.

1.4 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen der Bildung von IPv6 Interface Identifiern behandelt. Zusätzlich erläutert werden der Arbeit zugrundeliegende Features und Funktionsweisen von Nmap und der Aufbau von ICMPv6-Echo-Request-Paketen. Bevor im nächsten Abschnitt auf die Suche von IPv6-Adressen eingegangen wird, ist vorher noch zu klären, ob die Entdeckung einer IPv6-Adresse sicherheitsrelevant ist.
Das Kapitel 3 widmet sich den IPv6-Adresssuchverfahren. Dabei wird jedes Verfahren vorgestellt und, insofern vorhanden, eine Implementierung aufgelistet. Da manche dieser Verfahren nach Adressen suchen, die von der Vergabepraxis des Administrators eines Netzes abhängen, werden daraufhin die Ergebnisse zweier Untersuchungen zur Adressvergabe genannt. Der Abschluss dieses Kapitels besteht aus einer Zusammenfassung der Adressverfahren in Form einer Tabelle.
Bevor die Adresssuchverfahren für den Netzwerkscanner Nmap umgesetzt werden, ist zu klären, was die Vor- und Nachteile der Implementierungswege sind – dies geschieht in Kapitel 4.
Die Implementierung wird in Kapitel 5 vorgestellt. Dabei wird vor allem auf die Bedienung und Ausgaben der entstandenen Programme eingegangen. Implementierungsdetails sind dem Anhang (oder `https://projects.fslab.de/redmine/projects/nse2013`) zu entnehmen, in dem der gesamte Quelltext aufgelistet ist.
Auf mögliche Abwehrmaßnahmen gegen Adresssuchverfahren wird in Kapitel 6 noch kurz eingegangen und in Kapitel 7 folgt die Zusammenfassung der Arbeit.

2 Grundlagen

2.1 IPv6-Adressen

Globale IPv6-Unicast-Adressen haben eine Länge von 128 Bit und lassen sich in Global Routing Prefix, Subnet Identifier und Interface Identifier zerlegen. Der Global Routing Prefix ist das zugewiesene Netz und der Subnet Identifier beinhaltet alle Subnetze, die vergeben werden können. Der Interface Identifier wird dazu genutzt, Interfaces auf einem Link zu unterscheiden und ist 64 Bit lang (außer bei globalen Adressen, die mit dem binären Wert 000 beginnen). (vgl. Hinden und Deering 2006, S. 2, 7 ff.)
Der Aufbau ist grafisch in Abbildung 1 dargestellt.

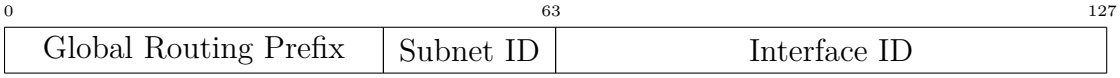

Abbildung 1: Globale IPv6-Unicast-Adressen (vgl. Hinden und Deering 2006, S. 9)

Zusammen sind Global Routing Prefix und Subnet Identifier der Prefix eines /64-Subnetzes.
Für alle im folgenden Kapitel beschriebenen Verfahren muss (bis auf wenige, nicht weiter erwähnte Ausnahmen) die sogenannte Duplicate Address Detection (DAD) für Unicast-Adressen durchgeführt werden, bevor eine Adresse an ein Interface gebunden werden kann (vgl. Thomson et al. 2007, S. 12). Dies bedeutet vereinfacht, dass via Multicast nachgefragt wird, ob ein anderer Host diese Adresse schon benutzt – wenn ja, dann muss eine andere Adresse gewählt werden (vgl. Hagen 2009, S. 126).

2.2 IPv6 Interface Identifier

2.2.1 Statisch zugewiesene Adressen

Wie auch schon bei IPv4 ist es bei IPv6 möglich, an einem Rechner Adressen statisch zu konfigurieren. Was für eine Adresse gewählt wird, obliegt dabei dem Nutzer. Eine Sammlung von Schemata, die gewählt werden können, ist in Kapitel 3 nachzulesen.

2.2.2 Stateless Address Autoconfiguration

Die Bildung des Interface Identifiers geschieht bei der Stateless Address Autoconfiguration (unter anderem) mit dem Institute of Electrical and Electronics Engineers (IEEE) Extended Unique Identifier (EUI). Bei Ethernet-Netzwerkinterfaces wird dazu die Media Access Control (MAC) Adresse verwendet, die eine Teilmenge des EUI-48 ist. Die ersten 4 Byte des EUI-48 werden vom IEEE vergeben (z.B. für die

Hersteller von Netzwerkkarten) und werden Organizational Unique Identifier (OUI) genannt. Die zweiten 4 Byte werden vom Hersteller an seine Geräte verteilt – sie heißen Manufacturer-Selected Extension Identifier.
Bit 6 des EUI-48 gibt an, ob eine Adresse universal (0) oder lokal (1) gültig ist und Bit 7 zeigt, ob sie individuell (0) verwendet wird oder einer Gruppe (1) zugewiesen wurde. Folgendes Verfahren wird nach RFC 4291 zur Generierung eines Interface Identifiers mittels EUI-48 durchgeführt (siehe auch Abbildung 2):

- 0xfffe wird zwischen dem dritten und vierten Byte des EUI-48 eingefügt, da der Interface Identifier 64 Bit lang sein muss.
- Bit 6 wird auf 1 gesetzt, was beim IPv6 Interface Identifier einen globalen Geltungsraum angibt.

Es war ursprünglich vorgesehen, für die MAC-Adresse 0xffff und für andere EUI-48 0xfffe einzufügen, jedoch gab es ein Missverständnis über den Unterschied von MAC und EUI-64, weswegen in früheren Spezifikationen auch 0xfffe für die MAC-Adresse verwendet wurde. Da dies in der Praxis zu keinem Problem führt und sich inzwischen eingebürgert hat, wird nun bei beiden Typen 0xfffe eingefügt. (vgl. Hinden und Deering 2006, S. 20 ff.)

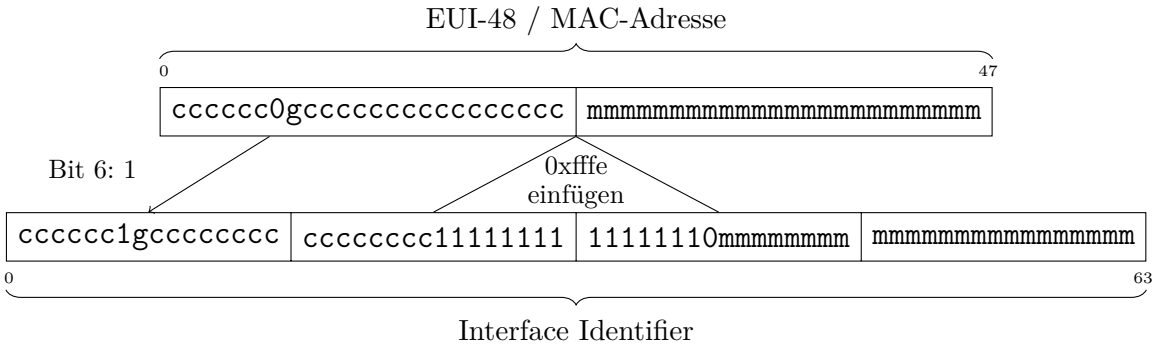

Abbildung 2: EUI-48 / MAC-Adresse zu Interface Identifier (vgl. Hinden und Deering 2006, S. 20 f.)

2.2.3 DHCPv6

Dynamic Host Configuration Protocol version 6 (DHCPv6) kann in zwei Modi betrieben werden – Stateless und Stateful.
Beim Stateless DHCPv6 können Rechner, die bereits eine Adresse haben, zum Beispiel via Stateless Address Autoconfiguration, weitere Parameter beziehen (unter anderem DNS-Server). Stateful DHCPv6 demgegenüber versorgt Hosts zusätzlich mit einer Adresse. (vgl. Hagen 2009, S. 324)
Wie schon bei IPv4 können Adressen entweder für jeden Host fest vergeben (z.B.

nach den Schemata in Kapitel 3) oder dynamisch aus einem Adresspool zugewiesen werden. Der Start- und Endpunkt des Pools kann dabei frei festgelegt werden, wobei die Prefixlänge 64 betragen sollte. Ob die Adressen dabei sequenziell oder zufällig vergeben werden, hängt von der Implementierung des DHCPv6-Servers ab. (vgl. Gont und Chown 2013, S. 9)
Es ist außerdem möglich, dass Clients eine temporäre Adresse vom Server anfordern. Der Server weist dem Client daraufhin eine temporäre Adresse zu, die das Format der Privacy Extension Adressen hat (siehe Kapitel 2.2.4). Bei DHCPv6 gibt es für die temporären Adressen keine Vorschriften, wie lange diese gültig sein sollen, wie sie vom Client benutzt werden oder wann neue temporäre Adressen gebildet werden sollen. (vgl. Droms et al. 2003, S. 25)
Da es bei DHCPv6 möglich ist, mehrere Adressen pro Interface zu benutzen, können die temporären Adressen neben den anderen zugewiesenen Adressen verwendet werden.

2.2.4 Privacy Extension

Die Stateless Address Autoconfiguration hat ein neues Problem geschaffen, das bei IPv4 noch nicht existierte. Bei IPv6 sind Adressen, wie in Kapitel 2.1 beschrieben, in Prefix und Interface Identifier trennbar. Das bedeutet, dass der von der Stateless Address Autoconfiguration geschaffene Interface Identifier in jedem Netzwerk gleich bleibt, unabhängig vom Prefix – bei IPv4 ist es zumindest nicht üblich, dass ein Netzwerkgerät in verschiedenen Netzen die gleiche Subnetzadresse erhält. Dadurch sind portable Geräte anhand ihres Interface Identifiers verfolgbar. Es könnte zum Beispiel ein Betreiber einer Webseite anhand von Log-Dateien feststellen, dass derselbe Rechner aus verschiedenen Netzwerken auf seine Seite zugegriffen hat und damit möglicherweise dieselbe Person verfolgen. (vgl. Narten et al. 2007, S. 7 f.)
Auch aufgrund dieser Verfolgbarkeit wurde die Privacy Extension geschaffen, deren (überarbeitete) Spezifikation in RFC 4941 vorliegt. Mit ihr können Clients einen temporären Interface Identifier mit zufälligem Wert generieren. Die daraus entstandene Adresse wird Privacy beziehungsweise Temporary Address genannt.
Das Verfahren setzt nach der Stateless Address Autoconfiguration an und verläuft in den folgenden Schritten – grafisch ist dies in Abbildung 3 dargestellt:

1. An dem von der SLAAC erstellten Interface Identifier wird ein History Value angehängt. Falls dieser nicht existiert, wird stattdessen eine neu generierte Zufallszahl verwendet.
2. Es wird eine (128 Bit lange) MD5-Hashsumme von dem erhaltenen Wert gebildet.

3. Die ersten 64 Bit der Hashsumme sollen zum Interface Identifier werden, weswegen Bit 6 auf 0 (lokal) gesetzt wird.
4. Nun wird verglichen, ob dieser Wert sich in einer Liste von reservierten Interface Identifiern befindet oder ob er bereits auf dem lokalen Rechner verwendet wird. Falls ja, werden die zweiten 64 Bit als neuer History Value gespeichert und es wird von Schritt 1 erneut begonnen.
5. Der erhaltene Wert wird als neuer Interface Identifier gespeichert.
6. Die zweiten 64 Bit ergeben den neuen History Value.
7. Prefix und Interface Identifier können konkateniert nach der Duplicate Address Detection als IPv6-Adresse genutzt werden. Falls die Adresse bereits von einem anderen Host benutzt wird, beginnt das Verfahren wieder von Schritt 1.

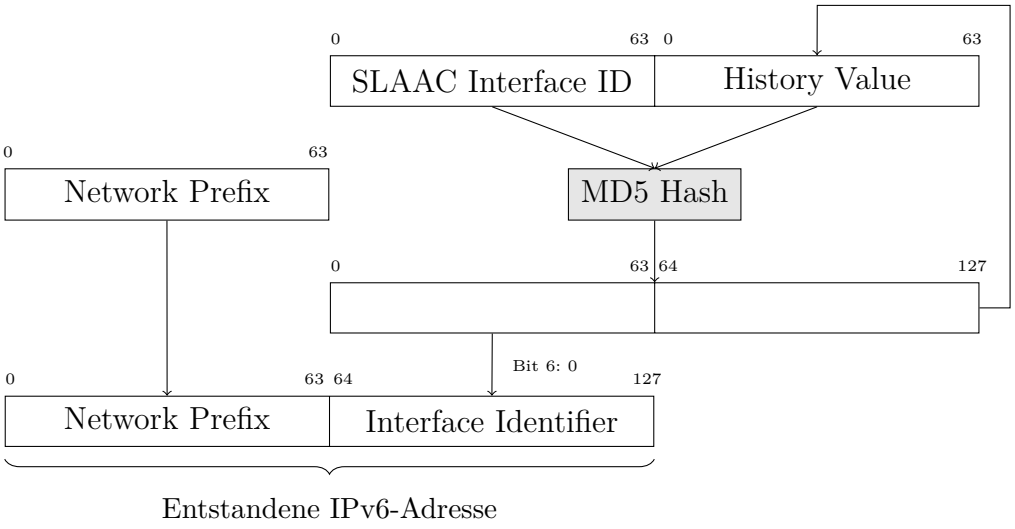

Abbildung 3: Funktion der Privacy Extension (nach Berrera et al. 2010)

Wieso ein History Value benutzt wird, statt den Interface Identifier komplett aus einer Zufallszahl zu generieren, begründen Narten et al. (2007) damit, dass dies zwar theoretisch keinen Nutzen bringe, aber praktische Vorteile habe. Es könnte dazu kommen, dass die Zufallszahlengeneratoren auf zwei verschiedenen Rechnern zum gleichen Ergebnis kommen, Duplicate Address Detection ausführen, die Adresse verwerfen und mit dem offensichtlich nicht zuverlässigen Generator wieder auf das gleiche Ergebnis kommen.
Es werden durch RFC 4941 außerdem Laufzeiten für die Gültigkeit dieser temporären Adressen vorgegeben. Sie sollen sieben Tage gültig sein und einen Tag präferiert benutzt werden. Das bedeutet, dass neue ausgehende Verbindungen die neue Adresse benutzen sollen und eine Verbindung sieben Tage bestehen bleiben kann. Nach einem Tag wird wieder eine neue Adresse gebildet und neue Verbindungen sollen nun diese Adresse benutzen und so weiter. (vgl. Narten et al. 2007, S. 11 ff.)

Ein wichtiger Punkt bei der Privacy Extension ist, dass die erstellten Adressen zusätzlich zu der SLAAC-Adresse existieren. Für ausgehende Verbindungen wird immer die temporäre Adresse benutzt, weil diese präferiert verwendet wird. Allerdings können eingehende Verbindungen sowohl über die temporäre Adresse als auch die SLAAC-Adresse stattfinden. Das führt dazu, dass ein Netzwerkscanner mit einem Verfahren zum Finden von Stateless Address Autoconfiguration Adressen einen Rechner trotz Privacy Extension entdecken kann. (vgl. Gont und Chown 2013, S. 7 f.)
Der Suchraum von Privacy-Adressen ist 2^{63} (nicht 2^{64}, da der Wert von Bit 6 des Interface Identifiers bekannt ist). Dieser ist damit fast so groß wie der Interface Identifier, weswegen das Suchen nach diesen Adressen (zumindest bei heutigen Netzwerkübertragungsgeschwindigkeiten) sehr lange dauert und in der Praxis, vor allem wegen der kurzen Adressgültigkeit, nicht durchführbar ist.
Neuere Versionen von Microsoft Windows bilden auch zufällige Adressen; allerdings wird anders als von RFC 4941 vorgeschlagen, nur einmalig der zufällige Interface Identifier gebildet und immer wiederverwendet. Nach diesen Privacy-Adressen kann in der Praxis auch nicht gesucht werden. Da sie sich jedoch nicht ändern, können Hosts anhand von Log-Dateien (siehe Kapitel 3.3) gefunden werden. (vgl. Gont und Chown 2013, S. 8)

2.3 Nmap

2.3.1 Über Nmap

Die erste Version von Nmap, abgeleitet von Network Mapper, wurde am 1. September 1997 in einem Magazin veröffentlicht. Das Ziel der Entwicklung von Nmap war, diverse Portscanner aus einem fragmentierten Bereich in einem Programm mit einheitlicher Oberfläche und effizienter Implementierung der Scantechniken zu vereinigen. Das Projekt ist unter der GPLv2, einer Open-Source-Lizenz, verfügbar und ist laut eigenen Angaben heute der weltweit beliebteste Netzwerksicherheitsscanner. (vgl. Lyon 2009, S. 19, 613 ff.)
Zu den Funktionen von Nmap gehören Host-Erkennung (Ping-Scanning), Port-Scanning, Versionserkennung von Diensten und Betriebssystemerkennung von entfernten Rechnern. Außerdem stehen viele Erweiterungen für spezielle Zwecke in einem Plugin-System zur Verfügung – der sogenannten Nmap Scripting Engine (NSE). (vgl. Lyon 2009, S. 6 ff.)
Am 29. November 2012 ist mit Nmap 6.25 die momentan aktuelle Version (Stand 10. Juli 2013) erschienen. Als Grundlage sämtlicher Beispiele wird in dieser Arbeit jene Version unter Debian GNU/Linux Jessie verwendet.

