Master's Thesis

TCP Idle Scans in IPv6

Mathias Morbitzer

m.morbitzer@student.ru.nl

Thesis number: 670

August 2013

Supervisors:
Erik Poll
Hendrik Schimmelpenninck van der Oye (Fox-IT)
Mark Koek (Fox-IT)

Department of Computing Science

Radboud University Nijmegen

Abstract

Port scans are done by an attacker to discover which services are offered by systems on a network and could be attacked. There are various approaches for port scanning, all providing advantages and disadvantages. One of those approaches is the TCP Idle Scan, in which the attacker spoofs messages of a third computer in order to remain undetected. To see the results of the scan, he utilizes the IPID in the IPv4 header.

With the slowly approaching upgrade of IPv4 with IPv6, one will not be able anymore to conduct the TCP Idle Scan as previously, as the IPID is not included statically in the IPv6 header, but only when fragmentation is needed. Therefore, this thesis started with an investigation whether the TCP Idle Scan is still feasible in IPv6.

The investigation illustrated that an attacker can use ICMPv6 Echo Request messages with large amounts of data as well as ICMPv6 Packet Too Big messages specifying a MTU smaller than the IPv6 minimum MTU. This way, the idle host can be forced to use the IPv6 extension header for fragmentation, which contains an *identification* value which is comparable to the IPID in IPv4, in each IPv6 packet sent to a specific host. Applying this method, the TCP Idle Scan is feasible in IPv6.

After establishing how to conduct the TCP Idle Scan in IPv6, 21 different operating systems and versions have been analyzed regarding their properties as idle hosts. Among those, all nine tested Windows operating systems were suitable. This shows that the mistake to use predictable IPIDs in IPv4 is being repeated in IPv6.

Also two alternatives to the TCP Idle Scan in IPv4 as well as in IPv6, which do not rely on predictable assignment of the IPID or *identification* values have been analyzed. The first one is the RST Rate Limit Scan, which utilizes the fact that some idle hosts only allow a certain amount of TCP segments with the RST-flag per second. Another alternative is the SYN Cache Scan, which makes use of the limited amount of half-open TCP connections an idle host is able to store. Additionally, with the second alternative, it might also be possible for an attacker to scan through a firewall into the internal network.

To show that the presented port scanning methods can also be used in practice, a proof of concept has been created for each scan. Additionally, a patch for the security scanner *Nmap* was created, which already provided a very elaborated version of the TCP Idle Scan in IPv4. This patch enables the scanner to execute the TCP Idle Scan in IPv6. Compared to the TCP Idle Scan in IPv4, the created implementation decreased in performance by less than 1% while at the same time having less requirements to the idle host.

Acknowledgements

- I would like to thank my parents for enabling my studies in the Netherlands and their support.
- Fox-IT enabled me to do an internship at their company to write my thesis, for which I am thankful. They sent me to a conference and provided me with the hardware necessary to perform my research.
- Within Fox-IT, I would like to thank my supervisors Hendrik Schimmelpenninck van der Oye and Mark Koek for supervising me, Joachim Schipper for reviewing my patch as well as everybody else within the company who provided me with feedback.
- I would also like to thank Erik Poll, my supervisor from Radboud University, who provided me with academic feedback and useful critiques.
- Also, I am grateful for my proofreaders who helped me to improve the quality of my writing.

Contents

List of Figures

1. Introduction

When an attacker receives access to a network, it is important for him to see which systems are connected to this network, and which kind of services those systems are offering [13]. After having gathered this valuable information, the attacker can decide about his next target.

In order to find out which services are provided by a system, an often used technique is port scanning, as described for example by Erickson [13] or Zhang et al. [52]. In the usual approach for port scanning, connection requests are sent to various ports on a target computer to evaluate which services are running, and which are not [46].

However, this method is easy to detect and to be traced back to the attacker. To remain stealthier, different methods for port scanning exist, all providing advantages and disadvantages [29].

One of those methods is the TCP Idle Scan, which will be introduced in Section 2.3. With this port scanning technique, the attacker uses the help of a third computer, the so-called idle host, to cover his tracks. Most modern operating systems have been improved so that they cannot be used as idle host, but research has shown that the scan can still be executed by utilizing network printers [33].