2.3.2 IPv6-Unterstützung

Seit dem Erscheinen der Version 3.10ALPHA1 am 28. August 2002 unterstützt Nmap grundlegend IPv6 (vgl. Lyon 2009, S. 57). Die vollständige Unterstützung folgte mit der Veröffentlichung von Version 6 am 21. Mai 2012.
Es sind seitdem viele Features, die vorher nur für IPv4 angeboten wurden, benutzbar. Dazu zählen das Versenden von rohen IPv6-Paketen (womit die diversen Scanmethoden, siehe Abschnitt 2.3.3, auch bei IPv6 möglich werden) und das Erkennen des Betriebssystems vom Zielrechner. (vgl. Nmap 2012a)
Um Nmap im IPv6-Modus zu betreiben, muss lediglich der Kommandozeilenparameter *-6* übergeben werden. *nmap -6 -PE -sn 2001:db8::1* würde Nmap dazu veranlassen, einen Internet Control Message Protocol version 6 (ICMPv6) Echo-Request (*-PE*) an 2001:db8::1 zu senden und die Erreichbarkeit anzuzeigen. *-sn* sorgt dafür, dass kein anschließender Portscan folgt. (vgl. Nmap 2011)
Trotz der angeblich vollständigen Unterstützung von IPv6 ist es nicht möglich, IPv6-Adressbereiche, zum Beispiel in der Form 2001:db8::/112 oder 2001:db8::0-ffff, anzugeben. Dies wurde ursprünglich damit begründet, dass es selten nützlich sei. (vgl. Lyon 2009, S. 93)

2.3.3 Scanmethoden

Nmap unterstützt eine Vielzahl von Scanmethoden zur Host-Erkennung. Die „Ping-Scan-Techniken“, die für entfernte IPv6-Netze implementiert sind (dies wurde durch Aufrufen von Nmap mit den verschiedenen Kommandozeilenoptionen beziehungsweise Durchsehen des Quelltextes herausgefunden), werden in Abbildung 4 kurz zusammengefasst. Welche dieser Methoden angewendet werden sollte, hängt von der Firewall (beziehungsweise Paketfilter) des Zielnetzwerkes oder Hosts ab. Manche dieser Verbindungsversuche können von Firewalls blockiert werden.

Scanmethode	**Schalter**	**Beschreibung**
TCP-SYN-Ping	*-PS*	Es wird ein Transmission Control Protocol (TCP) Paket mit SYN-Flag an das Zielsystem an Port 80 (wobei dieser konfigurierbar ist und mehrere Ports angegeben werden können) gesendet. Wenn die Gegenseite antwortet, entweder durch RST- (Port geschlossen) oder SYN/ACK-TCP-Paket (Port offen, weswegen eine Verbindung von der Gegenseite aufgebaut wird), dann ist der Rechner erreichbar.

TCP-ACK-Ping	*-PA*	Dieser Scantyp funktioniert ähnlich wie der TCP-SYN-Ping, außer dass statt eines SYN-Flags das ACK-Flag gesetzt wird. Dem Zielsystem erscheint dies als Bestätigung bei einer bisher bestehenden Verbindung, die allerdings bisher nicht besteht. Es sollte darauf mit einem RST-TCP-Paket geantwortet werden, wodurch die Existenz des Systems festgestellt wird. Diese Methode ist sinnvoll, wenn ein Paketfilter des Zielnetzwerkes SYN-TCP-Pakete für geschlossene Ports verwirft.
UDP-Ping	*-PU*	Ein leeres User Datagram Protocol (UDP) Paket wird an Port 31338 (konfigurierbar) des Zielrechners geschickt. Dieser Port wird selten genutzt, woraufhin ein aktiver Host zumeist mittels Internet Control Message Protocol (ICMP)-Paket antwortet, dass der Port nicht erreichbar ist. (Dienste hinter geöffneten Ports hingegen ignorieren zumeist leere Pakete.) Andere ICMP-Antworten oder keine Antwort werden als Nichterreichbarkeit gewertet.
SCTP-Init-Ping	*-PY*	Nmap versendet einen Stream Control Transmission Protocol (SCTP) Init-Chunk an Port 80 des Zielrechners (wobei dieser konfigurierbar ist und mehrere Ports angegeben werden können). Wenn mit ABORT (Port geschlossen) oder INIT-ACK (Port offen, weiter verfahren beim SCTP-Vier-Wege-Handschlag) geantwortet wird, ist der Rechner erreichbar. (vgl. Nmap 2011)
ICMP-Ping	*-PE*	Versenden eines ICMP-Echo-Requests – bei Empfang eines ICMP-Echo-Replys ist das Ziel erreichbar.

IP-Protokoll-Ping	*-PO*	Bei diesem Verfahren wird die Protokollnummer im IP-Header gesetzt und daraufhin das Paket an das Zielsystem gesendet. Für die Protokolle ICMP, Internet Group Management Protocol (IGMP), TCP, UDP und SCTP (vgl. Nmap 2011) werden dabei weitere protokollspezifische Felder in deren Headern gesetzt – bei anderen Protokollen wird nichts weiter als der IP-Header gesendet. Das Zielsystem ist erreichbar, falls es eine Antwort auf die Protokolle mit weiteren Headern gibt oder alternativ das im IP-Header angegebene Protokoll nicht unterstützt wird und mittels ICMP-Protokoll nicht erreichbar geantwortet wird. (Standardmäßig werden nur ICMP-, IGMP- und IP-in-IP-Pakete versendet, aber dies ist konfigurierbar.)

Abbildung 4: IPv6-Scanmethoden von Nmap (vgl. Lyon 2009, S. 110 ff.)

Die Performanz von Scans hat bei Nmap eine hohe Geltung, allerdings ist der Stellenwert von Genauigkeit noch größer. Dazu verfügt Nmap Algorithmen zur Überlaststeuerung und Erkennung von Paketverlusten, was dazu führt, dass Pakete langsamer hintereinander oder erneut gesendet werden. Trotzdem ist es möglich, sehr viele Pakete in kurzer Zeit zu versenden, was einfachere Netzwerkscanner ohne diese Algorithmen zum Beispiel tun. Mit *--min-rate* kann die Minimalzahl der zu versendenen Pakete in einer Sekunde definiert werden und mit *--max-retries* kann das erneute Versenden eines Paketes verhindert werden. Allerdings ist davon abzuraten, zu aggressive Werte zu setzen, da das dazu führen könnte, dass beim nächstgelegenen Router 99% der Pakete verworfen werden.
Nmap verfährt bei einem Scan im Normalfall so, dass von einer hohen Latenzzeit und Paketverlusten ausgegangen wird. Während des Scans werden die immer weiter eintreffenden statistischen Daten ausgewertet und die Geschwindigkeit des Vorgangs dynamisch angepasst. Falls Kenntnisse über das Zielnetzwerk (beziehungsweise den Weg dorthin) bestehen, können auch im Vorfeld Parameter optimiert werden, um die Lernphase zu verkürzen. (vgl. Lyon 2009, S. 212)
Ein einfacher Weg, die Scanzeit zu optimieren, ist, sogenannte Templates für die Zeiteinstellung zu verwenden. Jene setzen gleichzeitig viele Parameter, die die Scan-

zeit beeinflussen – viele der dynamischen Optimierungen können dabei trotzdem noch verwendet werden. Die Angabe eines Templates geschieht durch die Angabe von *-T* zusammen mit einer Nummer von 0 bis 5, wobei der Standard *-T3* ist. Die Templates 0 und 1 verlangsamen den Scan, um etwaige Intrusion Detection Systems (IDSs) zu umgehen, und 2 wird eingesetzt, um die verbrauchte Bandbreite und Ressourcen auf dem Zielrechner zu sparen. Template 4 und 5 gehen von schnellen und zuverlässigen Netzwerken aus, wobei *-T5* die aggressivsten Werte setzt. (vgl. Lyon 2009, S. 222 f.)

2.3.4 Erweiterbarkeit

Nmap ist ein Open-Source-Projekt, weswegen der Quelltext des Programmes, welcher zum Großteil in der Programmiersprache C++ geschrieben ist, frei zur Verfügung steht. Die GPLv2, als Lizenz des Projekts, erlaubt des Weiteren die Modifikation des Quelltextes, wodurch das Projekt grundlegend modifizierbar ist (vgl. Lyon 2009, S. 613 ff.). Beziehbar sind die Quellen von veröffentlichten Versionen unter `http://nmap.org/dist/` und der aktuelle Stand der Entwicklerversion über ein Subversion Repository unter `https://svn.nmap.org/nmap`.

Es gibt außerdem mit der NSE ein Plugin-System, womit es Benutzern möglich ist, durch Lua-Scripte Nmap zu erweitern (vgl. Lyon 2009, S. 313 ff.). Lua ist eine Erweiterungsprogrammiersprache und in C geschrieben. Sie ist dafür gedacht, Programme um eine mächtige, leichtgewichtige und einbettbare Scriptsprache zu ergänzen (vgl. Ierusalimschy et al. 2013).

Die Nmap Scripting Engine hatte das Ziel, folgende Aufgaben zu erledigen:

- Netzwerkerkennung
- Ausgefeiltere Versionserkennung
- Verwundbarkeitserkennung
- Backdoor-Erkennung
- Ausnutzung von Schwachstellen

Aber dies soll keine Beschränkung sein, da davon ausgegangen wird, dass noch weitere Anwendungen dafür gefunden werden können. (vgl. Lyon 2009, S. 313 ff.)

Diese Scripte bestehen aus drei Teilen: Der Kopf, die Regel und der Mechanismus. Im Kopf befinden sich Meta-Informationen (vgl. Lyon 2009, S. 319, 379 f.):

- **description**: Was tut das Programm?
- **usage**, **output** und **args**: Kommentare, wie ein möglicher Programmaufruf funktioniert, wie die Ausgabe aussieht und welche Argumente das Programm versteht.
- **author**: Liste der Autoren.

- **license**: Die Lizenz des Scriptes.
- **categories**: Zu welchen Kategorien ein Script gehört. (Zum Beispiel lässt sich Nmap mit *--script=malware* aufrufen, womit alle Scripte der Kategorie malware aufgerufen werden, um zu überprüfen, ob ein Ziel durch Schadsoftware befallen ist.)

Die Regel bestimmt, wann ein Script aufgerufen wird. Ein Beispiel dafür ist die sogenannte Port-Regel: Wenn Nmap feststellt, dass ein (festgelegter) Port auf einem Zielsystem offen ist, dann wird das Script aufgerufen.
Der Mechanismus bestimmt die Funktionalität des Scriptes – es ist der Programmablauf, der durchgeführt wird, wenn die Regel zutrifft. (vgl. Lyon 2009, S. 324, 380 f.)

Eine weitere Funktion, die Nmap anbietet, ist die Übergabe einer Liste von Hosts (die gescannt werden sollen) mittels des Kommandozeilenparameters *-iL*. Die Liste kann entweder ein Dateiname sein oder von der Standardeingabe mit „-“ als Argument eingelesen werden. Die Anforderungen an die Liste sind, dass die Hosts durch Leerzeichen, Tabs oder Zeilenumbrüche getrennt sein müssen und dass das Format der Listenelemente von Nmap unterstützt wird – das sind alle Formate, die auch auf der Kommandozeile akzeptiert werden (vgl. Lyon 2009, S. 93). Streng genommen ist es keine generelle Erweiterungsmöglichkeit, jedoch hat ein solches Verfahren in dieser Arbeit Relevanz, da ein externes Programm IPv6-Adressen in eine Datei schreiben kann (welche daraufhin von Nmap ausgelesen wird) oder diese mit einer Pipe an Nmap weiterleiten kann.

2.4 ICMPv6-Echo-Requests

Um im weiteren Verlauf der Arbeit Aussagen über die Laufzeit der verschiedenen IPv6-Adresssuchverfahren treffen zu können, wird an dieser Stelle eine Scanmethode zur Hosterkennung (ICMPv6-Echo-Requests) genauer betrachtet. Aufgrund dieser wird die Berechnung für die theoretische Laufzeit angefertigt.
Bei einem Echo-Request wird ein ICMPv6-Paket an den Zielrechner adressiert. Falls dieser mit einem ICMPv6-Echo-Reply antwortet, ist er erreichbar. Der Aufbau eines ICMPv6-Paketes ist in Abbildung 5 dargestellt. Es wird davon ausgegangen, dass das Paket vom Sender über Ethernet verschickt wird.
Der Type lautet beim Echo-Request 128 und der Code 0. Identifier und Sequence Number werden dazu benutzt, Echo-Replys (Type 129) zuzuordnen, können aber auch 0 sein. Data ist optional. (vgl. Conta et al. 2006, S. 13 f.)
Manchmal werden im Data-Feld Werte gesetzt, die auf den Hersteller des IPv6-Stacks oder das Ping-Tool Rückschlüsse zulassen (vgl. Hagen 2009, S. 97). Bei Nmap werden, nach eigenem Test, keine Werte gesetzt. Data fließt somit nicht in

die Größenberechnung des Paketes mit ein.
Insgesamt ist das Paket 74 Byte groß (26 Byte Ethernet Frame, wobei dieser größer sein kann; 40 Byte IPv6-Header; 8 Byte ICMPv6-Header). Da der Daten-Teil des Ethernet Frames (IPv6-Header und ICMPv6-Header) zusammen 48 Byte beträgt, muss das Pad-Feld nicht genutzt werden. Jenes wird dazu gebraucht, um das Daten-Feld auf eine Minimalgröße von 46 Byte aufzufüllen. Das Minimum ist dafür gedacht, intakte von defekten Ethernet Frames zu unterscheiden. (vgl. Tanenbaum und Wetherall 2011, S. 300 ff.)
Da allerdings zwischen jedem gesendeten Ethernet Frame eine kurze Zeit gewartet werden muss (das sogenannte interPacketGap, auch als Interframe Gap bezeichnet), ist dies in die Berechnung mit einzubeziehen. Für die in dieser Arbeit verwendeten Geschwindigkeiten von 100 Mbit/s und 1000 Mbit/s, als Grundlage zur theoretisch minimalen Laufzeit, beträgt der Wert des interPacketGaps mindestens 12 Byte. (vgl. IEEE-SA 2008, S. 581 f.)
Daher nimmt das Versenden eines ICMPv6-Paketes (mindestens) 86 Byte in Anspruch.

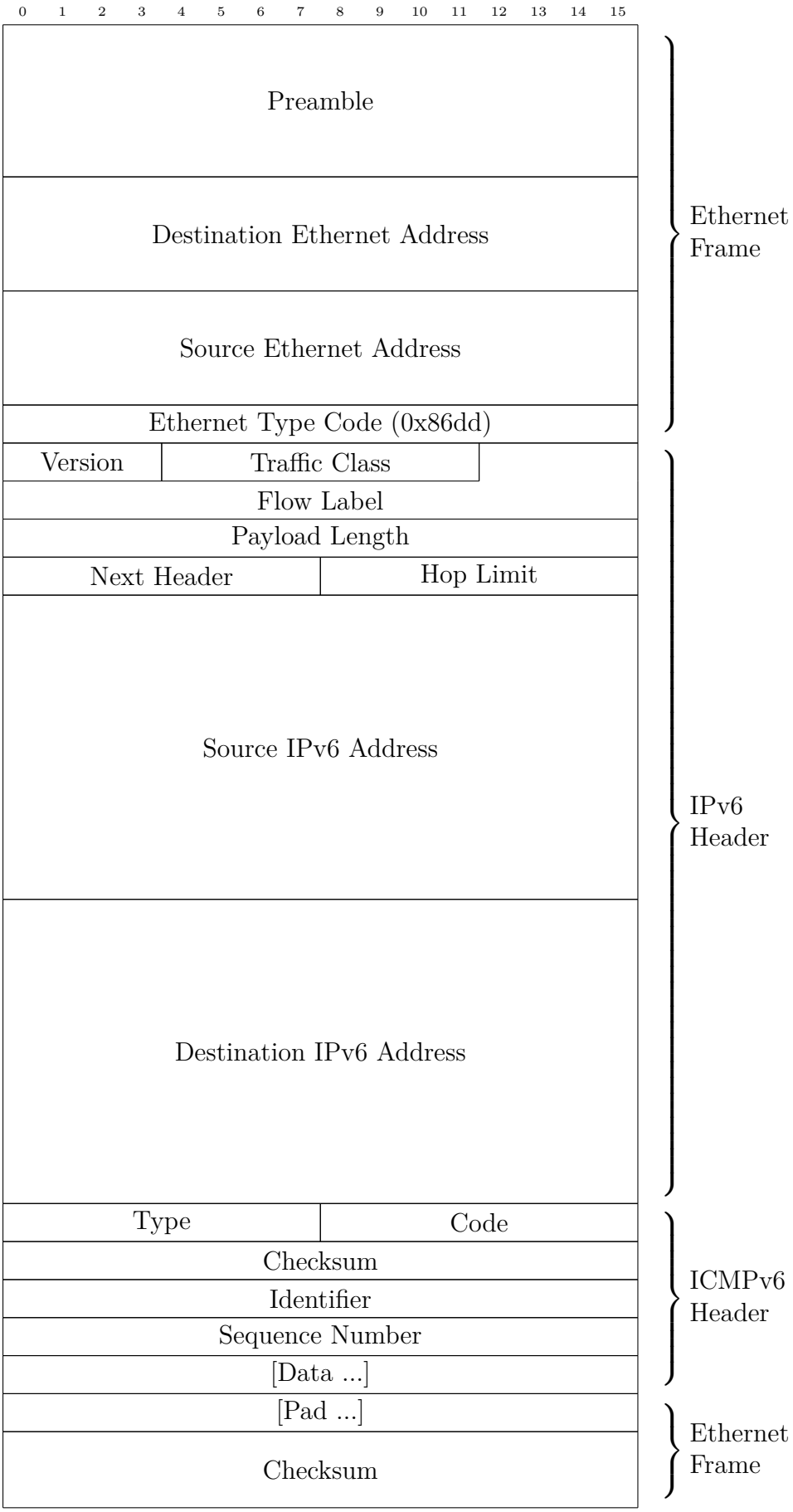

Abbildung 5: ICMPv6-Paket (nach Deering und Hinden (1998), Crawford (1998), Conta et al. (2006), Tanenbaum und Wetherall (2011))

2.5 Sicherheitsaspekte bei Entdeckung einer IP

Es gibt allgemein keine besonderen Sicherheitsbedenken, wenn ein Host durch ein Adresssuchverfahren entdeckt wird. Zumindest bei IPv4 ist es aufgrund des kleineren Adressraumes möglich, schnell Hosts in Subnetzen zu finden. Die Entdeckung impliziert lediglich, dass weitere Scanmethoden angewandt werden können. Dazu zählen Portscans, um laufende Serverdienste zu finden – bösartige Angreifer könnten diese bei verwundbaren Versionen kompromittieren. Allerdings waren bisher bei IPv4 viele Hosts aufgrund der Adressknappheit nur über Network Address Translation (NAT) mit dem Internet verbunden, weswegen sie (bis auf eventuelle Portweiterleitungen) gar nicht von außen gefunden werden konnten. Diese Hosts könnten in Zukunft mit einer IPv6-Adresse direkt über das Internet adressiert werden.
Letztendlich muss jeder Administrator eines Netzwerkes selbst entscheiden, ob eine Gefahrenlage durch die Entdeckung von Rechnern in seinem Netzwerk existiert. Abwehrmaßnahmen gegen Adresssuchverfahren werden in Kapitel 6 behandelt, wobei diese nur eingeschränkt bei Servern mit öffentlich angebotenen Serverdiensten anwendbar sind.