Due to the slowly approaching upgrade of IPv4 with IP version 6 (IPv6), one will not be able anymore to conduct the TCP Idle Scan as previously. In IPv4, the attacker utilizes the *identification* field, mostly called IPID, in the IPv4 header to detect if a port on a target is opened or closed by triggering a message from the target to the idle host. Afterwards, the IPIDs of different answers from the idle host before and after the scan are compared to receive the result.

Originally, the IPID value is intended for fragmentation. In case a packet is too big to be transported over a network, the node will split it into parts, the so-called fragments. Each of those fragment has an IPID that indicates to which original packet the fragment belongs, which enables the receiver to correctly reassemble the original packet.

Compared to IPv4, no fragmentation is done by routers along the path anymore in IPv6, but only by the end nodes [10]. Hence, there is no IPID field in the IPv6 header, as it is not needed. Instead, IPv6 uses the optional extension header for fragmentation in case fragmentation is needed, which provides an *identification* field comparable to the IPID in IPv4.

This thesis started with an investigation to see whether the TCP Idle Scan could be transferred to IPv6, which turned out to be feasible. After giving an overview about the TCP Idle Scan in IPv4 in Section 2, Section 3 shows how the TCP Idle Scan can be transferred from IPv4 to IPv6, and which adjustments need to be made within this transformation. This is followed by an overview given in Section 4 of which types of systems assign their *identification* value of the optional extension header for fragmentation in IPv6 in a predictable and global way, and can therefore be used as idle hosts. Additionally, other noticeable findings in operating systems, such as bugs, are discussed. Section 4 also deals with the differences of the TCP Idle Scan in IPv4 and IPv6 regarding the amount of messages and the requirements for the idle host.

Section 5 discusses two alternatives to the TCP Idle Scan technique, which do not rely on a predictable assignment of the IPID or *identification* value, which are the RST Rate Limit Scan and the SYN Cache Scan. In Section 6, the proofs of concept created for the TCP Idle Scan in IPv6, the RST Rate Limit Scan and the SYN Cache Scan are discussed. Also, a patch for the security scanner *Nmap*, which already provided a very elaborated version of the TCP Idle Scan in IPv4, is introduced, which enables the tool to also execute the TCP Idle Scan in IPv6. Finally, it is discussed how the different types of port scanning methods can be prevented on the short term by system administrators, and on the long term by manufacturers of devices and operating systems, which is followed by the conclusion.

2. Background

This chapter explains the TCP Idle Scan technique within an IPv4 network, but begins by explaining some basics about the Internet Protocol version 4 (IPv4) as well as the Transmission Control Protocol (TCP), which are necessary to understand the technique.

2.1. IPv4

According to Tanenbaum, the Internet Protocol (IP) is “the glue that holds the whole Internet together” [48, Page 432]. The protocol is designed to deliver packets from its source to its destination, regardless if those are located in the same network.

A detailed description of the whole protocol would go beyond the scope of this thesis, it can be found for example in the work of Tanenbaum [48]. For this research, the important part of the protocol is the layout its packages use in the protocol’s currently most used version, IPv4, as shown in Figure 1.

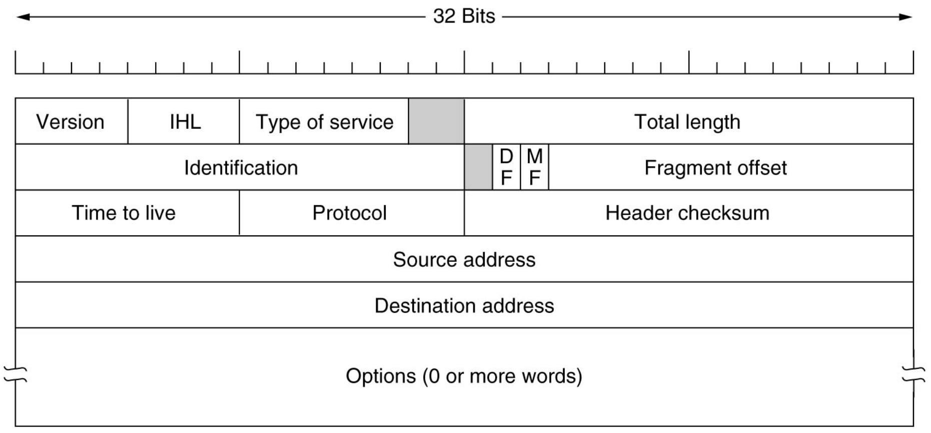
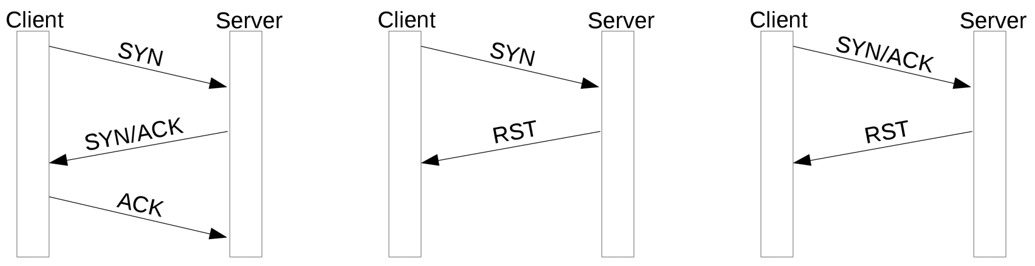