3 IPv6-Adresssuchverfahren

3.1 Präfixe

Die Internet Assigned Numbers Authority (IANA) ist verantwortlich für die weltweite Koordination von IPv6-Adressen. Die Verteilung der Adressblöcke geschieht hierarchisch zunächst an eine Regional Internet Registry (RIR), beispielsweise die Réseaux IP Européens Network Coordination Centre (RIPE NCC) für Europa, den Mittleren Osten und Zentralasien. Von dort werden sie weiter verteilt über eine National Internet Registry (NIR) an eine Local Internet Registry (LIR) und dann an den Internet Service Provider (ISP). (vgl. IANA 2012)
Falls eine IPv6-Adresse vorliegt (z.B. 2001:638:408::1) und das zugehörige vergebene IPv6-Netz und dessen Inhaber gesucht werden, könnte folgendermaßen vorgegangen werden: Die IANA listet die vergebenen Adressblöcke unter `http://www.iana.org/assignments/ipv6-unicast-address-assignments/` auf. Dort ist herauszufinden, dass 2001:0600::/23 an die RIPE NCC vergeben wurde. Die RIPE NCC (wie auch andere RIRs) bietet eine Suchfunktion für Adressen unter `https://apps.db.ripe.net/search/query.html` an. Das vergebene Netz ist nach dem Ergebnis der Suche 2001:0638:0408::/48 und wird von der „Fachhochschule Bonn-Rhein-Sieg“ betrieben. Es kann zusätzlich bei der RIPE NCC das zu einem Namen (z.B. Universität, Unternehmen, ...) zugehörige IPv6-Netzwerk über die Volltextsuche erlangt werden.
Der hier relevante Teil der Datenbank lässt sich außerdem als gepackte Textdatei unter `ftp://ftp.ripe.net/ripe/dbase/split/ripe.db.inet6num.gz` beziehen.
Die Abfrage von vergebenen IPv6-Netzen sagt aber nichts darüber aus, ob jene auch benutzt werden. Einen Anhaltspunkt bietet das Border Gateway Protocol (BGP), in dem bestehende Netzwerke von Routern annonciert werden. Unter `https://www.vyncke.org/ipv6status/prefixes.php` werden zum Beispiel solche BGP-Daten aufbereitet und zeigen, ob Traffic von einem bestimmten Netzwerk bereits gesehen worden ist.

3.2 Domain Name System

3.2.1 Hosts mit DNS-Eintrag

Jeder Host mit (öffentlichem) Eintrag im Domain Name System (DNS) kann potenziell gefunden werden, da der Domainname auf die IPv6-Adresse auflösbar ist. Eine Möglichkeit, IPv6-Adressen über DNS zu finden, ist, mittels eines Wörterbuches häufig genutzte Subdomainnamen, wie etwa www und mail, aufzulösen. (vgl. Chown 2008, S. 6; Gont und Chown 2013, S. 18)
Nmap enthält für diesen Zweck ein Script namens `dns-brute`; ein eigenständiges

Programm ist beispielsweise `dnsmap` (`https://code.google.com/p/dnsmap/`). Beide Tools beinhalten eine Liste von häufig genutzten Subdomainnamen (die erweiter- beziehungsweise veränderbar ist) und gehen diese bei einer angegebenen Domain durch.
Durch das Herausfinden dieser Adressen kann eventuell ein Muster gefunden werden, wie die Adressierung bei diesem Netz aussieht, um weitere Hosts zu finden. (vgl. Chown 2008, S. 6; Gont und Chown 2013, S. 18)

3.2.2 Zonentransfer

Mittels Zonentransfer können DNS-Einträge (beziehungsweise die DNS-Zone) von einem Nameserver auf diverse andere gespiegelt werden. Zonentransfers müssen unter anderem bei Änderungen von DNS-Einträgen stattfinden. (vgl. Mockapetris 1987, S. 28)
Falls der Zugang zu diesen Daten nicht eingeschränkt wird, können Dritte sämtliche DNS-Informationen einer Zone erlangen. Es ist deswegen bei IPv4 eine bewährte Praxis, unautorisierte Zonentransfers zu unterbinden – bei IPv6 ist dies ebenfalls ratsam. (vgl. Chown 2008, S. 6)
Es gibt diverse Webseiten, die einen Test auf Zonentransfer anbieten. Auch Nmap besitzt mit `dns-zone-transfer` ein Script für diese Aufgabe.

3.2.3 Reverse Mapping

Eine weitere Methode, über DNS Hosts (und auch Subnetze) zu finden, wurde im Jahr 2012 von Peter van Dijk vorgestellt. Sie funktioniert durch das Entlanggehen von ip6-arpa-Zonen und dem Suchen von DNS-PTR-Records (die einer IP-Adresse einen Fully Qualified Domain Name (FQDN) zuordnen).
Van Dijk erläutert die Methode folgendermaßen (eine grafische Aufarbeitung ist unter Abbildung 6 zu finden):

- Es wird innerhalb 2001:DB8:80::/48 gesucht – die DNS-Zone dazu lautet 0.8.0.0.8.b.d.0.1.0.0.2.ip6.arpa..
- In dieser Zone wird nach den PTR-Records von hexadezimal 0 bis f gesucht (0.0.8.0.0.8.b.d.0.1.0.0.2.ip6.arpa. bis f.0.8.0.0.8.b.d.0.1.0.0.2.ip6.arpa.) und ein DNS-Server danach gefragt.
  - Wenn ein PTR-Record nicht existiert, dann antwortet der Server mit NXDOMAIN. Das bedeutet üblicherweise auch, dass unter diesem PTR-Record keine weiteren PTR-Records vorhanden sind, womit in diesem Baum nicht weiter gesucht werden muss.
  - NOERROR heißt hingegen, dass weitere Einträge in einem Baum existieren, woraufhin dort weiter gesucht wird.

Durch dieses Verfahren verringert sich der Suchraum im DNS deutlich.
Ob es funktioniert, hängt vom DNS-Server ab – während `BIND` und `NSD` mit NXDOMAIN und NOERROR antworten, so tun dies `tinydns` und `PowerDNS` (in den meisten Konfigurationen) nicht. (vgl. van Dijk 2012)
Van Dijk bietet eine Implementierung als eigenständiges Programm unter `https://github.com/habbie/ip6-arpa-scan/` an. Nmap enthält ein Script namens `dns-ip6-arpa-scan` für diesen Zweck.

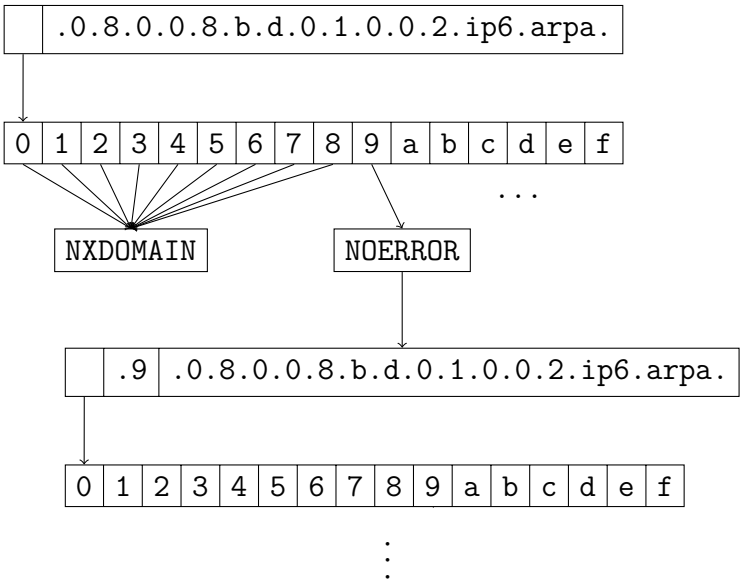

Abbildung 6: DNS Reverse Mapping

3.3 Ausgehende Verbindungen

Alle von einem Netzwerk ausgehenden Verbindungen können die IP-Adresse eines Hosts preisgeben. So ist zum Beispiel davon auszugehen, dass viele Webserver aufgerufene Webseiten, inklusive der IP-Adresse des Aufrufers, in Log-Dateien speichern – dies gilt auch für andere Protokolle (vgl. Chown 2008, S. 6). Insbesondere ist dies problematisch bei Diensten, bei denen die IP-Adresse nicht nur von einem Administrator des jeweiligen Dienstes einsehbar ist. Unter anderem sind dabei Mailinglisten zu nennen, bei denen Mail-Header-Informationen (zum Beispiel „Received from:“) ausgelesen werden können, um die Absenderadresse herauszufinden. (vgl. Gont und Chown 2013, S. 21)
Durch Erlangung dieser Adressen könnte von Dritten auf Muster der Adressvergabe geschlossen werden.

3.4 Bereichsscan

Eine häufige Form von IPv6-Adressen in einem Bereich sind solche, bei denen im Interface Identifier die zwei niederwertigsten Bytes gesetzt sind und alle anderen Bytes den Wert 0 haben – zum Beispiel 2001:db8::1, 2001:db8::2 und so weiter. Eine weitere gebräuchliche Art dieser Adressen hat die vier niederwertigsten Bytes gesetzt. So ist es außerdem nach Gont und Chown (2013) nicht unüblich, dass die zweitletzten zwei niederwertigen Bytes einen Wert von 0-255 haben, während in den niederwertigsten zwei Bytes alle Werte genutzt werden. Solche Adressen werden „Low-Byte-Adressen“ genannt.
Im schlechtesten Fall beträgt der Suchraum 2^{24} Adressen, wobei sich viele Hosts schon unter den ersten 2^{16} oder sogar 2^{8} Adressen befinden. (vgl. Gont und Chown 2013, S. 10)
Auch ist es möglich, dass Hosts IPv6-Adressen in anderen Bereichen benutzen. Dadurch würde sich der Suchraum auf bis zu 2^{64} vergrößern. Die Verkleinerung dieses Raumes ist möglich, wenn Muster aus den (in Kapitel 3.2 behandelten) DNS-Verfahren oder sonstigen Methoden gefunden werden können.
IPv6-Adressen in einem Bereich kommen entweder durch statische Konfiguration von Adressen oder durch DHCPv6 (entweder durch sequentielle Vergabe beziehungsweise kleinen Adresspool oder durch feste Vergabe) zustande.
Eine Implementierung zur Suche nach Low-Byte-Adressen gibt es von Fernando Gont, einem der Autoren des RFC-Draft „Network Reconnaissance in IPv6 Networks“ in der Applikation `scan6` aus der Programmsammlung `IPv6 Toolkit` unter `http://www.si6networks.com/tools/ipv6toolkit/`. Im Prinzip kann mit jedem Netzwerkscanner, der die Angabe von Adressbereichen unterstützt, nach diesen Adressen gesucht werden.
In den Untersuchungen der Verteilung von IPv6-Adressen in Kapitel 3.9 wird außerdem der Begriff Byte-Muster verwendet. Jener wird in der Literatur nicht definiert, stammt aber vermutlich aus dem Programm `addr6` aus dem `IPv6 Toolkit` zur Auswertung der Verteilung von IPv6-Adresslisten. Im Quelltext des Programms ist zu erkennen, dass diese Adressen mindestens 3 Byte im Interface Identifier enthalten, die auf 0 gesetzt sind, wobei die drei einzelnen Bytes nicht direkt aneinander liegen müssen. Aufgrund dieser im Quelltext festgelegten Definition der Byte-Muster-Adressen können auch die in Kapitel 3.8 behandelten wortreichen Adressen in diese Kategorie fallen.

3.5 SLAAC-Identifier

Die Bildung von Identifiern, die bei der Stateless Address Autoconfiguration aus EUI-48 entstehen, wurde in Kapitel 2.2.2 behandelt.

Nun ist die Frage, wie groß der Suchraum für diese Adressen ist und welche Werte diese annehmen. Eindeutig definiert ist bereits, dass beim Interface Identifier Bit 6 auf 1 und 0xfffe in der Mitte gesetzt ist. Davor ist die OUI zu finden und danach der Manufacturer-Selected Extension Identifier. Da die Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA) eine Liste veröffentlicht (siehe `https://standards.ieee.org/develop/regauth/oui/oui.txt`), welche OUIs vergeben wurden, verkleinert sich der Suchraum weiter. Momentan (Stand 5. August 2013) sind 18212 OUIs vergeben.
Weil der Manufacturer-Selected Extension Identifier 2^{24} Werte annehmen kann ist der Suchraum aktuell $18212 * 2^{24}$ Adressen groß. Falls weiterhin nicht mehr benutzte OUIs aus der Liste entfernt werden könnten, würde der Suchraum noch ein wenig geringer werden. (Allerdings scheint nirgends darüber Buch geführt zu werden.)
Auch ist es möglich, dass Wissen darüber besteht, von welchem Hersteller Geräte in einem Zielnetzwerk eingesetzt werden oder ob eine SLAAC-Adresse über DNS beziehungsweise Online-Archive bezogen wurde. In beiden Fällen verringert sich der Suchraum eventuell drastisch, falls viele baugleiche Geräte einer Charge angeschafft wurden, denn diese haben oft aufeinander folgende MAC-Adressen.
Eine weitere Suche, bei der der Adressraum eingegrenzt werden kann, ist die nach IPv6-Adressen innerhalb von Virtualisierungssoftware. So werden bei VirtualBox automatisch generierte MAC-Adressen mit der OUI 08:00:27 gebildet. Bei VMWare ESX Servern ist die OUI bei manueller Konfiguration (der MAC-Adresse) 00:05:59 – alle weiteren Bits der MAC-Adresse können nur von 0x000000 bis 0x3fffff gesetzt werden, damit es keine Konflikte mit anderen VMWare-Produkten gibt. Die MAC-Adresse kann außerdem mit der OUI 00:05:59 automatisch generiert werden, wobei die höchstwertigen 2 Byte nach dem OUI auf die zwei niederwertigsten Bytes der primären IPv4-Adresse des VMWare Konsolenbetriebssystems gesetzt werden. Das verbleibende Byte wird aus einem Hash des Namens der Konfiguration der virtuellen Maschine erstellt. Falls also nach virtuellen Maschinen gesucht wird und eventuell deren IPv4-Adresse bekannt ist, kann der Suchraum auf bis zu 2^8 Adressen gesenkt werden. (vgl. Gont und Chown 2013, S. 6 f.)
Eine Implementierung ist auch bei diesem Verfahren im Programm `scan6` zu finden.

3.6 Eingebettete IPv4-Adressen

Eine weitere Form von IPv6-Adressen, nach denen gesucht werden kann, sind jene mit eingebetteten IPv4-Adressen. Sie werden entweder durch statische Konfiguration oder DHCPv6 gesetzt.
Die gebräuchlichste Art solcher Adressen ist jene, bei der die vier niederwertigsten Bytes im Interface Identifier auf den Wert einer IPv4-Adresse gesetzt werden und die anderen 4 Byte 0 sind. Eine grafische Darstellung dieser Form, inklusive eines

Beispiels mit dem Prefix 2001:db8::/64 und der IPv4-Adresse 192.0.2.1, ist in Abbildung 7 zu finden. (Die genannten Adressen werden auch bei weiteren Beispielen benutzt.)

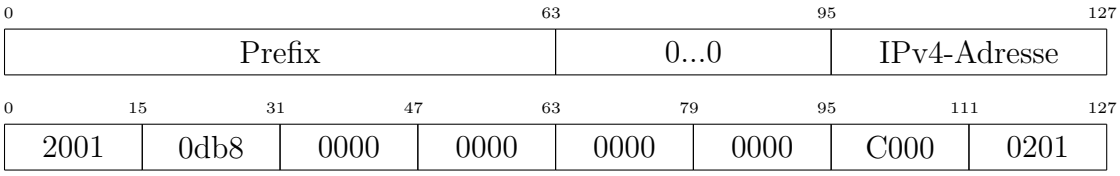

Abbildung 7: Eingebettete IPv4-Adresse (IPv4-Compatible-IPv6-Adressenstil) (nach Hinden und Deering 2006, S. 10)

Die Schreibweise (neben der Hexadezimalen) ist folgende: 2001:db8::192.0.2.1 . Diese Adresse hat weitestgehend das Format der in RFC 4291 behandelten IPv4-Compatible-IPv6-Adressen, weswegen im weiteren Verlauf dieser Arbeit (und der Implementierung) von „IPv4-Compatible-IPv6-Adressenstil“ gesprochen wird.
Auch üblich ist das Einfügen der vier, durch Punkte getrennten Dezimalzahlen einer IPv4-Adresse in die vier 2 Byte Hexadezimalblöcke des Interface Identifiers – zum Beispiel 2001:db8::192:0:2:1. (vgl. Gont und Chown 2013, S. 10)
Die Dezimalzahlen werden jedoch nicht in Hexadezimal umgerechnet, weshalb sich der ursprüngliche Wert der Adressteile ändert. Siehe dazu Abbildung 8. In dieser Arbeit wird diese Form „IPv4-Inserted-IPv6-Adressenstil“ genannt.

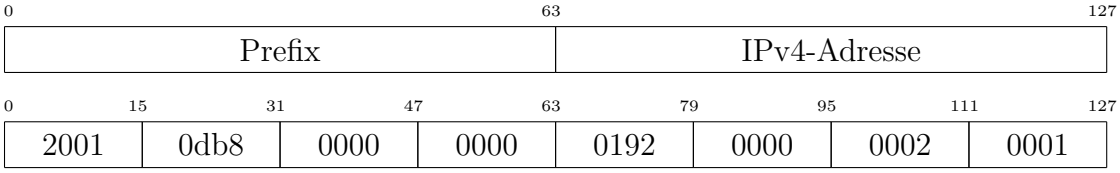

Abbildung 8: Eingebettete IPv4-Adresse (IPv4-Inserted-IPv6-Adressenstil) (nach Gont und Chown 2013, S. 10)

Eine weitere mögliche Form, die jedoch nicht im vorliegenden RFC Draft behandelt wird, könnten eingebettete IPv4-Adressen im Stil der IPv4-Mapped-IPv6-Adressen (siehe RFC 4291) sein. Das Format unterscheidet sich nur wenig vom IPv4-Compatible-IPv6-Adressenstil – statt die höchstwertigen 2 Byte des Interface Identifiers auf 0x00000000 zu setzen, wird der Wert 0x0000ffff verwendet. Abbildung 9 zeigt diese Adressen.