Figure 1: IPv4 (Internet Protocol) header [48]

Special attention should be paid to the 16 bit long *identification* field, also called IPID. If a packet needs to be split into parts, the so-called fragments, during transportation due to certain limitations such as hardware, operating system, protocols or other reasons, the IPID is used by the receiver to determine which parts of a packet belong together [48]. All fragments which belong to the same packet will have the same IPID, and a fragment offset field, which indicates at which place of the original packet the data of the fragment belongs. This process is called fragmentation.

The IPID value can be either assigned incrementally, randomly, or in any other way. Request for Comments (RFC) 791, “Internet Protocol” describes the possibilities for assigning the IPID value as follows:

> The choice of the Identifier for a datagram is based on the need to provide a way to uniquely identify the fragments of a particular datagram. The



(a) Successful (b) Unsuccessful (c) Unexpected

Figure 2: TCP three way handshake

> protocol module assembling fragments judges fragments to belong to the same datagram if they have the same source, destination, protocol, and Identifier. Thus, the sender must choose the Identifier to be unique for this source, destination pair and protocol for the time the datagram (or any fragment of it) could be alive in the internet.
>
> It seems then that a sending protocol module needs to keep a table of Identifiers, one entry for each destination it has communicated with in the last maximum packet lifetime for the internet.
>
> However, since the Identifier field allows 65,536 different values, some host may be able to simply use unique identifiers independent of destination [38, Page 28].

The RFC is not specific about assigning the IPID, and leaves the decision to the developer. In the end, most early operating systems decided to use a global counter for the assignment, while newer ones often use random counters [29]. Section 2.3 shows in which way any predictable assignment with a global counter for all hosts can be misused.

2.2. TCP three way handshake

The Transmission Control Protocol (TCP) is a reliable protocol which works on top of the IP protocol. To communicate via TCP, a connection must be established between two participants, usually referred to as client and server [39].

In order to establish this connection, a three way handshake is executed. First, the client sends a TCP segment which contains the control flag SYN (for synchronize) to the server. After receiving that, the server, when willing to open a connection, answers with a segment containing the control flag SYN- as well as ACK (for acknowledge). To finish the handshake, the client answers with a segment containing the ACK-flag. Now that the connection is established, data can be exchanged [39]. The procedure is illustrated in Figure 2a.

If the server does not want to open a connection, for instance because the service requested by the client is not available, it answers with a segment containing the flag RST (for reset), see Figure 2b. The handshake was unsuccessful.

When a segment with a SYN- and ACK-flag is received without having previously sent one with the SYN-flag, a host will also answer with a segment containing the control flag RST, as shown in Figure 2c [39].

2.3. TCP Idle Scan in IPv4

In order to find out which services are provided on a specific host and can be abused without revealing his own IP address, an attacker can execute the TCP Idle Scan, which makes use of the properties of IP and TCP mentioned in the previous sections. The TCP Idle Scan (or zombie scan) is a stealthy port scanning method, which allows an attacker to scan a target without the need of sending a single IP-Packet which contains his own IP address to the target. Instead, he uses the IP address of a third host, the idle host (also known as zombie host) for the scan. To be able to retrieve the results from the idle host, the attacker utilizes the IPID value in the IPv4 header, mentioned in Section 2.1.

While the idea of this technique was first introduced by Salvatore Sanfilippo in 1998 [43], the first tool for executing the scan was created by Filipe Almeida in 1999 [1].

The technique is described by Lyons [27] as follows:

1. To scan a port on the target host, the attacker first sends a TCP segment with the SYN- and ACK-flag to the idle host.

2. As the host is not expecting this segment, it will answer with a segment containing the RST-flag, and also its IPID.

3. Afterwards, the attacker sends a segment with the SYN-flag to the target host, dedicated to the port he wants to scan, and sets as source IP-address the spoofed IP-address of the idle host. Due to this spoofed address, the target will answer to the idle host, not to the attacker.

4. If the port is closed, a segment with a RST-flag will be sent to the idle host (4a). If the scanned port on the target is open, the host will send a segment containing the SYN- and ACK-flag (4b) to continue the TCP three-way handshake (Section 2.2).

5. In case of receiving a segment with a RST-flag, the idle host will not execute further actions. But if a segment with the SYN- and ACK-flag is received, it will respond by sending a segment with the RST-flag, as the previous one was not expected. For this answer, the host will use its next available IPID.

6. To get the result of the scan, the attacker sends now again a segment with the SYN- and ACK-flag to the idle host.

7. The idle host answers to the received segment with a segment containing the RST-flag and an IPID. In case the port is closed, the received IPID will have increased once compared to the previously received IPID, while for an open port, it will have increased twice.

Figure 3 shows the TCP Idle Scan in IPv4 in a schematic manner.