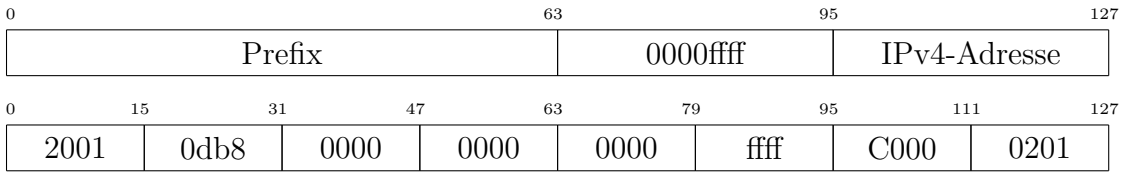

Abbildung 9: Eingebettete IPv4-Adresse (IPv4-Mapped-IPv6-Adressenstil) (nach Hinden und Deering 2006, S. 10 f.)

Der Suchraum für diese Adressart ist so groß wie die Anzahl der IPv4-Adressen, nach denen gesucht wird. Da davon auszugehen ist, dass Administratoren die IPv4-Adressen ihres zugewiesenen IPv4-Subnetzes benutzen, ist der Suchraum höchstens so groß wie dieses. (vgl. Gont und Chown 2013, S. 10)
In den meisten Fällen wird es weniger als ein IPv4-/8-Netzwerk sein, da von diesen nur wenige an Endabnehmer abgegeben wurden – also ist der Suchraum in den meisten Fällen höchstens 2^{24} Adressen groß. Schlimmstenfalls beinhaltet er aber $3 * 2^{32}$ Adressen, da nach allen möglichen IPv4-Adressen und allen drei Verfahren gesucht werden müsste.
Eine Implementierung ist auch in diesem Fall im Programm `scan6` zu finden.

3.7 Dienst-Port-Adressen

Hosts können auch die TCP-/UDP-Portnummer des angebotenen Dienstes in der IPv6-Adresse inkludiert haben. Der Port, zum Beispiel 80, wird dabei in den zwei niederwertigsten beziehungsweise den zwei zweitniederwertigsten Bytes untergebracht – das jeweils andere Bytefeld wird dabei als Zähler verwendet, falls mehrere Webserver betrieben werden. Solche Adressen sehen folgendermaßen aus: 2001:db8::0:80, 2001:db8::1:80 oder 2001:db8::80:0, 2001:db8::80:1 und so weiter. Dabei wird vermutet, dass der Zähler bis 255 verwendet wird. Der benutzte Port könnte außerdem als Hexadezimalzahl angegeben werden.
Wenn nach einem Maximum von 20 häufig genutzten Ports (sowohl in Dezimal- als auch Hexadezimalschreibweise) bei zwei Zählern bis 255 (entweder in den letzten oder zweitletzten 2 Byte) gesucht wird, beträgt der Suchraum $40 * 2^9$ beziehungsweise $10 * 2^{11}$. (vgl. Gont und Chown 2013, S. 10 f.)
Bei Dienst-Port-Adressen ist allerdings anzunehmen, dass sie (zumindest für öffentliche Dienste) über das DNS publiziert werden, weswegen sie auch über DNS-Suchverfahren zu finden sein können.
Eine Implementierung befindet sich im Programm `scan6`. Hosts, die dieses Schema einsetzen, sind zum Beispiel:

- **ns.netbsd.org**: 2001:4f8:3:7::53 (Port 53 = DNS)
- **mail.netbsd.org**: 2001:4f8:3:7::25 (Port 25 = SMTP)

- **www.freebsd.org**: 2001:1900:2254:206a::50:0 (Port 0x50 = 80 = HTTP)
- **mx1.freebsd.org**: 2001:1900:2254:206a::19:1 (Port 0x19 = 25 = SMTP)
- **mx2.freebsd.org**: 2001:1900:2254:206a::19:2 (Port 0x19 = 25 = SMTP)

3.8 Wortreiche Adressen

Hexadezimalzahlen erlauben es, Worte darzustellen – dies wird auch bei IPv6-Adressen getan. Ein geläufiges Beispiel sind die Wörter 0xdead und 0xbeef (die schon früher als Rückgabewerte in Computerprogrammen benutzt wurden), woraus sich eine Adresse bilden ließe: 2001:db8::dead:beef.
Diese Adressen können durch Wörterbuchangriffe herausgefunden werden. Der Suchraum ist somit nicht genau anzugeben und hängt von vielen Faktoren ab. Laut Fernando Gont wird die Scanmethode auch im Programm `scan6` unterstützt. (vgl. Gont und Chown 2013, S. 11, 35)
Allerdings wird in dem Programm nicht anhand eines Wörterbuches gesucht, sondern wie bei einem Bereichsscan, wo sich die Hexwörter schon in der Adresse befinden. Internetseiten, die dieses Schema einsetzen, sind zum Beispiel:

- **www.taz.de**: 2001:67c:13c::7a2:de (7a2:de = taz.de)
- **www.facebook.com**: 2a03:2880:f010:301:face:b00c:0:1 (face:b00c = facebook)

3.9 Benutzung der Adressen

Es gibt zwei Untersuchungen aus dem Jahr 2013 darüber, welche Interface Identifier benutzt werden – eine bezieht sich auf Server, die andere auf Clients.
Für Server wurden die DNS-Records von den in „Alexa Top 1.000.000 Sites“ (`http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`) aufgelisteten und den Teilnehmern des World-IPv6-Launch-Day (`http://www.worldipv6launch.org/participants/`) verzeichneten Webseiten ausgewertet. Daraufhin wurden für jede Domain DNS-Records abgefragt – AAAA, NS und MX. Für die Hosts, die in NS- und MX-Records gelistet waren, wurde danach auch der AAAA-Record ermittelt. (vgl. Gont 2013, S. 21)
Das Ergebnis ist in den folgenden drei Abbildungen zu sehen:

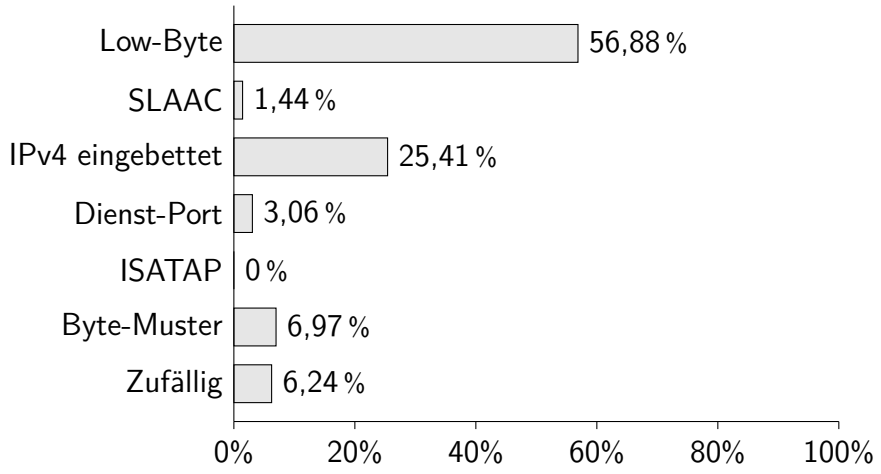

Abbildung 10: Verteilung IPv6-Webserver-Adressen (vgl. Gont und Chown 2013, S. 12)

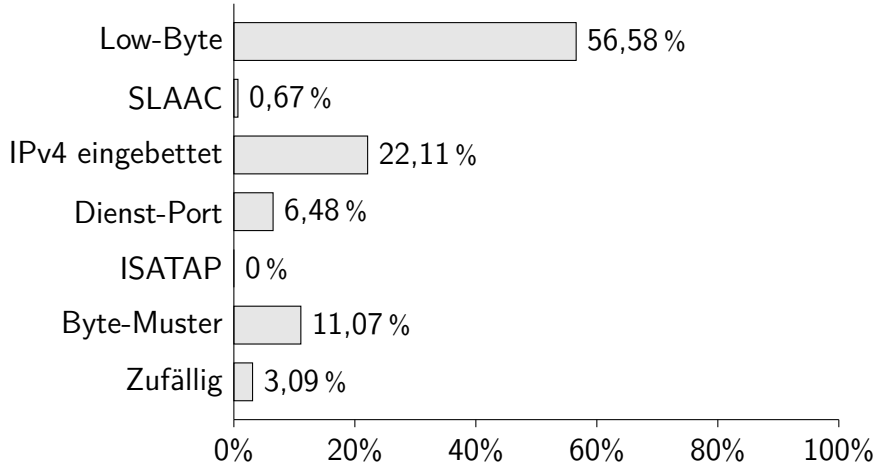

Abbildung 11: Verteilung IPv6-Nameserver-Adressen (vgl. Gont und Chown 2013, S. 12)

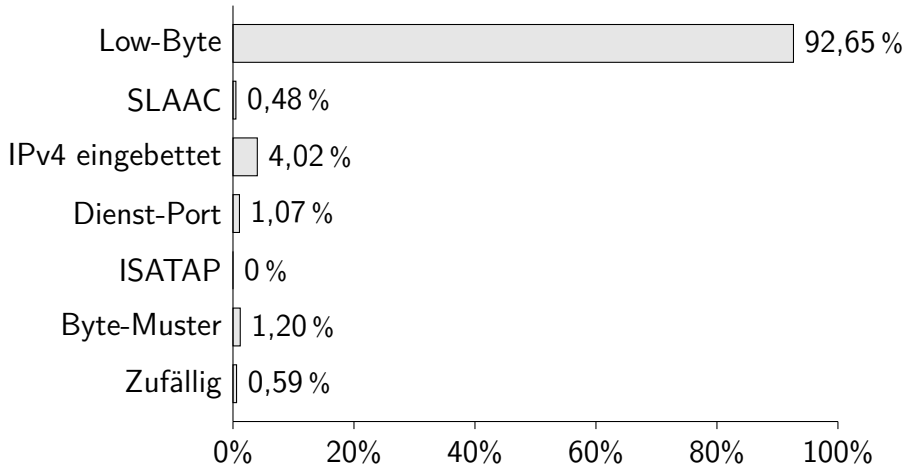

Abbildung 12: Verteilung IPv6-Mailserver-Adressen (vgl. Gont und Chown 2013, S. 13)

Anhand der Ergebnisse ist zu entnehmen, dass bei Servern nur wenige SLAAC und ISATAP (ein in RFC 5214 definierter IPv4 nach IPv6 Übergangsmechanismus) Interface Identifier verwendet werden. Alle anderen Arten scheinen (je nach Dienst etwas verschieden) benutzt zu werden, wobei vor allem Low-Byte und eingebettete IPv4-Adressen einen Großteil ausmachen.
Für Clients wurden die Webserver-Logs der Seite `http://www.internetsociety.org/` ausgewertet. Es flossen 50000 unterschiedliche IPv6-Adressen in die Berechnung ein (vgl. Ford 2013):

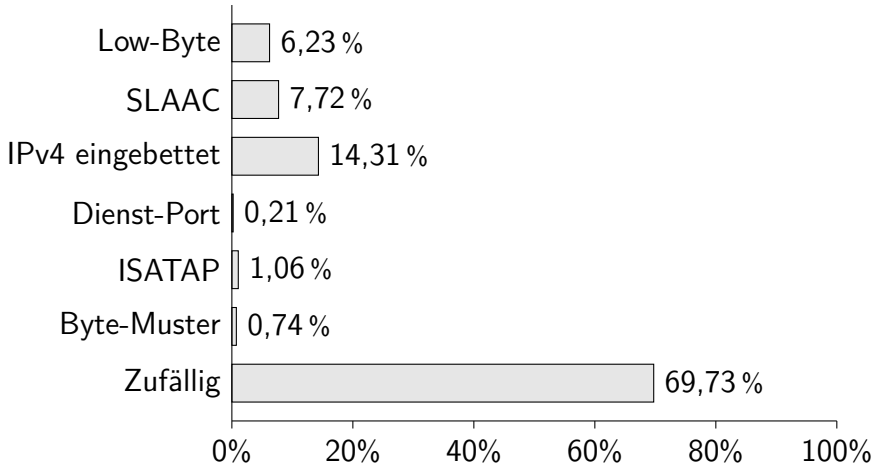

Abbildung 13: Verteilung IPv6-Client-Adressen (vgl. Gont und Chown 2013, S. 14)

Hier fällt auf, dass mit knapp 70% sehr viele Clients die Internetseite mit einer zufälligen Adresse (die eine Privacy-Adresse sein könnte) aufgerufen haben. Einen

kleineren Teil machen hingegen eingebettete IPv4-Adressen, SLAAC und Low-Byte-Adressen aus. Der Rest kann vernachlässigt werden.

3.10 Übersicht der Adressen

Abschließend werden die Informationen der vorherigen Kapitel in einer Tabelle zusammengefasst, damit ein rascher Überblick möglich ist:

Adressart	Statisch	DHCPv6	SLAAC	Client	Server	Scanbarkeit
Low-Byte	✓	✓	✘	✓	✓	✓
SLAAC	✓	✓	✓	✓	✓	✓
IPv4 eingebettet	✓	✓	✘	✓	✓	✓
Dienst-Port	✓	✓	✘	✘	✓	✓
Wortreich	✓	✓	✘	-	✓	-
Privacy/zufällig	✓	✓	✓	✓	✓	✘

Abbildung 14: Übersicht aller Adressverfahren

Über wortreiche Adressen liegen nicht genug Informationen vor, um zu entscheiden, ob diese bei Clients verwendet werden – es könnte durchaus sein, da sie in den Untersuchungen zur Verteilung von IPv6-Adressen in die Kategorie „Byte-Muster“ oder „zufällig“ fallen können. Inwieweit sinnvollerweise nach wortreichen Adressen gesucht werden kann, wird noch in Kapitel 5.5 diskutiert, wobei vorwegzunehmen ist, dass keine schlussendliche Aussage zu treffen ist.

4 Evaluierung des Implementierungsweges für Nmap

4.1 Nmap Scripting Engine

Mit Hilfe der NSE ist es recht einfach, Nmap zu erweitern – deswegen wurde sie schließlich entwickelt.
Es gibt zwei Wege, wie das grundsätzliche Scannen von IPv6-Adressen in der Nmap Scripting Engine realisiert werden kann:

1. Zusammenbauen von IPv6-Paketen im Script und Verschicken dieser Pakete. Dies wird in den Scripten `targets-ipv6-multicast-*.nse` benutzt, um spezielle Multicastpakete zu erstellen.
2. Benutzen der `target`-Bibliothek, um IPv6-Adressen zu der Nmap-Scan-Queue hinzuzufügen, was dazu führt, dass Nmap sich um den Scanvorgang kümmert. Dies geschieht durch den Aufruf von *target.add("2001:db8::1")* im Script (vgl. Nmap 2012b).

Ersteres hat den Vorteil, dass es (je nach Implementierung) sehr schnell sein kann. Der Nachteil hingegen ist, dass die diversen Scanmethoden von Nmap (siehe Kapitel 2.3.3) nicht benutzt werden und diese bei Bedarf selbst implementiert werden müssten.
Der zweite Weg ermöglicht es, sämtliche Scanmethoden zu benutzen. Allerdings gibt es auch hier einen Nachteil: Das Hinzufügen von Hosts zur Queue kann in diesem Fall nur in der sogenannten prerule-Regel geschehen. Das bedeutet, dass Nmap erst nach dem Terminieren des Scripts (beziehungsweise aller aufgerufenen Scripte mit prerule-Regel) anfängt, die Scan-Queue abzuarbeiten – die Scanziele müssen also vorher hinzugefügt worden sein. Da die Speicherung der Ziele in einer Datenstruktur im Arbeitsspeicher des Computers stattfindet, wird diese bei vielen Hosts sehr groß. Bei einem Test zeigte sich, dass 2^{24} Ziele fast 1,3 Gigabyte verbrauchen.
Ein Gespräch mit einem Nmap-Entwickler ergab, dass es in Zukunft möglich sein soll, für solche Zwecke Ziele während des Scanvorgangs von Nmap hinzuzufügen, was den Speicherverbrauch deutlich verringern sollte. Außerdem soll zukünftig unterstützt werden, IPv6-Adressen in Prefix-Längen-Notation anzugeben, wodurch die Scanziele aggregiert übergeben werden können. Auf der Nmap-TODO-Liste steht dazu folgendes:
„Add IPv6 subnet/pattern support like we offer for IPv4.
Obviously we can't go scanning a /48 in IPv6, but small subnets do make sense in some cases. For example, the VPS hosting company Linode assigns only one IPv6 address per user (unless they pay) and you can find many Linode machines by scanning certain /112's. And patterns might be useful because people assigned /64's might still put their machines at ::1, ::2, etc.“, (Nmap 2013). Nach der Veröffentlichung von Nmap 6.25 wurde dieses Feature am 19. Dezember 2012 in die

Entwicklerversion aufgenommen. Es könnte somit in der nächsten veröffentlichten Version von Nmap enthalten sein.
Um ein Script auszuführen, muss es in den Script-Ordner von Nmap kopiert werden und danach *nmap --script-updatedb* als privilegierter Benutzer aufgerufen werden, damit Nmap seine Script-Datenbank aktualisiert. Danach kann es mit *nmap --script=scriptname.nse* aufgerufen werden.

4.2 Nmap-Quelltext

Eine Implementierung der Adresssuchverfahren im Nmap-Quelltext könnte sinnvoll sein. So steht auf der Nmap-TODO-Liste, dass Scripte zur Entdeckung von IPv6-Hosts im lokalen Netzwerk in den Kern aufgenommen werden sollen:
„Move advanced IPv6 host discovery features from NSE into core Nmap. We'll probably add the functionality of targets-ipv6-multicast-invalid-dst, targets-ipv6-multicast-echo, and maybe targets-ipv6-multicast-slaac. - The idea is that Nmap does them automatically if it gets a large target specification and sees that it is local so can be multicast pinged.“, (Nmap 2013).
Allerdings ist es ohne gründliches (und deswegen langwieriges) Studium des Nmap-Quelltextes nicht möglich abzuschätzen, wie lange dies dauern wird – insbesondere könnten infrastrukturelle Änderungen am Quelltext notwendig sein. Solch eine Analyse könnte sich vereinfachen, nachdem die im Zitat erwähnten Scripte integriert wurden.
Auch ist unklar, ob den Entwicklern von Nmap der Patch, der bei der Implementierung im Quelltext entsteht, gut genug ist, sodass die Änderung übernommen wird. Im Zweifel müsste dann bei letztendlicher Nichtakzeptierung des Patches jeder, der diese Funktionen benutzen möchte, den Quelltext von Nmap inklusive Patch selbst kompilieren.