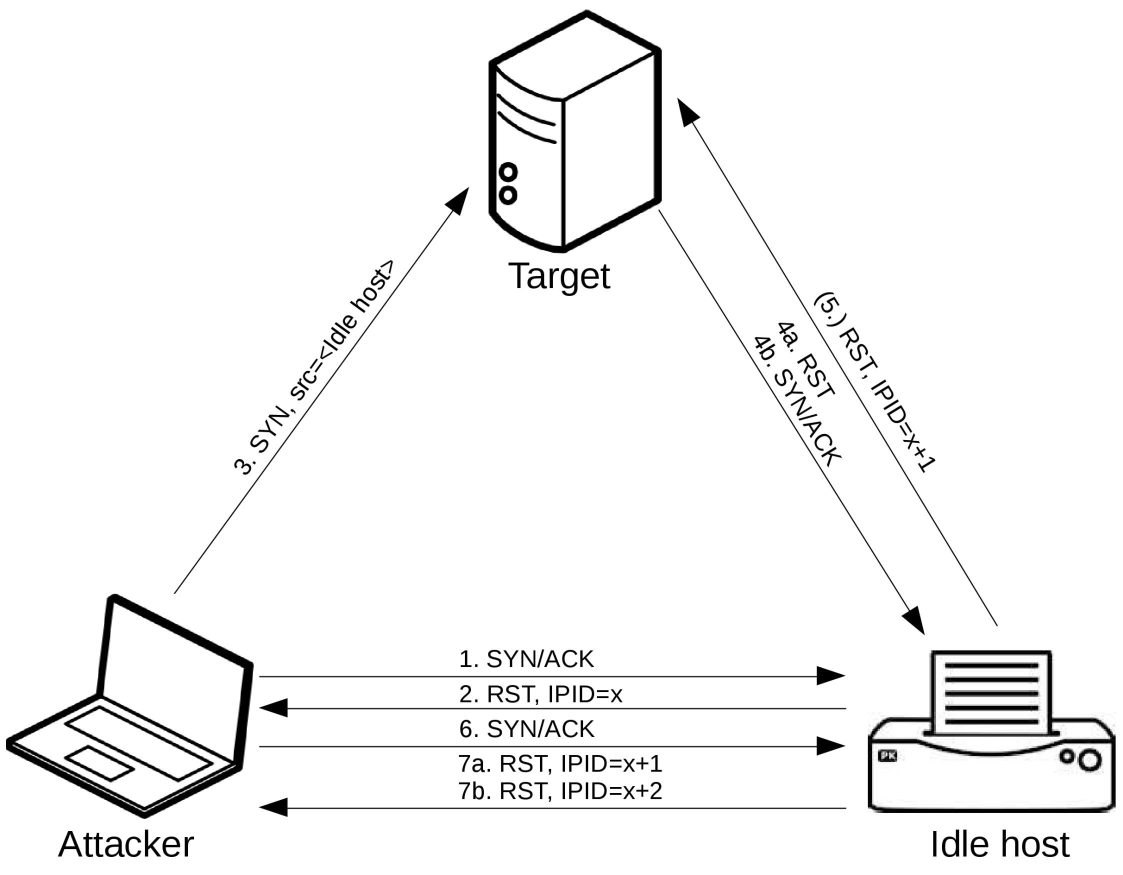


Figure 3: TCP Idle Scan in IPv4

2.4. Existing research into the TCP Idle Scan (for IPv4)

While there is scientific literature on how to detect TCP Idle Scans [12] [47], literature on the scan itself remains very limited. The best source of information is the book "Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning", written by Lyon [27], the author of the port scanner *Nmap* [28]. Most contents of the book were also published in the free "Nmap reference guide" [29]. In the book "Hacking: the art of exploitation" [13], Erickson also gives an explanation of the TCP Idle Scan using *Nmap*.

Another publication about the TCP Idle Scan has been made by Kamerling [25]. Unlike Lyon and Erickson, this document makes use of the network security tool *Hping* [44], written by Sanfilippo. Kamerling also refers to Bellovin [3], who mentioned the danger of predictable sequence numbers already in 1989. Although Bellovin was covering TCP, the attack introduced by him was also enabled by predictable sequence numbers.

Based on this literature, the author of this thesis demonstrated that the TCP Idle Scan, although discovered 15 years ago, is still feasible in modern networks as long as the right device is used as idle host [33]. The research tested 38 different types of network printers for the assignment of their IPID in the IPv4 header, of which 34 turned out to be suitable as idle hosts because of predictable and globally assigned values.

In comparison to the TCP Idle Scan, sufficient literature can be found on IPv6. The specifications of the protocol are described in Request for Comments (RFC) 2460 [10], and information about the Internet Control Message Protocol for IPv6 (ICMPv6) can be found in RFC 4443 [8]. Hagen dedicated a whole book to IPv6 [18], and another chapter about IPv6 was written by Tanenbaum [48]. For the advanced reader, Davies [9] gives additional information about various implementation details regarding Microsoft Windows. Some of this information will be utilized in Section 4, which discusses the behavior of different operating systems. In addition to the available literature, up to date information is exchanged in various conferences, such as "IPv6 World Congress" [23], the "V6 World Congress" [49] and the "IPv6-Kongress" [19], which are held regularly.

For porting the TCP Idle Scan to IPv6, the RFCs "Transmission Control Protocol" [39], "The TCP Maximum Segment Size and Related Topics" [40] and "Path MTU Discovery for IP version 6" [30] provide useful details and definitions.

3. Applying the TCP Idle Scan in IPv6

3.1. Differences to IPv4

The Internet Protocol version 6 (IPv6) introduces a lot of changes compared to version 4 (IPv4). Discussing all those changes would exceed the dimension of this thesis. Because of that, this section will only analyze the differences that affect the TCP Idle Scan. Additional information about all changes between IPv4 and IPv6 can be found in [18] and [48].

Figure 4 shows the IPv6 header. Compared to the IPv4 header shown in Figure 1 on page 9, the most noticeable changes are that apart from the version as well as the source and destination fields, all fields are different. Also, the source and destination fields increased in size in order to be able to store IPv6 addresses.