4.3 Externe Programme

Wie bereits in Kapitel 2.3.4 geschrieben, kann Nmap mittels des Kommandozeilenparameters *-iL* eine Liste von zu scannenden Hosts einlesen.
Eine einfache Methode, Nmap um die Adresssuchverfahren zu ergänzen, ist, ein externes Programm zu schreiben (wobei die Programmiersprache nicht relevant ist), welches die gewünschten IPv6-Adressen produziert. Die Adressen können dann entweder in eine Datei geschrieben oder mittels Pipe an Nmap übergeben werden. Der Nachteil einer Datei ist, dass diese bei vielen zu scannenden Hosts sehr groß wird (da Nmap noch keine IPv6-Prefix-Längen-Notation versteht, siehe Kapitel 4.1), weswegen die Ausgabe über die Standardausgabe zusammen mit einer Pipe genutzt werden sollte.
Bei einer (namenlosen) Pipe wird die Standardausgabe eines Prozesses (hier das

Adressgenerierprogramm) an die Standardeingabe eines anderen Prozesses (hier Nmap) geleitet. Neben dieser Funktionalität bietet eine Pipe zusätzlich eine Flusskontrolle. Eine Pipe kann nämlich nur eine bestimmte Menge von Daten (standardmäßig wenige Kilobyte bei unixoiden Systemen) aufnehmen, weswegen bei vollem Pipespeicher der Sender angehalten werden muss, bis der Empfänger wieder mindestens ein Byte ausgelesen hat. Bei leerer Pipe hingegen wird der Empfänger angehalten, bis wieder Daten in die Pipe geschrieben werden. (vgl. Wolf 2009, S. 276 f.)
Der Nachteil der Umsetzung der Adresssuchverfahren mit einem externen Programm ist, dass jenes niemals zusammen mit Nmap ausgeliefert werden wird. Falls das externe Programm außerdem an keiner sehr prominenten Stelle auffindbar ist, wird es vermutlich kaum beachtet beziehungsweise gefunden werden können und wird somit kaum genutzt werden.
Falls ein solches Programm gewünscht sein sollte, könnte mit relativ wenig Aufwand das Programm `scan6` im `IPv6 Toolkit` umgeschrieben werden, sodass es anstatt selber zu scannen die Adressen einfach ausgibt.

4.4 Auswahl des Implementierungsweges

In dieser Arbeit wird die Umsetzung aller Scanmethoden mit Hilfe der NSE gewählt. Die Gründe dafür sind folgende:

- Die Scripte lassen sich in einem vorhersehbaren Zeitraum erstellen. (Gleiches gilt für externe Programme.)
- Wenn die Scripte in den Augen der Entwickler gut genug sind, werden sie zusammen mit Nmap ausgeliefert – für den Nmap-Quelltext gelten vermutlich weit höhere Standards.
  - Selbst wenn die Scripte nicht mit aufgenommen werden, können sie im Nachhinein in den Script-Ordner von Nmap kopiert und genutzt werden, ohne den Quelltext patchen zu müssen. Das Problem des möglicherweise Nichtauffindens des Scripts im Internet bleibt dabei, wie bei externen Programmen, bestehen.

5 Implementierung ausgewählter Verfahren

5.1 Bereichsscan

5.1.1 Bewertung des Verfahrens

Low-Byte-Adressen werden sowohl bei Clients als auch bei Servern benutzt. Die Tatsache rechtfertigt allein schon die Erstellung eines Programmes zum Suchen dieser Adressen. Außerdem ist der Suchraum im schlechtesten Fall mit 2^{24} deutlich geringer als der gesamte Interface Identifier, was eine Suche in der Praxis ermöglichen kann. Das Suchen anderer Adressen aus einem Bereich ist ohne weitere Informationen über die Verteilung nicht möglich. Abhilfe könnten zum Beispiel durch DNS-Suchverfahren erlangte Adressen schaffen – im eigenen Netzwerk sollte hingegen die Adressverteilung bekannt sein.

5.1.2 Implementierung

Zum Scannen von IPv6-Adressen in einem Bereich wurde das Script `targets-ipv6-range.nse` geschrieben. „targets-ipv6-“ als Anfang des Scriptnamens wurde (wie auch bei allen anderen erstellten Programmen) gewählt, da es bereits Scripte bei Nmap (zum Scannen in lokalen IPv6-Netzen) gibt, die das gleiche Benennungschema aufgreifen.
Das Script versteht mit *targets-ipv6-range.net6* einen Übergabeparameter, mit dem ein oder mehrere IPv6-Netze angegeben werden können. Dabei werden die Angaben von Prefix-Längen-Notation (z.B. 2001:db::/112) oder Von-Nach-Notation (z.B. 2001:db::1-f:1-ff) unterstützt. (In folgenden Beschreibungen von Übergabeparametern wird der Scriptname weggelassen – also wird nur noch von *.net6* gesprochen. Beim Ausführen des Scripts ist es allerdings zwingend notwendig, den gesamten String als Parameter anzugeben!)
Der Aufruf des Scripts zum Erstellen von zu scannenden IPv6-Adressen im Netz 2001:db8::/127 sieht beispielsweise folgendermaßen aus:

```
# nmap -6 --script=targets-ipv6-range.nse --script-args \
'targets-ipv6-range.net6=2001:db8::/127'

Starting Nmap 6.25 ( http://nmap.org ) at [...]
Pre-scan script results:
| targets-ipv6-range:
|   IP: 2001:db8:0:0:0:0:0:0
|   IP: 2001:db8:0:0:0:0:0:1
|_  Use --script-args=newtargets to add the results as targets
WARNING: No targets were specified, so 0 hosts scanned.
```

```
Nmap done: 0 IP addresses (0 hosts up) scanned in 0.26 seconds
```

Wie die Ausgabe bereits verrät, muss zur Ausführung des Scans noch „*,newtargets*“ an die Argumente des Scripts angehängt werden – sonst werden nur die zu scannenden Adressen angezeigt. (Alternativ kann zur Anzeige der Adressen das Kommando *nmap -6 -sL –script=targets-ipv6-range.nse –script-args 'targets-ipv6-range.net6=2001:db8::/127,newtargets'* verwendet werden. Dabei werden die Adressen, im Gegensatz zum Aufruf ohne *newtargets*, an die Nmap-Scan-Queue übergeben, aber von Nmap durch *-sL* (List Scan) lediglich ausgegeben.)
Mehrere Netze können dem Script durch eine Lua-Tabelle übergeben werden. Die Tabelle wird dabei von { } umschlossen und die Werte werden jeweils durch ein Komma getrennt – zum Beispiel { 2001:db8::/127 , 2001:db8:99::0-1 }.
Zum Suchen nach 2^{24} Low-Byte-Adressen müsste also entweder 2001:db8::/104 oder 2001:db8::1-ff:1-ffff als Ziel angegeben werden.
Dieses Script ist wie die folgenden in der Nmap-Script-Kategorie „intrusive“ eingeordnet, da bei Angabe von vielen Zielen in einem entfernten Netzwerk jenes überbeziehungsweise ausgelastet werden könnte. Daraus folgt möglicherweise die Nichterreichbarkeit von Netzwerkdiensten.

5.2 Suche nach SLAAC-Identifiern

5.2.1 Bewertung des Verfahrens

Diese Adressen werden durch ein automatisches Verfahren erstellt und haben somit immer den gleichen Aufbau. Da sie sowohl bei Clients als auch bei Servern eingesetzt werden, kann die Suche sinnvoll sein. Falls die OUIs der Netzwerkkarten bekannt sind, könnte sich das Verfahren eignen, um vorhandene Rechner zu inventarisieren, die nur über einen oder mehrere Router erreichbar sind.

5.2.2 Implementierung

Zur Generierung von IPv6-Adressen mit SLAAC Interface Identifiern wurde das Script `targets-ipv6-slaac.nse` geschrieben. Mittels *.net6* können dem Script IPv6-Adressnetze in Von-Nach-Notation übergeben werden.
Die Angabe der OUIs, nach denen in den Interface Identifiern gesucht werden soll, kann in zwei Formen erfolgen – beide Formate können außerdem gleichzeitig verwendet werden. Zum einen direkt über das Argument *.ouicid*, wobei die Angabe einer oder mehrerer OUIs ohne Doppelpunkte in der OUI erfolgen muss. Zum anderen können mit *.ouicidname* ein oder mehrere Namen von OUI-Inhabern angegeben werden. Dabei werden dem Namen die OUI-Einträge aus der Datei /usr/share/nmap/nmap-mac-prefixes zugeordnet. Falls die Datei nicht existieren sollte oder eine andere Datei benutzt werden soll, dann kann jene mit *.ouicidfile* angegeben werden – das For-

mat muss dabei „OUI␣Name“ gefolgt von einem Zeilenumbruch sein. Eine aktuelle Zuordnung von OUIs zu Namen kann unter unixoiden Systemen folgendermaßen erstellt werden:

```
$ curl http://standards.ieee.org/develop/regauth/oui/oui.txt | sed \
-ne 's/^ *\([0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]\
[0-9A-Fa-f]\) *(base 16)\t\t\(.*\)$/\1 \2/p'
```

Die Zuordnung vom Namen zum OUI erfolgt intern mit Hilfe von Lua Patterns, weswegen es möglich ist, mit *.** auf alle OUI-Namen zu matchen. Weiterhin matcht in diesem Script Apple auf Apple.*, da es in der Liste der IEEE neben dem Eintrag „Apple“ noch mehrere Einträge wie „Apple, Inc.“ oder „Apple Computer“ gibt. Falls nur Apple gewünscht ist, dann ist Apple$ anzugeben.
Ein Aufruf des Programmes könnte zum Beispiel folgendermaßen aussehen:

```
# nmap -6 -n --script=targets-ipv6-slaac.nse --script-args \
'targets-ipv6-slaac.net6=2001:db8::1-2,\
targets-ipv6-slaac.ouicidname=Nvidia'

Starting Nmap 6.25 ( http://nmap.org ) at [...]
Pre-scan script results:
| targets-ipv6-slaac:
|   IP: 2001:db8:0:0:204:4bff:fe00:1
|   IP: 2001:db8:0:0:204:4bff:fe00:2
|_  Use --script-args=newtargets to add the results as targets
WARNING: No targets were specified, so 0 hosts scanned.
Nmap done: 0 IP addresses (0 hosts up) scanned in 0.41 seconds
```

5.3 Suche nach eingebetteten IPv4-Adressen

5.3.1 Bewertung des Verfahrens

Das Verfahren ist eindeutig definiert, allerdings ist der Suchraum im Worst-Case relativ groß. Da hingegen einfach herauszufinden sein sollte, welche IPv4-Subnetze verwendet werden (vor allem, wenn es sich um Geräte in der eigenen Firma handelt), kann dieser deutlich verkleinert werden, wodurch die Suche ermöglicht wird. Da Clients und Server solche Adressen benutzen, kann dieses Verfahren bei beiden angewendet werden.

5.3.2 Implementierung

Das Script `targets-ipv6-embedded-ipv4.nse` generiert IPv6-Adressen, in die IPv4-Adressen eingebettet sind. Dabei können sowohl mittels *.net6* IPv6-

Adressnetze als auch *.net4* IPv4-Adressnetze übergeben werden. Das Ergebnis sind alle möglichen Kombinationen von IPv6-Adressen mit eingebetteten IPv4-Adressen. Bei der Angabe von Netzen wird sowohl die Prefix-Längen- als auch die Von-Nach-Notation unterstützt.
Ein Aufruf des Scripts sieht beispielsweise folgendermaßen aus:

```
# nmap -6 --script=targets-ipv6-embedded-ipv4.nse --script-args \
'targets-ipv6-embedded-ipv4.net6=2001:db8::,\
targets-ipv6-embedded-ipv4.net4={192.0.2.0/30,198.51.100.1}'

Starting Nmap 6.25 ( http://nmap.org ) at [...]
Pre-scan script results:
| targets-ipv6-embedded-ipv4:
|   IP: 2001:db8:0:0::192.0.2.0
|   IP: 2001:db8:0:0::192.0.2.1
|   IP: 2001:db8:0:0::192.0.2.2
|   IP: 2001:db8:0:0::192.0.2.3
|   IP: 2001:db8:0:0::198.51.100.1
|_  Use --script-args=newtargets to add the results as targets
WARNING: No targets were specified, so 0 hosts scanned.
Nmap done: 0 IP addresses (0 hosts up) scanned in 1.67 seconds
```

Ohne weitere Angabe des Adressformats erstellt das Script Adressen im IPv4-Compatible-IPv6-Adressen-Stil. Durch den Parameter *.addressstyle* lassen sich darüberhinaus Adressen im IPv4-Inserted-IPv6-Adressen-Stil mit dem Wert *inserted* und IPv4-Mapped-IPv6-Adressen-Stil mit dem Wert *mapped* erstellen.
Falls nach allen möglichen eingebetteten IPv4-Adressen mit allen gegebenen Adressstilen gesucht wird, dann müsste neben der Angabe von 0.0.0.0/0 als Wert für die IPv4-Adressen das Script dreimal aufgerufen werden mit jeweils verschiedenem Wert für den Adressenstil.

5.4 Suche nach Dienst-Port-Adressen

5.4.1 Bewertung des Verfahrens

Der Suchraum dieses Verfahrens ist mit $10 * 2^{11}$ klein, weswegen die Suche auch bei langsamen Netzwerken durchführbar sein sollte – allerdings werden damit meistens nur Server gefunden werden. Es stellt sich des Weiteren die Frage, nach welchen eingebetteten Ports gesucht werden soll.
Eine Implementierung lohnt sich, weil das Verfahren eindeutig definiert ist und einige Dienste diese Adressen benutzen.

5.4.2 Implementierung

Das Generieren von Dienst-Port-Adressen übernimmt das Script `targets-ipv6-ports.nse`. Es funktioniert ähnlich wie `targets-ipv6-range.nse` (siehe Kapitel 5.1.2) und ist durch das Argument *.net6* lauffähig, wobei nur die Von-Nach-Notation unterstützt wird. Im Gegensatz zum Bereichsscan muss in angegebenen Adressen ein (oder mehrere) „x“ an die Stelle gesetzt werden, an der eine Portnummer eingefügt werden soll. Standardmäßig werden dann Adressen mit folgenden Ports generiert: 21, 22, 23, 25, 49, 53, 80, 110, 123, 179, 220, 389, 443, 547, 993, 995, 1194, 3306, 5060, 5061, 5432, 6446, 8080. Die Auswahl der Portnummern wurde vom Programm `scan6` aus dem `IPv6 Toolkit` übernommen.
Im Zuge der Generierung der Adressen wird die Portnummer im Normalfall sowohl in dezimaler (also zum Beispiel 80 für http) als auch in hexadezimaler Schreibweise (50 bei http) verwendet. Dieses Verhalten lässt sich durch den Übergabeparameter *.format* beeinflussen. Durch den Übergabewert *dec* wird nur die Dezimaldarstellung gewählt und *hex* sorgt für ausschließliche Verwendung der Hexadezimalwerte. Alle anderen Eingaben verwenden beide Werte.
Ein Aufruf des Scripts mit der Adresse 2001:db8::x und ausschließlicher Verwendung von Dezimalwerten sieht wie folgt aus:

```
# nmap -6 --script=targets-ipv6-ports.nse --script-args \
'targets-ipv6-ports.net6=2001:db8::x,targets-ipv6-ports.format=dec'

Starting Nmap 6.25 ( http://nmap.org ) at [...]
Pre-scan script results:
| targets-ipv6-ports:
|   IP: 2001:db8:0:0:0:0:0:21
|   IP: 2001:db8:0:0:0:0:0:22
|   IP: 2001:db8:0:0:0:0:0:23
|   IP: 2001:db8:0:0:0:0:0:25
|   IP: 2001:db8:0:0:0:0:0:49
|   IP: 2001:db8:0:0:0:0:0:53
|   IP: 2001:db8:0:0:0:0:0:80
|   IP: 2001:db8:0:0:0:0:0:110
|   IP: 2001:db8:0:0:0:0:0:123
|   IP: 2001:db8:0:0:0:0:0:179
|   IP: 2001:db8:0:0:0:0:0:220
|   IP: 2001:db8:0:0:0:0:0:389
|   IP: 2001:db8:0:0:0:0:0:443
|   IP: 2001:db8:0:0:0:0:0:547
```

```
|   IP: 2001:db8:0:0:0:0:0:993
|   IP: 2001:db8:0:0:0:0:0:995
|   IP: 2001:db8:0:0:0:0:0:1194
|   IP: 2001:db8:0:0:0:0:0:3306
|   IP: 2001:db8:0:0:0:0:0:5060
|   IP: 2001:db8:0:0:0:0:0:5061
|   IP: 2001:db8:0:0:0:0:0:5432
|   IP: 2001:db8:0:0:0:0:0:6446
|   IP: 2001:db8:0:0:0:0:0:8080
|_  Use --script-args=newtargets to add the results as targets
WARNING: No targets were specified, so 0 hosts scanned.
Nmap done: 0 IP addresses (0 hosts up) scanned in 0.31 seconds
```

Es ist außerdem möglich, weitere Ports mit *.addports* der Liste hinzuzufügen oder eine andere Liste mit *.ports* zu übergeben. Dies geschieht erneut mit einer Lua-Tabelle, in der die Zahlen in Dezimalschreibweise angegeben werden müssen.
Die Suche nach allen Dienst-Port-Adressen des Suchraumes (der in Kapitel 3.7 definiert wurde) erfolgt mit der Angabe der Adressen in der folgenden Form: { 2001:db8::1-ff:x , 2001:db8::x:1-ff }.