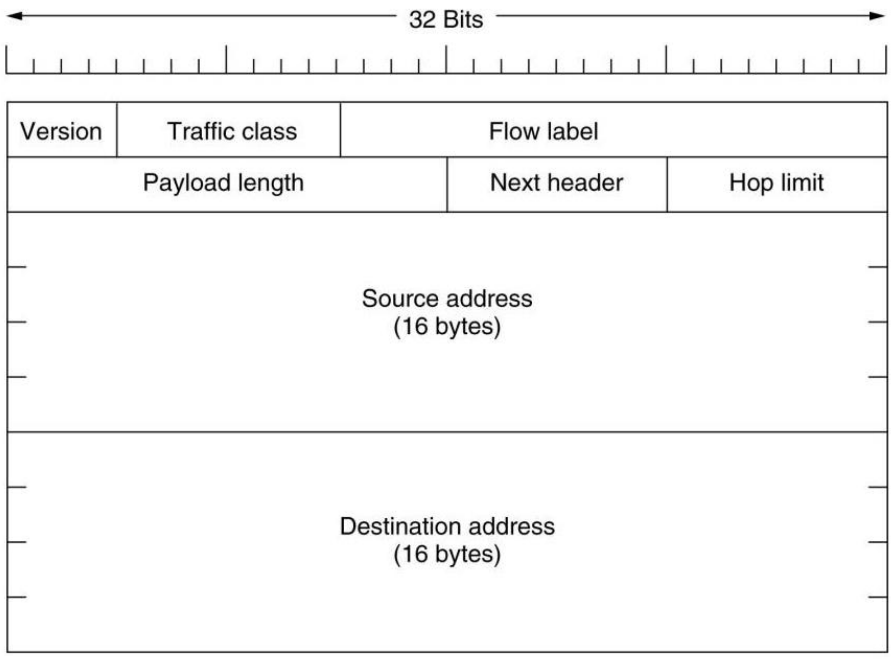


Figure 4: IPv6 (Internet Protocol) header [48]

One of the fields that has been removed is the IPID field, which is used for the TCP Idle Scan in IPv4. In IPv6, fragmentation is only done by end nodes [10]. If a node on the path receives a packet which is too big, an Internet Control Message Protocol version 6 (ICMPv6) Packet Too Big message [8] is sent back to the source address, notifying it about the Maximum Transfer Unit (MTU) of the node on the path. This ICMPv6 message piggybacks as much as possible of the originally received packet, which caused the ICMPv6 message to be sent, without exceeding the minimum IPv6 MTU. After receiving such a message, the source node now fragments its packets with a maximum size of the

MTU received in the ICMPv6 message. This whole process is done to unburden nodes on the path, and makes the permanent use of an IPID field for reassembling received packets unnecessary.

For this reason, the IPID field has been removed in the IPv6 header. If the sending host needs to fragment a packet, it uses the extension header for fragmentation, also called fragmentation header [10]. Such an extension header is placed after the IPv6 header and followed by the upper-layer header, for instance a TCP header. Therefore, a fragmented packet contains at first the usual IPv6 header, afterwards the IPv6 extension header for fragmentation, and then the payload. Besides the extension header for fragmentation, IPv6 also provides extension headers for Hop-by-Hop options, Routing, Destination options, Authentication and the Encapsulating Security Payload. More information about extension headers can be found in RFC 2460 [10].

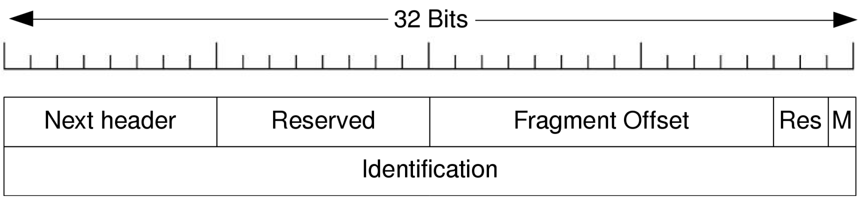


Figure 5: IPv6 extension header for fragmentation (based on RFC 2460 [10])

Figure 5 shows the extension header for fragmentation. The *next header* field states the protocol of the following header in the original packet, such as TCP [10]. Afterwards, the following eight bits are reserved, and currently not in use. The *fragmentation offset* specifies the offset of this fragment in the complete packet. After this value, there are again two bits reserved, which are followed by the *m* or *more fragments* flag, which states if a received fragment was the last one, or if there will be more fragments.

The last field in the extension header for fragmentation is the 32 bit long *identification* field, which has equally to IPv4 the purpose of identifying which fragments belong together. Unlike the IPID in IPv4, this value is usually referred to with no specific name. This method will also be used within this thesis to differ the *identification* values from IPv4 and IPv6. Compared to the IPID in IPv4, the size of the *identification* field in IPv6 doubled. RFC 2460 states the following about the *identification* value:

> The Identification must be different than that of any other fragmented packet sent recently* with the same Source Address and Destination Address. If a Routing header is present, the Destination Address of concern is that of the final destination [10, Page 18].

Among others, the asterisk gives the following information:

> it is assumed that the requirement can be met by maintaining the Identification value as a simple, 32- bit, "wrap-around" counter, incremented each time a packet must be fragmented. It is an implementation choice whether

> to maintain a single counter for the node or multiple counters, e.g., one for each of the node's possible source addresses, or one for each active (source address, destination address) combination [10, Page 19].

As stated by the RFC, the method of assigning the value in the *identification* field is a choice of implementation, and it is therefore hard to predict how a specific system will behave. If the value is assigned on a per-host-basis, the TCP Idle Scan is impossible with the device being used as idle host, since the attacker would not be able to detect if a packet has been sent from the idle host to the target by analyzing the *identification* value.

The description given for the *identification* value resembles the one describing the IPID in IPv4, as shown in Section 2.1. Although having the possibility to choose, most devices implemented a global counter for the IPID in IPv4 [29]. Additionally, one can see that the RFC for IPv6 suggests the use of a simple incremented counter, which plays an important role in making the TCP Idle Scan feasible in IPv6. Section 4.1 gives an overview over the tested systems and their assignment strategies.

3.2. Forcing fragmentation

To be able to execute the TCP Idle Scan in IPv6, all three participating hosts must use the extension header for fragmentation in their outgoing packets. Still, a closer look at Figure 2.3 though shows that it is not necessary to use the extension header in all seven steps of the scan. For the attacker, it is only relevant whether step 5, sending the TCP segment with the RST flag, has been executed by the idle host or not. Therefore, the use of the extension header for fragmentation needs to be forced only in step 5, for packets being sent from the idle host to the target.

Additionally, it will also be necessary that the idle host uses the extension header in step 2 and 7 in order for the attacker to be able to compare the received *identification* values. Enforcing the extension header in those steps is though easier for the attacker, as they do not require the use of TCP, and also because the attacker is directly participating in the conversation.

3.2.1. Forcing fragmentation of step 2 and 7 using ICMPv6 Echo Requests

The Internet Control Message Protocol (ICMP) is an example for a protocol that has been adapted in order to fit IPv6, and is therefore in its new version known as ICMP for IPv6 (ICMPv6), defined in RFC 4443 [8]. It is a user of the IPv6 protocol and provides various possibilities to debug a network connection. Its most common message is the ICMPv6 Echo Request, also known as "ping". When an Echo Request is received, a host will answer with a ICMPv6 Echo Response. For this response, the RFC states the following:

> The data received in the ICMPv6 Echo Request message MUST be returned entirely and unmodified in the ICMPv6 Echo Reply message.

For the attacker, this means that an ICMPv6 Echo Request to the idle host, which contains enough data that it needs to be fragmented, will also be returned in fragments, and contain an extension header for fragmentation with the *identification* value. Therefore, the attacker can send ICMPv6 Echo Requests containing a large amount of data in step 1 and 6 in order to receive answers containing an extension header for fragmentation in steps 2 and 7.

3.2.2. Forcing fragmentation of step 5

Forcing fragmentation in step 5 in the TCP Idle Scan is more complex. In order to scan for open ports, it is necessary to use TCP on top of the IPv6 header. For a first experiment, a TCP segment with a SYN- and ACK-flag was sent to the idle host, which contained 1800 bytes of data as a payload, hoping that the data would be returned in the segment with the RST-flag, being big enough to require fragmentation. However, in the response of the idle host, the payload was ignored and therefore no extension header for fragmentation was used by the tested idle hosts running Linux 3.x and Windows 7. For this reason, a similar approach to the ICMPv6 Echo Requests, which returns the received data, can be excluded. Also, sending the TCP segment in fragments did not lead to a fragmented answer of the idle hosts.

The solution to this problem can be found in RFC 1981, "Path MTU Discovery for IP version 6". As mentioned previously, fragmentation is usually requested from a node by sending an ICMPv6 Packet Too Big message as answer to a received, but too large IPv6 packet. This ICMPv6 message piggybacks as much of the original packet as possible, and also contains the MTU a node can process. The host which sent the original package will set the MTU in the ICMPv6 message as the new Path MTU (PMTU), and will from now on fragment packets bigger than this PMTU. If an attacker is now able to spoof this ICMPv6 message, changing of another host's PMTU is possible.