5.5 Suche nach wortreichen Adressen

5.5.1 Bewertung des Verfahrens

Dieses Verfahren ist im Gegensatz zu den anderen Verfahren nicht eindeutig definiert. Es könnte natürlich immer nach ::dead:beef in Adressen gesucht werden, allerdings gibt es weitaus mehr Möglichkeiten, die in der Praxis Verwendung finden. Auch ist unklar, ob es allgemein üblich ist, in solchen Adressen Zähler einzusetzen, falls mehr als ein Host mit einer bestimmten Wortkombination versehen werden soll. Bevor ein derartiges Verfahren sinnvoll umgesetzt werden kann, müssten zunächst Untersuchungen stattfinden mit dem Ziel, Muster herauszufinden, wie solche Adressen mehrheitlich aussehen.
Ein Ansatz, wie nach wortreichen Adressen gesucht werden könnte, wird im folgenden Kapitel behandelt.

5.5.2 Mögliche Implementierung

Das Suchen von wortreichen Adressen könnte anhand eines Wörterbuchs durchgeführt werden. Mangels Untersuchungen der benutzten Worte in wortreichen Adressen muss dieses allerdings zunächst erstellt werden. Dazu wird ein Alphabet benötigt – die Benutzung der Hexadezimalzahlen a bis f ist dabei selbstverständlich. Zusätzlich können auch Zahlen aufgrund ihrer gewissen Ähnlichkeit zu bestimmten

Buchstaben verwendet werden. Die folgende Abbildung zeigt, wie eine Zuordnung von Hexadezimalzahlen zu Buchstaben aussehen könnte:

0	→	o
1	→	i
2	→	z
5	→	s
6	→	g
7	→	t oder l
a	→	a
b	→	b
c	→	c oder k
d	→	d
e	→	e
f	→	f

Abbildung 15: Zuordnung Hexadezimalzahl → Buchstabe (nach Hardee (2011), mit eigenen Ergänzungen)

Es kann nun in Wörterbüchern (zum Beispiel einer Sprache) nach Wörtern gesucht werden, die durch diese Zuordnung gebildet werden können. Ein Befehl zum Finden dieser Wörter (die durch Zeilenumbrüche getrennt sein müssen) unter unixoiden Systemen könnte folgendermaßen aussehen:

```
sed -ne 's/^\([0-9AaBbCcDEedFfGgIiKkLlOoSsTtZz]\
[0-9AaBbCcDEedFfGgIiKkOoLlSsTtZz]*\)$/\1/p' WORTDATEI |\
sed -e '/^.*[iI].*[lL].*\|.*[lL].*[iI].*$/d' -e \
'/^.*[cC].*[kK].*\|.*[kK].*[cC].*$/d' |\
tr 'ABCDEFGgIiKkLlOoSsTtZz' 'abcdef6611cc1100557722'
```

Die weitere Verfahrensweise ist allerdings schwierig, da nicht genug Erkenntnisse bestehen, wie wortreiche Adressen im Allgemeinen aufgebaut sind. Es wäre zum Beispiel möglich, alle Kombinationen der Wörter bis 16 Zeichen zu bilden. Zunächst könnten auch alle Wörter von vorne mit Nullen aufgefüllt werden, bis sie 4, 8, 12, oder 16 Buchstaben lang sind, damit Wörter in den Adressen durch Doppelpunkte getrennt werden. Sonst wird zum Beispiel aus „a dead beef“ adea:dbee:f, statt a:dead:beef.
In die Liste der zu konkatenierenden Wörter müsste außerdem 0000 aufgenommen werden, da ein Interface Identifier auch nur ::beef, oder ::beef:, sein könnte – auch ein Zähler könnte sinnvoll sein.
Wie bereits beschrieben, ist bei all den offenen Fragen ohne weitere Forschung (die in einer anderen Arbeit aufgegriffen werden müsste) keine sinnvolle Implementierung möglich.

5.6 Laufzeit der Adresssuchverfahren

Nach den Beschreibungen der Verfahren und der Implementierung stellt sich die Frage, ob die Adresssuchverfahren überhaupt in einer annehmbaren Laufzeit durchzuführen sind.
Um dies zu analysieren, wurden zwei Wege verfolgt:

1. Berechnung der minimalen theoretischen Laufzeit bei 100 Mbit/s und 1000 Mbit/s Ethernet (beim Protokoll ICMPv6).
2. Messung der Laufzeit beim Netzwerkscanner Nmap (beim Protokoll ICMPv6).

Für die minimale theoretische Laufzeit wird der in Kapitel 2.4 ermittelte Wert der minimalen Größe eines ICMPv6-Paketes (86 Byte) genommen und hochgerechnet, wobei mögliche Latenzen und Antworten nicht in die Berechnung mit einfließen. Dadurch ist das Ergebnis allerdings nur eine theoretische untere Schranke bei Vernachlässigung sämtlicher Parameter, die in der Praxis Relevanz haben. So müssten unter anderem alle Rechner auf dem Weg der Pakete diese bei der jeweiligen Geschwindigkeit verarbeiten können, ohne sie zu verwerfen und weiterhin garantieren, dass jedes Paket ankommt.
Ein praxisnäheres Ergebnis soll durch den Testlauf mit Nmap erzielt werden. Für den Versuch wurden auf einem Computer mit „Intel Celeron 550 (2.00GHz)“-Prozessor zwei VirtualBox-Instanzen betrieben. Verbunden hatte diese ein internes VirtualBox-Netz mit „Intel PRO/1000 MT Server“-Netzwerkadaptern. Beide Instanzen wurden mit Debian GNU/Linux testing (jessie) betrieben, wobei eine Instanz mittels Nmap scannte und die andere jeweils fünf IPv6-Adressen, verteilt über den jeweiligen Adressbereich, konfiguriert hatte. Mit den konfigurierten Adressen sollte überprüft werden, dass es auch Antworten auf ICMPv6-Echo-Request gibt und nicht nur massenhaft Pakete versendet werden.
Da es nicht relevant sein sollte, in welchem Adressbereich gesucht wird, sondern darauf ankommt, wie viele Adressen gesucht werden, wurden alle Tests mit dem Script `targets-ipv6-range` in dem Bereich der niederwertigsten Bytes durchgeführt. (Es ist zusätzlich nicht relevant, mit welchem Script die Adressen erstellt werden, da alle Scripte aufgrund ihres ähnlichen Aufbaus eine ähnliche Laufzeit haben.)
Für drei Adressbereichsgrößen (2^8, $10 * 2^{11}$, 2^{16}) wurden jeweils zehn Tests durchlaufen (die Rohdaten sind im Anhang unter Nmap-Messung zu finden) und da es keine großen Ausreißer gab, das Arithmetische Mittel bestimmt. Für größere Adressbereiche wurde aufgrund zu langer Testzeiten der Wert des 2^{16}-Tests hochgerechnet.
Die Testreihen wurden mit zwei verschiedenen Nmap-Templates durchgeführt, *-T3* (Nmap-Default) und *-T5* (für sehr schnelle Netzwerke). Bei allen Tests wurden die auf der anderen VirtualBox-Instanz konfigurierten Adressen gefunden, weswegen noch weitere Parameter beim Nmap-Scan optimiert werden könnten. Allerdings ist

bei Anwendungsfällen zur Suche von IPv6-Adressen nicht generell davon auszugehen, dass die Laufzeit noch signifikant reduziert werden kann – zumindest birgt es das Risiko, dass Hosts nicht entdeckt werden, weswegen weitere Optimierungen in dieser Arbeit nicht weiter behandelt werden.

Adressen	Verfahren	100 Mbit/s	1000 Mbit/s	Nmap -T3	Nmap -T5
2^8	-	1,76 µs	0,18 µs	3,55 s	1,90 s
$10 * 2^{11}$	Dienst-Port	0,14 s	0,01 s	9,63 min	3,98 min
2^{16}	-	0,45 s	0,05 s	22,18 min	12,26 min
2^{24}	Low-Byte	1,92 min	0,19 min	94,65 h †	52,33 h †
2^{32}	-	8,20 h	0,82 h	2,77 y †	1,53 y †
$3 * 2^{32}$	IPv4 eingebettet	24,62 h	2,46 h	8,30 y †	4,59 y †
$18212 * 2^{24}$	SLAAC	24,33 d	2,43 d	196,77 y †	108,81 y †
2^{64}	-	$4 * 10^6$ y	$4 * 10^5$ y	10^{10} y †	$7 * 10^9$ y †

†: hochgerechnet

Abbildung 16: Worst-Case-Laufzeiten der Adresssuchverfahren

Bei den Tests fällt sofort auf, dass die Testwerte sich deutlich von den theoretisch minimalen Werten unterscheiden. Dies ist damit zu begründen, dass Nmap Überlast versucht zu verhindern und Pakete unter Umständen mehrfach erneut sendet. Auf einem schnelleren Rechner würde sich das Ergebnis nicht drastisch verändern, da während des Tests Prozessor und Arbeitsspeicher zu weiten Teilen nicht mehr als 50% ausgelastet waren. Es ist weiterhin anzunehmen, dass Tests in realen Netzwerken deutlich länger dauern, da sich noch Router zwischen den Rechnern befinden können und die Latenz größer ist.
Dieser Test zeigt außerdem, dass viele der Verfahren, zumindest im Worst-Case, momentan nicht ohne sehr lange Laufzeiten durchführbar sind. Es kommt somit in vielen Fällen darauf an, den Suchraum zu verkleinern, indem zum Beispiel Kenntnisse über den verwendeten IPv4-Adressbereich bestehen oder welche OUIs von Netzwerkkarten verwendet werden – natürlich wäre es auch sinnvoll, im Vorhinein zu wissen, welche Verfahren verwendet werden, um nicht alle durchführen zu müssen.
Hinzu kommt, dass alle Verfahren nur ein /64-Netz durchsuchen. Allerdings werden in vielen Fällen selbst an Privatleute größere Netzbereiche vergeben, damit mehrere /64-Subnetze erstellt werden können. Falls also nicht bekannt ist, in welchem dieser Subnetze gesucht werden soll, dann muss die Suche bei allen Subnetzen stattfinden

(was die in dieser Arbeit erstellten Scripte erlauben). Falls zum Beispiel in einem /48-Netz gesucht werden soll, dann müssen die Laufzeiten jedes Verfahrens mit 2^{16} multipliziert werden, womit die Scans bei größeren Subnetzen in der Praxis kaum noch durchführbar sind.

5.7 Zukünftige Anpassung der Implementierung

5.7.1 Nmap-Veränderungen

Wie bereits in Kapitel 4.1 geschrieben, wird Nmap in einer neuen Version die Angabe von IPv6-Netzen in Prefix-Längen-Notation erlauben – ob eine Angabe auch in Von-Nach-Notation möglich sein wird, ist noch unklar. Damit wäre das Script `targets-ipv6-range.nse` obsolet. Alle anderen Scripte würden von dem neuen Feature profitieren, da nicht mehr jede Adresse einzeln an Nmap übergeben werden muss, was aktuell zu einer hohen Auslastung des Arbeitsspeichers führen kann. Um dieses neue Feature benutzen zu können, müssten die hier angefertigten Implementierungen angepasst werden.
Es ist zu hoffen, dass auch die Von-Nach-Notation Unterstützung findet, da manche Adressangaben in der Prefix-Längen-Notation wenig Sinn ergeben. Die Angabe von ::1-ff:80 als Dienst-Port-Adresse ergibt zum Beispiel einen Sinn. ::80/104 macht dies hingegen nicht beziehungsweise sagt sogar etwas anderes aus.
Falls die Von-Nach-Notation nicht unterstützt werden wird (wofür es langfristig rational keinen Grund geben sollte), könnte es durch eine neue Nmap-Regel zukünftig möglich sein, während des Scans neue Adressen zur Scan-Queue hinzuzufügen. Dies wird auch den Arbeitsspeicherverbrauch einschränken können.

5.7.2 Gültigkeit jetziger Evaluierung

Die in dieser Arbeit erfolgte Evaluation über den Implementierungsweg der Adresssuchverfahren für Nmap bleibt bestehen. Dabei bleibt abzuwarten, ob und wie andere Scripte in den Quelltext von Nmap übernommen werden.

6 Abwehr der Adresssuche

6.1 Allgemein

Insofern eine Gefährdungslage durch die Entdeckung eines Hosts vorliegen sollte (wobei dann die grundlegende Frage besteht, ob der jeweilige Host überhaupt mit dem Internet verbunden sein muss), kann über etwaige Abwehrmaßnahmen nachgedacht werden.
Generell ist zu sagen, dass Rechner, die einen öffentlichen DNS-Eintrag haben, nicht in einem Subnetz versteckt werden können. Clients brauchen nicht unbedingt einen öffentlichen DNS-Eintrag und Server, die öffentlich Dienste anbieten, sind naturgemäß auffindbar. Falls Serverdienste nur von einem kleinen Personenkreis genutzt werden sollen (zum Beispiel Mitarbeiter eines Unternehmens), könnte die Verwendung eines Virtual Private Networks (VPNs) diskutiert werden (vgl. Lyon 2009, S. 444). Damit würde nur ein öffentlicher Server als Relaisstation für Dienste auffindbar sein.

6.2 Adresswahl

Auf der Hand liegt, dass Hosts, die eines in dieser Arbeit beschriebenen Adressschemen verwenden, über kurz oder lang gefunden werden können – dies gilt insbesondere, wenn der Suchraum weiter eingeschränkt werden kann durch Kenntnisse über das Netzwerk. Eine einfache Methode, die Entdeckung eines Hosts weiter hinauszuzögern, ist offensichtlich: Andere Adressen verwenden.
Damit erübrigt sich SLAAC als Adressvergabeverfahren, da die entstandenen Adressen das beschriebene feste Schema haben. Es bleiben statische oder Stateful-DHCPv6-Adressen. Jedoch ist zu beachten, dass die vergebenen Adressen nicht aus einem Bereich stammen, da sonst durch einen entdeckten Host auf alle anderen geschlossen werden kann. Bei DHCPv6 müsste dann entweder jedem Host eine feste Adresse zugeordnet werden (ohne einen Bereich zu verwenden) oder Adressen müssten zufällig aus dem Adresspool verteilt werden, wobei dies der DHCPv6-Server unterstützen muss.
Auch wenn die Adressvergabe ohne Schema erst einmal vor der Entdeckung bei eingehenden Scans schützt, so ist die Preisgabe der IPv6-Adresse bei ausgehenden Verbindungen immer noch möglich. Es ist also ratsam, für ausgehende Verbindungen temporäre Adressen zu benutzen, was DHCPv6 unterstützt. Bei statischer Konfiguration müsste hingegen ein eigenes Verfahren entwickelt werden, wie immer neue temporäre Adressen erstellt werden.
Aber all die Maßnahmen stellen keine garantierte Sicherheit dar, dass ein Host unentdeckt bleibt, da es sich nur um eine Verschleierung des Interface Identifiers für Angreifer handelt.

6.3 Paketfilter

Das Entdecken von Hosts durch Adresssuchverfahren von extern kann, zumindest bei Clients, durch einen Paketfilter (Firewall) auf einem Router verhindert werden. Es ist generell eine gute Idee, eine Firewall zu benutzen, die mit „Deny by default“ betrieben wird – statt nur verdächtigen Traffic zu blockieren, wird generell alles verweigert und für jede erwünschte Funktionalität eine Ausnahme geschaffen (vgl. Lyon 2009, S. 445). Da es bei Clients oft nur wenige oder gar keine Dienste gibt, die direkt von außerhalb des Netzwerkes erreichbar sein müssen, kann in vielen Fällen sämtlicher eingehender Netzwerkverkehr blockiert werden. Damit der zu ausgehenden Verbindungen korrespondierende eingehende Netzwerkverkehr die Firewall passieren kann, muss ein State vorgehalten werden. Dieser speichert, welche Verbindungen von welchem Rechner aus dem Netzwerk ausgegangen ist. Weil keinerlei direkte Verbindungen zu Clients möglich sind, können diese somit nicht von extern mit Adresssuchverfahren gefunden werden. Dies entspricht dem Verhalten von IPv4, bei der Benutzung von NAT.
Verlangsamt werden kann der Scan eines Angreifers, wenn bei blockierten (oder nicht genutzten) TCP-Ports statt mit RST gar nicht geantwortet wird – das Scanprogramm weiß in diesem Fall nicht, ob das Paket aufgrund von Überlast verworfen wurde und muss dieses (wenn es den Anspruch hat, genau zu arbeiten) erneut senden. Bei UDP gilt Ähnliches – wenn ein Port nicht erreichbar ist, wird normalerweise eine ICMPv6-Nachricht zurückgesandt. Um den Scan zu verlangsamen, sollte somit keine Antwort erfolgen. (vgl. Lyon 2009, S. 445 f.)
Eine andere Methode, Adresssuchverfahren zu verhindern beziehungsweise auszubremsen, ist, mittels eines Paketfilters oder IDS scannende Hosts zu blockieren. Zum Beispiel ist es beim Paketfilter `iptables` möglich, Rechner zu blockieren, die in einem gewissen Zeitabschnitt eine bestimmte Anzahl von bestimmten Paketen versenden. Somit könnten Hosts, die beispielsweise in einer Minute 100 ICMPv6-Echo-Request an das Netzwerk adressiert haben, für eine gewisse Zeit oder für immer auf eine Blacklist gesetzt werden.
Diese Paketfiltertechnik kann bei Servern eingesetzt werden. Allerdings ist sie kein wirksamer Schutz. Ein Scan kann von mehreren Hosts ausgeführt werden, oder der Scannende bemerkt das Blockieren und verlangsamt sein Scanprogramm bei zukünftigen Suchen.

7 Zusammenfassung

Das Scannen von entfernten IPv6-Subnetzen wurde lange Zeit für als nicht durchführbar deklariert. Der Suchraum kann (wie diese Arbeit zeigt) mit neueren Adresssuchverfahren deutlich verkleinert werden. Allerdings ist er selbst dann noch immer größer als der gesamte Adressraum bei IPv4, weswegen das Scannen von entfernten Subnetzen sehr lange dauert. Falls der Suchraum allerdings weiter eingegrenzt werden kann (zum Beispiel durch Kenntnis des Netzwerkes oder der darin verwendeten Geräte und Adressvergabeverfahren), dann ist das Scannen in der Praxis denkbar. Zur Inventarisierung von Hosts im eigenen Netzwerk sollten, insofern möglich, lokale Adresssuchverfahren eingesetzt werden, da diese deutlich schneller sind.
Zum Suchen von entfernten IPv6-Adressen wurden in dieser Bachelorarbeit vier NSE-Programme angefertigt, die unter der GPLv2 lizensiert sind. Mit Nmap kann nun somit auch nach bestimmten Adressmustern gesucht werden. Offen bleibt, ob sinnvollerweise IPv6-Adressen gescannt werden können, die in Hexadezimalschreibweise Wörter inkludiert haben – dies bedarf weiterer Forschung.
Da auch Angreifer Hosts suchen, um diese eventuell zu kompromittieren, wurden auch kurz Abwehrmaßnahmen behandelt. Zumindest bei Clients ist es bei IPv6 durchführbar, dass diese nicht aufzufinden sind. Bei öffentlichen Servern hingegen kann der Zeitpunkt der Entdeckung aufgeschoben werden, da sie Dienste anbieten, sind sie jedoch letztendlich auffindbar.

8 Literaturverzeichnis

Berrera et al. 2010

Berrera, D. ; Wurster, G. ; Oorschot, P.C. van: Back to the Future: Revisiting IPv6 Privacy Extension / Carleton University - School of Computer Science. Version: September 2010. `http://www.ccsl.carleton.ca/~dbarrera/files/TR-10-17.pdf`. 2010. – Forschungsbericht. – (Zuletzt aufgerufen: 7. Juni 2013)

Chown 2008

Chown, T.: *IPv6 Implications for Network Scanning*. RFC 5157 (Informational). `http://www.ietf.org/rfc/rfc5157.txt`. Version: März 2008 (Request for Comments)

Conta et al. 2006

Conta, A. ; Deering, S. ; Gupta, M.: *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. RFC 4443 (Draft Standard). `http://www.ietf.org/rfc/rfc4443.txt`. Version: März 2006 (Request for Comments). – Updated by RFC 4884

Crawford 1998

Crawford, M.: *Transmission of IPv6 Packets over Ethernet Networks*. RFC 2464 (Proposed Standard). `http://www.ietf.org/rfc/rfc2464.txt`. Version: Dezember 1998 (Request for Comments). – Updated by RFC 6085

Deering und Hinden 1998

Deering, S. ; Hinden, R.: *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). `http://www.ietf.org/rfc/rfc2460.txt`. Version: Dezember 1998 (Request for Comments). – Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946

van Dijk 2012

Dijk, P. van: *Finding v6 hosts by efficiently mapping ip6.arpa*. `http://7bits.nl/blog/2012/03/26/finding-v6-hosts-by-efficiently-mapping-ip6-arpa`. Version: März 2012. – (Zuletzt aufgerufen: 4. August 2013)

Droms et al. 2003

Droms, R. ; Bound, J. ; Volz, B. ; Lemon, T. ; Perkins, C. ; Carney, M.: *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*. RFC 3315 (Proposed Standard). `http://www.ietf.org/rfc/rfc3315.txt`. Version: Juli 2003 (Request for Comments). – Updated by RFCs 4361, 5494, 6221, 6422, 6644

Ford 2013

Ford, M.: *IPv6 Address Analysis - Privacy In, Transition Out*. http://www.internetsociety.org/blog/2013/05/ipv6-address-analysis-privacy-transition-out. Version: Mai 2013. – (Zuletzt aufgerufen: 15. August 2013)

Gont 2013

Gont, F.: *IPv6 Network Reconnaissance: Theory & Practice (LACSEC 2013 Conference, Medellin, Colombia.)*. http://www.si6networks.com/presentations/lacnic19/lacsec2013-fgont-ipv6-network-reconnaissance.pdf. Version: Mai 2013. – (Zuletzt aufgerufen: 15. August 2013)

Gont und Chown 2013

Gont, F. ; Chown, T.: *Network Reconnaissance in IPv6 Networks draft-ietf-opsec-ipv6-host-scanning-02*. Draft (Informational). http://tools.ietf.org/id/draft-ietf-opsec-ipv6-host-scanning-02.txt. Version: Juli 2013 (Request for Comments Draft)

Hagen 2009

Hagen, S.: *IPv6 Grundlagen - Funktionalität - Integration (2. Auflage)*. Maur : Sunny Edition, 2009

Hardee 2011

Hardee, M.: *Fun with IPv6*. http://blogs.cisco.com/socialmedia/fun-with-ipv6/. Version: Mai 2011. – (Zuletzt aufgerufen: 20. August 2013)

Hinden und Deering 2006

Hinden, R. ; Deering, S.: *IP Version 6 Addressing Architecture*. RFC 4291 (Draft Standard). http://www.ietf.org/rfc/rfc4291.txt. Version: Februar 2006 (Request for Comments). – Updated by RFCs 5952, 6052

IANA 2012

IANA: *IANA — Number Resources*. https://www.iana.org/numbers. Version: 2012. – (Zuletzt aufgerufen: 3. August 2013)

IEEE-SA 2008

IEEE-SA: IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. In: *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)* (2008)

Ierusalimschy et al. 2013

Ierusalimschy, R. ; Figueiredo, L. H. ; Celes, W.: *Lua 5.2 Reference Manual*. http://www.lua.org/manual/5.2/manual.html. Version: März 2013. – (Zuletzt aufgerufen: 21. Juli 2013)

Lyon 2009

Lyon, G.: *Nmap - Netzwerke scannen, analysieren und absichern. Deutsch von Dino Gherman*. München : Open Source Press, 2009 (root's reading)

Mockapetris 1987

Mockapetris, P.V.: *Domain names - concepts and facilities*. RFC 1034 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc1034.txt. Version: November 1987 (Request for Comments). – Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936

Narten et al. 2007

Narten, T. ; Draves, R. ; Krishnan, S.: *Privacy Extensions for Stateless Address Autoconfiguration in IPv6*. RFC 4941 (Draft Standard). http://www.ietf.org/rfc/rfc4941.txt. Version: September 2007 (Request for Comments)

Nmap 2011

Nmap: *Host Discovery*. http://nmap.org/book/man-host-discovery.html. Version: 2011. – (Zuletzt aufgerufen: 15. Juli 2013)

Nmap 2012a

Nmap: *Nmap 6 Release Notes*. http://nmap.org/6/. Version: Mai 2012. – (Zuletzt aufgerufen: 14. Juli 2013)

Nmap 2012b

Nmap: *target NSE Library*. http://nmap.org/nsedoc/lib/target.html. Version: 2012. – (Zuletzt aufgerufen: 8. August 2013)

Nmap 2013

Nmap: *TODO*. https://svn.nmap.org/nmap/todo/nmap.txt. Version: Mai 2013. – (Zuletzt aufgerufen: 1. Mai 2013)

Tanenbaum und Wetherall 2011

Tanenbaum, A. S. ; Wetherall, D. J.: *Computer Networks*. 5th. Boston : Pearson Prentice Hall, 2011

Thomson et al. 2007

Thomson, S. ; Narten, T. ; Jinmei, T.: *IPv6 Stateless Address Autoconfigu-*

ration. RFC 4862 (Draft Standard). `http://www.ietf.org/rfc/rfc4862.txt`. Version: September 2007 (Request for Comments)

Wolf 2009

Wolf, J.: *Linux-UNIX-Programmierung: Das umfassende Handbuch (3., aktualisierte und erweiterte Auflage)*. Bonn : Galileo Press GmbH, 2009 (Galileo Computing)

Anhang

Quelltext targets-ipv6-range.nse

```
local ipOps = require "ipOps"
local nmap = require "nmap"
local stdnse = require "stdnse"
local target = require "target"

description = [[
Generates targets for IPv6 ranges.

Supported are range definitions in prefix length (i.e. 2001:db0::/112) and from-to (i.e. 2001:db0::1-200:1-300:)
notation.

At this point Nmap doesn't support IPv6 prefix length notation and starts scanning after all targets have been added.
This leads to massive memory consumption if you want to add a huge amount of addresses.
(2^24 addresses will take 1.2 gigabyte (or more) of RAM.)
]]

---
-- @usage
-- nmap -6 --script=targets-ipv6-range.nse --script-args 'targets-ipv6-range.net6=2001:db8::/127'
-- @output
-- Pre-scan script results:
-- | targets-ipv6-embedded-ipv4:
-- |   IP: 2001:db8:0:0:0:0:0:0
-- |   IP: 2001:db8:0:0:0:0:0:1
-- |_  Use --script-args=newtargets to add the results as targets
-- @args newtargets  If true, add generated targets to the scan queue.
-- @args targets-ipv6-range.net6  The IPv6 networks(s) to scan in prefix length or from-to notation.

author = "Manuel Hachtkemper"

license = "Same as Nmap--See http://nmap.org/book/man-legal.html"

categories = {"intrusive"}

prerule = function() return true end

action = function()

    local result = {}
    local targetaction

    if target.ALLOW_NEW_TARGETS then
        targetaction = function(a, b, c, d, e, f, g, h)
            target.add(string.format("%x:%x:%x:%x:%x:%x:%x:%x", a, b, c, d, e, f, g, h))
        end
    else
        targetaction = function(a, b, c, d, e, f, g, h)
            result[#result + 1] = string.format("IP: %x:%x:%x:%x:%x:%x:%x:%x", a, b, c, d, e, f, g, h)
        end
    end

    -- IPv6 input
    local net6 = stdnse.get_script_args(SCRIPT_NAME .. ".net6")

    if not net6 then
        stdnse.print_debug("You have to specify an IPv6 address range!")
        return false
    end

    if type(net6) ~= "table" then
        net6 = {net6}
    end

    -- iterate IPv6 networks
    for _, rangestart in ipairs(net6) do
        local rangeend = ""
        -- split IPv6 address and netsize
        if string.find(rangestart, "/") then
            rangestart, rangeend = ipOps.get_ips_from_range(rangestart)
        else
        -- set IPv6 range start- and endpoint
            local count = 0
            for _, s in ipairs(stdnse.strsplit(":", rangestart)) do
                local tmp1, tmp2
                local dashpos = string.find(s, "-")
```

```
                if dashpos then
                    tmp1 = string.sub(s, 1, dashpos - 1)
                    tmp2 = string.sub(s, dashpos + 1)
                else
                    tmp1 = s
                    tmp2 = s
                end
                if count == 0 then
                    rangestart = tmp1
                    rangeend = tmp2
                elseif s == "" then
                    rangestart = rangestart .. ":"
                    rangeend = rangeend .. ":"
                else
                    rangestart = rangestart .. ":" .. tmp1
                    rangeend = rangeend .. ":" .. tmp2
                end
                count = count + 1
            end
        end
        stdnse.print_debug("IPv6 range start address: " .. rangestart)
        stdnse.print_debug("IPv6 range end address: " .. rangeend)

        -- split and verify rangestart and rangeend
        local t, u, err
        t, err = ipOps.get_parts_as_number(rangestart)
        if not t then
            stdnse.print_debug("Invalid IPv6 address! (range start)")
            return false
        end
        u, err = ipOps.get_parts_as_number(rangeend)
        if not u then
            stdnse.print_debug("Invalid IPv6 address! (range end)")
            return false
        end

        -- generate addresses
        for a=t[1], u[1], 1 do
            for b=t[2], u[2], 1 do
                for c=t[3], u[3], 1 do
                    for d=t[4], u[4], 1 do
                        for e=t[5], u[5], 1 do
                            for f=t[6], u[6], 1 do
                                for g=t[7], u[7], 1 do
                                    for h=t[8], u[8], 1 do
                                        targetaction(a,b,c,d,e,f,g,h)
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end

    if target.ALLOW_NEW_TARGETS then
        return true
    else
        result[#result + 1] = "Use --script-args=newtargets to add the results as targets"
        return stdnse.format_output(true, result)
    end
end
```

Quelltext targets-ipv6-slaac.nse

```
local bit = require "bit"
local ipOps = require "ipOps"
local nmap = require "nmap"
local stdnse = require "stdnse"
local target = require "target"

description = [[
Generates targets for IPv6 SLAAC addresses.
See http://tools.ietf.org/html/draft-ietf-opsec-ipv6-host-scanning-02 chapter 3.1.1.1.

Supported is only the from-to (i.e. 2001:db0::1-ff:1-ffff) notation. Values inserted in the reserved field of the
Interface Identifier (Bits 64-111 MSB 0) will be silently ignored and overwritten with the SLAAC values.

With "targets-ipv6-slaac.ouicname" you can provide (multiple) name(s) to search for their corresponding OUI. Note
that "Apple" will also match "Apple Computers" and ".*" matches all entries. To only match "Apple" provide "Apple$" as
value.
If you want to use the name feature and don't have a file called /usr/share/nmap/nmap-mac-prefixes on your system you
have to provide a file with "targets-ipv6-slaac.ouicfile" containing "OUI company_name" pairs separated by
newlines. The OUI must not contain colons.

At this point Nmap doesn't support IPv6 prefix length notation and starts scanning after all targets have been added.
This leads to massive memory consumption if you want to add a huge amount of addresses.
(2^24 addresses will take 1.2 gigabyte (or more) of RAM.)
]]

---
-- @usage
-- nmap -6 --script=targets-ipv6-slaac.nse --script-args 'targets-ipv6-slaac.net6=2001:db8::1-2,\
-- targets-ipv6-slaac.ouicid={012345,6789AB},targets-ipv6-slaac.ouicidname=Atmel'
-- @output
-- Pre-scan script results:
-- | targets-ipv6-embedded-ipv4:
-- |   IP: 2001:db8:0:0:323:45ff:fe00:1
-- |   IP: 2001:db8:0:0:323:45ff:fe00:2
-- |   IP: 2001:db8:0:0:6789:abff:fe00:1
-- |   IP: 2001:db8:0:0:6789:abff:fe00:2
-- |   IP: 2001:db8:0:0:204:25ff:fe00:1
-- |   IP: 2001:db8:0:0:204:25ff:fe00:2
-- |   IP: 2001:db8:0:0:fec2:3dff:fe00:1
-- |   IP: 2001:db8:0:0:fec2:3dff:fe00:2
-- |_  Use --script-args=newtargets to add the results as targets
-- @args newtargets  If true, add generated targets to the scan queue.
-- @args targets-ipv6-slaac.net6  The IPv6 network(s) to scan in from-to notation.
-- @args targets-ipv6-slaac.ouicid  OUIs (without colons).
-- @args targets-ipv6-slaac.ouicidfile  File which contains "OUI company_name" pairs seperated with newlines.
-- @args targets-ipv6-slaac.ouicidname  Match OUI from given company name(s).

author = "Manuel Hachtkemper"

license = "Same as Nmap--See http://nmap.org/book/man-legal.html"

categories = {"intrusive"}

prerule = function() return true end

action = function()

    local result = {}
    local targetaction

    if target.ALLOW_NEW_TARGETS then
        targetaction = function(a, b, c, d, e, f, g, h)
            target.add(string.format("%x:%x:%x:%x:%x:%x:%x:%x", a, b, c, d, e, f, g, h))
        end
    else
        targetaction = function(a, b, c, d, e, f, g, h)
            result[#result + 1] = string.format("IP: %x:%x:%x:%x:%x:%x:%x:%x", a, b, c, d, e, f, g, h)
        end
    end

    -- IPv6 input
    local net6 = stdnse.get_script_args(SCRIPT_NAME .. ".net6")

    if not net6 then
        stdnse.print_debug("You have to specify an IPv6 address range!")
        return false
    end
```

```
if type(net6) ~= "table" then
    net6 = {net6}
end

-- iterate IPv6 networks
for _, rangestart in ipairs(net6) do
    if string.find(rangestart, "/") then
        stdnse.print_debug("Prefix length notation isn't supported in this script!")
        return false
    end
    -- set IPv6 range start- and endpoint
    local rangeend = ""
    local count = 0
    for _, s in ipairs(stdnse.strsplit(":", rangestart)) do
        local tmp1, tmp2
        local dashpos = string.find(s, "-")
        if dashpos then
            tmp1 = string.sub(s, 1, dashpos - 1)
            tmp2 = string.sub(s, dashpos + 1)
        else
            tmp1 = s
            tmp2 = s
        end
        if count == 0 then
            rangestart = tmp1
            rangeend = tmp2
        elseif s == "" then
            rangestart = rangestart .. ":"
            rangeend = rangeend .. ":"
        else
            rangestart = rangestart .. ":" .. tmp1
            rangeend = rangeend .. ":" .. tmp2
        end
        count = count + 1
    end
    stdnse.print_debug("IPv6 range start address: " .. rangestart)
    stdnse.print_debug("IPv6 range end address: " .. rangeend)

    -- split and verify rangestart and rangeend
    local t, u, err
    t, err = ipOps.get_parts_as_number(rangestart)
    if not t then
        stdnse.print_debug("Invalid IPv6 address! (range start)")
        return false
    end
    u, err = ipOps.get_parts_as_number(rangeend)
    if not u then
        stdnse.print_debug("Invalid IPv6 address! (range end)")
        return false
    end

    -- OUI to IPv6 Interface Identifier
    local ouicid = stdnse.get_script_args(SCRIPT_NAME .. ".ouicid")
    if type(ouicid) ~= "table" then
        ouicid = {ouicid}
    end
    local ouicidfile = stdnse.get_script_args(SCRIPT_NAME .. ".ouicidfile") or
      "/usr/share/nmap/nmap-mac-prefixes"
    local ouicidname = stdnse.get_script_args(SCRIPT_NAME .. ".ouicidname")
    if ouicidname then
        if type(ouicidname) ~= "table" then
            ouicidname = {ouicidname}
        end
        if ouicidfile then
            local f = io.open(ouicidfile, "r")
            if not f then
                stdnse.print_debug("Couldn't open OUI file %s!", ouicidfile)
                return false
            end
            for line in f:lines() do
                for _, ouicidname in ipairs(ouicidname) do
                    local oui = string.match(line, '^(%x%x%x%x%x%x) .*' .. ouicidname)
                    if oui then
                        stdnse.print_debug("Adding company_id: %s", oui)
                        ouicid[#ouicid + 1] = oui
                    end
                end
            end
        else
```

```
                stdnse.print_debug("You have to provide both ouicidfile and ouicidname!")
                return false
            end

        end

        -- generate SLAAC style addresses
        for _, ouicid in ipairs(ouicid) do
            -- set bit to 1 as required by RFC 4862
            t[5] = bit.bor(tonumber(string.sub(ouicid, 1, 4), 16), 0x0200)
            u[5] = t[5]
            -- last two hex digits of OUI and ff
            t[6] = tonumber(string.sub(ouicid, 5) .. "ff", 16)
            u[6] = t[6]
            -- merge of user defined values and fe00
            t[7] = bit.band(bit.bor(0xfe00, t[7]), 0xfeff)
            u[7] = bit.band(bit.bor(0xfe00, u[7]), 0xfeff)

            for a=t[1], u[1], 1  do
                for b=t[2], u[2], 1 do
                    for c=t[3], u[3], 1 do
                        for d=t[4], u[4], 1 do
                            for e=t[5], u[5], 1 do
                                for f=t[6], u[6], 1 do
                                    for g=t[7], u[7], 1 do
                                        for h=t[8], u[8], 1 do
                                            targetaction(a,b,c,d,e,f,g,h)
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end

    if target.ALLOW_NEW_TARGETS then
        return true
    else
        result[#result + 1] = "Use --script-args=newtargets to add the results as targets"
        return stdnse.format_output(true, result)
    end
end
```

Quelltext targets-ipv6-embedded-ipv4.nse

```
local ipOps = require "ipOps"
local nmap = require "nmap"
local stdnse = require "stdnse"
local target = require "target"

description = [[
Generates targets for IPv4 addresses which are embedded in IPv6 addresses.
See http://tools.ietf.org/html/draft-ietf-opsec-ipv6-host-scanning-02 chapter 3.1.3.2.

Currently this script supports three types of these addresses:
    - IPv4-Compatible IPv6 Address style (2001:db0::192.0.2.1)
    - IPv4-Mapped IPv6 Address style (2001:db0::ffff:192.0.2.1)
    - IPv4-Inserted IPv6 address style (2001:db0::192:0:2:1)

IPv6 range definitions in prefix length (i.e. 2001:db0::/63) and from-to (i.e. 2001:db0:1-200:1-300::) notation are
supported. Values inserted in the reserved field of the Interface Identifier (Bits 64-111 MSB 0) will be silently
ignored and overwritten with the IPv4 values.
IPv4 range definitions in prefix length (i.e. 192.0.2.0/30) and from-to (i.e. 192.0.2.0-4) notation are supported.

At this point Nmap doesn't support IPv6 prefix length notation and starts scanning after all targets have been added.
This leads to massive memory consumption if you want to add a huge amount of addresses.
(2^24 addresses will take 1.2 gigabyte (or more) of RAM.)
]]

---
-- @usage
-- nmap -6 --script=targets-ipv6-embedded-ipv4.nse --script-args 'targets-ipv6-embedded-ipv4.net6=2001:db8::/63,\
--   targets-ipv6-embedded-ipv4.net4={192.0.2.0/30,198.51.100.1}'
-- @output
-- Pre-scan script results:
-- | targets-ipv6-embedded-ipv4:
-- |   IP: 2001:db8:0:0::192.0.2.0
-- |   IP: 2001:db8:0:0::192.0.2.1
-- |   IP: 2001:db8:0:0::192.0.2.2
-- |   IP: 2001:db8:0:0::192.0.2.3
-- |   IP: 2001:db8:0:1::192.0.2.0
-- |   IP: 2001:db8:0:1::192.0.2.1
-- |   IP: 2001:db8:0:1::192.0.2.2
-- |   IP: 2001:db8:0:1::192.0.2.3
-- |   IP: 2001:db8:0:0::198.51.100.1
-- |   IP: 2001:db8:0:1::198.51.100.1
-- |_  Use --script-args=newtargets to add the results as targets
-- @args newtargets  If true, add generated targets to the scan queue.
-- @args targets-ipv6-embedded-ipv4.net4  The IPv4 network(s) to insert into the IPv6 address.
-- @args targets-ipv6-embedded-ipv4.net6  The IPv6 network(s) to scan.
-- @args targets-ipv6-embedded-ipv4.addressstyle  Format of IPv4 embedded in IPv6 address. Possible values are \
--   compatible, inserted or mapped. The default is compatible.

author = "Manuel Hachtkemper"

license = "Same as Nmap--See http://nmap.org/book/man-legal.html"

categories = {"intrusive"}

prerule = function() return true end

action = function()

    local result = {}
    local targetaction

    -- select address format
    local addressstyle = stdnse.get_script_args(SCRIPT_NAME .. ".addressstyle")
    if addressstyle == "mapped" then
        addressstyle = "%x:%x:%x:%x::ffff:%d.%d.%d.%d"
    elseif addressstyle == "inserted" then
        addressstyle = "%x:%x:%x:%x:%d:%d:%d:%d"
    else
        addressstyle = "%x:%x:%x:%x::%d.%d.%d.%d"
    end

    if target.ALLOW_NEW_TARGETS then
        targetaction = function(a, b, c, d, e, f, g, h)
            target.add(string.format(addressstyle, a, b, c, d, e, f, g, h))
        end
    else
        targetaction = function(a, b, c, d, e, f, g, h)
```

```
        result[#result + 1] = string.format("IP: " .. addressstyle, a, b, c, d, e, f, g, h)
    end
end

-- IPv6 input
local net6 = stdnse.get_script_args(SCRIPT_NAME .. ".net6")

if not net6 then
    stdnse.print_debug("You have to specify an IPv6 address range!")
    return false
end

if type(net6) ~= "table" then
    net6 = {net6}
end

-- IPv4 input
local net4 = stdnse.get_script_args(SCRIPT_NAME .. ".net4")

if not net4 then
    stdnse.print_debug("You have to specify an IPv4 address range!")
    return false
end

if type(net4) ~= "table" then
    net4 = {net4}
end

-- iterate IPv6 networks
for _, rangestart in ipairs(net6) do
    local rangeend = ""
    if string.find(rangestart, "/") then
        rangestart, rangeend = ipOps.get_ips_from_range(rangestart)
    else
        -- set IPv6 range start- and endpoint
        local count = 0
        for _, s in ipairs(stdnse.strsplit(":", rangestart)) do
            local tmp1, tmp2
            local dashpos = string.find(s, "-")
            if dashpos then
                tmp1 = string.sub(s, 1, dashpos - 1)
                tmp2 = string.sub(s, dashpos + 1)
            else
                tmp1 = s
                tmp2 = s
            end
            if count == 0 then
                rangestart = tmp1
                rangeend = tmp2
            elseif s == "" then
                rangestart = rangestart .. ":"
                rangeend = rangeend .. ":"
            else
                rangestart = rangestart .. ":" .. tmp1
                rangeend = rangeend .. ":" .. tmp2
            end
            count = count + 1
        end
    end
    stdnse.print_debug("IPv6 range start address: " .. rangestart)
    stdnse.print_debug("IPv6 range end address: " .. rangeend)

    -- split and verify rangestart and rangeend
    local t, u, err
    t, err = ipOps.get_parts_as_number(rangestart)
    if not t then
        stdnse.print_debug("Invalid IPv6 address! (range start)")
        return false
    end
    u, err = ipOps.get_parts_as_number(rangeend)
    if not u then
        stdnse.print_debug("Invalid IPv6 address! (range end)")
        return false
    end

    -- iterate IPv4 networks
    for _, rangestart in ipairs(net4) do
        local rangeend
        if string.find(rangestart, "/") then
            rangestart, rangeend = ipOps.get_ips_from_range(rangestart)
```

```
            else
                -- set IPv4 range start- and endpoint
                local count = 0
                for _, s in ipairs(stdnse.strsplit("%.", rangestart)) do
                    local tmp1, tmp2
                    local dashpos = string.find(s, "-")
                    if dashpos then
                        tmp1 = string.sub(s, 1, dashpos - 1)
                        tmp2 = string.sub(s, dashpos + 1)
                    else
                        tmp1 = s
                        tmp2 = s
                    end
                    if count == 0 then
                        rangestart = tmp1
                        rangeend = tmp2
                    else
                        rangestart = rangestart .. "." .. tmp1
                        rangeend = rangeend .. "." .. tmp2
                    end
                    count = count + 1
                end
            end
            stdnse.print_debug("IPv4 range start address: " .. rangestart)
            stdnse.print_debug("IPv4 range end address: " .. rangeend)

            -- split and verify rangestart and rangeend
            local v, w
            v, err = ipOps.get_parts_as_number(rangestart)
            w, err = ipOps.get_parts_as_number(rangeend)
            if not v then
                stdnse.print_debug("Invalid IPv4 address!")
                return false
            end
            if not w then
                stdnse.print_debug("Invalid IPv4 address!")
                return false
            end

            -- generate addresses
            for a=t[1], u[1], 1 do
                for b=t[2], u[2], 1 do
                    for c=t[3], u[3], 1 do
                        for d=t[4], u[4], 1 do
                            for e=v[1], w[1], 1 do
                                for f=v[2], w[2], 1 do
                                    for g=v[3], w[3], 1 do
                                        for h=v[4], w[4], 1 do
                                            targetaction(a,b,c,d,e,f,g,h)
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end

    if target.ALLOW_NEW_TARGETS then
        return true
    else
        result[#result + 1] = "Use --script-args=newtargets to add the results as targets"
        return stdnse.format_output(true, result)
    end
end
```

Quelltext targets-ipv6-ports.nse

```
local ipOps = require "ipOps"
local nmap = require "nmap"
local stdnse = require "stdnse"
local target = require "target"

description = [[
Generates targets for IPv6 service-port addresses.
See http://tools.ietf.org/html/draft-ietf-opsec-ipv6-host-scanning-02 chapter 3.1.3.3.

Supported are only range definitions in from-to (i.e. 2001:db0::1-200:1-300:) notation. To insert the port numbers
into an address place a "x" in the address definition (i.e. 2001:db0::1-200:1-300:x).
The default values for x are:
21 22 23 25 49 53 80 110 123 179 220 389 443 547 993 995 1194 3306 5060 5061 5432 6446 8080

At this point Nmap doesn't support IPv6 prefix length notation and starts scanning after all targets have been added.
This leads to massive memory consumption if you want to add a huge amount of addresses.
(2^24 addresses will take 1.2 gigabyte (or more) of RAM.)
]]

---
-- @usage
-- nmap -6 --script=targets-ipv6-ports.nse --script-args 'targets-ipv6-ports.net6=2001:db8::x,\
--   targets-ipv6-ports.format=dez'
-- @output
-- Pre-scan script results:
-- | targets-ipv6-ports:
-- |   IP: 2001:db8:0:0:0:0:0:21
-- |   IP: 2001:db8:0:0:0:0:0:22
-- |   IP: 2001:db8:0:0:0:0:0:23
-- |   IP: 2001:db8:0:0:0:0:0:25
-- |   IP: 2001:db8:0:0:0:0:0:49
-- |   IP: 2001:db8:0:0:0:0:0:53
-- |   IP: 2001:db8:0:0:0:0:0:80
-- |   IP: 2001:db8:0:0:0:0:0:110
-- |   IP: 2001:db8:0:0:0:0:0:123
-- |   IP: 2001:db8:0:0:0:0:0:179
-- |   IP: 2001:db8:0:0:0:0:0:220
-- |   IP: 2001:db8:0:0:0:0:0:389
-- |   IP: 2001:db8:0:0:0:0:0:443
-- |   IP: 2001:db8:0:0:0:0:0:547
-- |   IP: 2001:db8:0:0:0:0:0:993
-- |   IP: 2001:db8:0:0:0:0:0:995
-- |   IP: 2001:db8:0:0:0:0:0:1194
-- |   IP: 2001:db8:0:0:0:0:0:3306
-- |   IP: 2001:db8:0:0:0:0:0:5060
-- |   IP: 2001:db8:0:0:0:0:0:5061
-- |   IP: 2001:db8:0:0:0:0:0:5432
-- |   IP: 2001:db8:0:0:0:0:0:6446
-- |   IP: 2001:db8:0:0:0:0:0:8080
-- |_  Use --script-args=newtargets to add the results as targets
-- @args newtargets  If true, add generated targets to the scan queue.
-- @args targets-ipv6-ports.format  Format of ports. Possible values are hex, dec or both. The default is both.
-- @args targets-ipv6-ports.net6  The IPv6 network(s) to scan in from-to notation.
-- @args targets-ipv6-ports.ports  Define a custom port list.
-- @args targets-ipv6-ports.addports  Add ports to the port list.

author = "Manuel Hachtkemper"

license = "Same as Nmap--See http://nmap.org/book/man-legal.html"

categories = {"intrusive"}

prerule = function() return true end

action = function()

    local result = {}
    local targetaction

    if target.ALLOW_NEW_TARGETS then
        targetaction = function(a, b, c, d, e, f, g, h)
            target.add(string.format("%x:%x:%x:%x:%x:%x:%x:%x", a, b, c, d, e, f, g, h))
        end
    else
        targetaction = function(a, b, c, d, e, f, g, h)
            result[#result + 1] = string.format("IP: %x:%x:%x:%x:%x:%x:%x:%x", a, b, c, d, e, f, g, h)
        end
```

```
end

-- IPv6 input
local net6 = stdnse.get_script_args(SCRIPT_NAME .. ".net6")

if not net6 then
    stdnse.print_debug("You have to specify an IPv6 address range!")
    return false
end

if type(net6) ~= "table" then
    net6 = {net6}
end

local ports = stdnse.get_script_args(SCRIPT_NAME .. ".ports") or
    {21,22,23,25,49,53,80,110,123,179,220,389,443,547,993,995,1194,3306,5060,5061,5432,6446,8080}
if type(ports) ~= "table" then
    ports = {ports}
end

local addports = stdnse.get_script_args(SCRIPT_NAME .. ".addports")
if type(addports) ~= "table" then
    addports = {addports}
end
if addports then
    for _, addport in ipairs(addports) do
        ports[#ports + 1] = addport
    end
end

local format = stdnse.get_script_args(SCRIPT_NAME .. ".format")
local portlist = {}
if format == "dec" then
    portlist = ports
elseif format == "hex" then
    for _, port in ipairs(ports) do
        portlist[#portlist + 1] = string.format("%x", port)
    end
else
    for _, port in ipairs(ports) do
        portlist[#portlist + 1] = string.format("%x", port)
        portlist[#portlist + 1] = port
    end
end

local nets6 = {}
for _, net6 in ipairs(net6) do
    for _, port in ipairs(portlist) do
        nets6[#nets6 + 1] = string.gsub(net6, "x", port)
    end
end

-- iterate IPv6 networks
for _, rangestart in ipairs(nets6) do
    if string.find(rangestart, "/") then
        stdnse.print_debug("Prefix length notation isn't supported in this script!")
        return false
    end
    -- set IPv6 range start- and endpoint
    local rangeend = ""
    local count = 0
    for _, s in ipairs(stdnse.strsplit(":", rangestart)) do
        local tmp1, tmp2
        local dashpos = string.find(s, "-")
        if dashpos then
            tmp1 = string.sub(s, 1, dashpos - 1)
            tmp2 = string.sub(s, dashpos + 1)
        else
            tmp1 = s
            tmp2 = s
        end
        if count == 0 then
            rangestart = tmp1
            rangeend = tmp2
        elseif s == "" then
            rangestart = rangestart .. ":"
            rangeend = rangeend .. ":"
        else
            rangestart = rangestart .. ":" .. tmp1
            rangeend = rangeend .. ":" .. tmp2
```

```
            end
            count = count + 1
        end
        stdnse.print_debug("IPv6 range start address: " .. rangestart)
        stdnse.print_debug("IPv6 range end address: " .. rangeend)

        -- split and verify rangestart and rangeend
        local t, u, err
        t, err = ipOps.get_parts_as_number(rangestart)
        if not t then
            stdnse.print_debug("Invalid IPv6 address! (range start)")
            return false
        end
        u, err = ipOps.get_parts_as_number(rangeend)
        if not u then
            stdnse.print_debug("Invalid IPv6 address! (range end)")
            return false
        end

        -- generate addresses
        for a=t[1], u[1], 1 do
            for b=t[2], u[2], 1 do
                for c=t[3], u[3], 1 do
                    for d=t[4], u[4], 1 do
                        for e=t[5], u[5], 1 do
                            for f=t[6], u[6], 1 do
                                for g=t[7], u[7], 1 do
                                    for h=t[8], u[8], 1 do
                                        targetaction(a,b,c,d,e,f,g,h)
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end

    if target.ALLOW_NEW_TARGETS then
        return true
    else
        result[#result + 1] = "Use --script-args=newtargets to add the results as targets"
        return stdnse.format_output(true, result)
    end
end
```

Nmap-Messung

nmap -6 -n -PE -sn –script=targets-ipv6-range.nse –script-args=newtargets,targets-ipv6-range.net6=2001:4dd0:fdb0:ffff::/120 -T3									
2,86s	4,09s	3,12s	3,26s	3,04s	3,06s	3,10s	4,04s	4,05s	4,86s
nmap -6 -n -PE -sn –script=targets-ipv6-range.nse –script-args=newtargets,targets-ipv6-range.net6=2001:4dd0:fdb0:ffff::/120 -T5									
1,96s	1,41s	1,95s	1,93s	1,65s	1,63s	1,66s	2,56s	2,57s	1,70s
nmap -6 -n -PE -sn –script=targets-ipv6-range.nse –script-args=newtargets,targets-ipv6-range.net6=2001:4dd0:fdb0:ffff::1-5000 -T3									
392,49s	599,67s	400,09s	812,98s	593,31s	587,18s	585,51s	397,00s	613,56s	800,72s
nmap -6 -n -PE -sn –script=targets-ipv6-range.nse –script-args=newtargets,targets-ipv6-range.net6=2001:4dd0:fdb0:ffff::1-5000 -T5									
250,07s	249,64s	312,50s	242,49s	207,47s	208,00s	209,49s	211,41s	211,18s	286,85s
nmap -6 -n -PE -sn –script=targets-ipv6-range.nse –script-args=newtargets,targets-ipv6-range.net6=2001:4dd0:fdb0:ffff::/112 -T3									
1333,94s	1323,08s	1349,80s	1321,83s	1331,48s	1323,96s	1315,32s	1343,01s	1328,89s	1337,08s
nmap -6 -n -PE -sn –script=targets-ipv6-range.nse –script-args=newtargets,targets-ipv6-range.net6=2001:4dd0:fdb0:ffff::/112 -T5									
742,46s	724,69s	744,60s	724,43s	744,88s	731,49s	755,51s	732,88s	727,44s	731,35s

Abbildung 17: Rohdaten der Messung mit Nmap