However, spoofing such a message still does not seem enough to achieve fragmentation of step 5 in the TCP Idle Scan, which is the host answering the target with a TCP-datagram containing the RST-flag. This datagram will have a maximum size of 60 bytes, assuming an IPv6 header of 40 bytes and a TCP header of 20 bytes [39] [40]. Considering that the minimal MTU of IPv6 is 1280 bytes [10], the answer of the idle host should never be fragmented under normal circumstances.

What still enables the attacker to force the extension header for fragmentation in the answer from the idle host to the target is RFC 1981. This states:

> When a node receives a Packet Too Big message, it MUST reduce its estimate of the PMTU for the relevant path, based on the value of the MTU field in the message [30, Page 3].

as well as:

> A node MUST NOT reduce its estimate of the Path MTU below the IPv6 minimum link MTU. Note: A node may receive a Packet Too Big message reporting a next-hop MTU that is less than the IPv6 minimum link MTU. In

that case, the node is not required to reduce the size of subsequent packets sent on the path to less than the IPv6 minimum link MTU, but rather must include a Fragment header in those packets [30, Page 4].

Therefore, receiving an ICMPv6 Packet Too Big message with a MTU smaller than 1280 bytes, which is the smallest IPv6 MTU [10], causes a host to append the extension header for fragmentation to all its IPv6 packages to a certain host. This behavior is described in RFC 6946 as "atomic fragments" [16]. Although this extension headers are mostly empty, they contain the *identification* field, which is the only relevant field for the attacker.

3.2.3. Spoofing ICMPv6 Packet Too Big Messages

As mentioned in Section 3.2.2, a host can be forced to use the extension header for fragmentation for each packet to a specific destination. This behavior is called atomic fragments. Therefore, in order to execute the TCP Idle Scan in IPv6, it is necessary to send an ICMPv6 Packet Too Big message from the target to the idle host. The header of such a message is described in RFC 4443 [8] and shown in Figure 6.

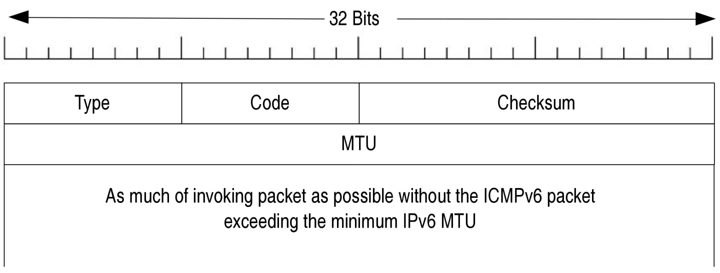

32 Bits
Type
Code
Checksum
MTU
As much of invoking packet as possible without the ICMPv6 packet exceeding the minimum IPv6 MTU

Figure 6: ICMPv6 Packet Too Big message header (based on RFC 4443 [8])

The fields *type*, *code* and *checksum* are defined by RFC 4443. In the *MTU* field, it is possible to specify the MTU the sending host can process. A MTU smaller than the minimum MTU of 1280 bytes will cause the receiving host to use atomic fragments for all IPv6 packets to the sending host. After setting those fields, the RFC states that the space left in the packet without exceeding the minimum MTU of IPv6 should be filled with the invoking packet, which is the too big IPv6 packet that caused the ICMPv6 Packet Too Big message to be sent.

When an attacker now wants to spoof such a message in order to force the idle host to use atomic fragments, the degree of complexity of the task depends on the host which is to manipulate.

- Linux kernels up to 3.8, which was the most recent stable kernel at the time of writing, accept any spoofed ICMPv6 Packet Too Big Message without prior established communication and change the Path MTU (PMTU) accordingly. Besides that, different kernel versions also provide different bugs, which will be shown in Section 4.1.

- Hosts running for example Windows 7 as operating system are more precise and only accept incoming ICMPv6 Packet Too Big Messages where packets have been sent earlier and exactly those packets are piggybacked by the ICMPv6 message.

One possibility to spoof an ICMPv6 Packet Too Big message for a Windows 7 host is to send an ICMPv6 Echo Request with the source address of the target to the idle host which contains enough data that it needs to be fragmented. The idle host will send an ICMPv6 Echo Response to the target, which will not respond to this unexpected message, as responses to responses are not scheduled, even if they are unexpected [8]. Afterwards, the attacker can spoof an ICMPv6 Packet Too Big message from the target to the idle host, informing it about the target's new MTU. The composition of this message is shown in Figure 7.

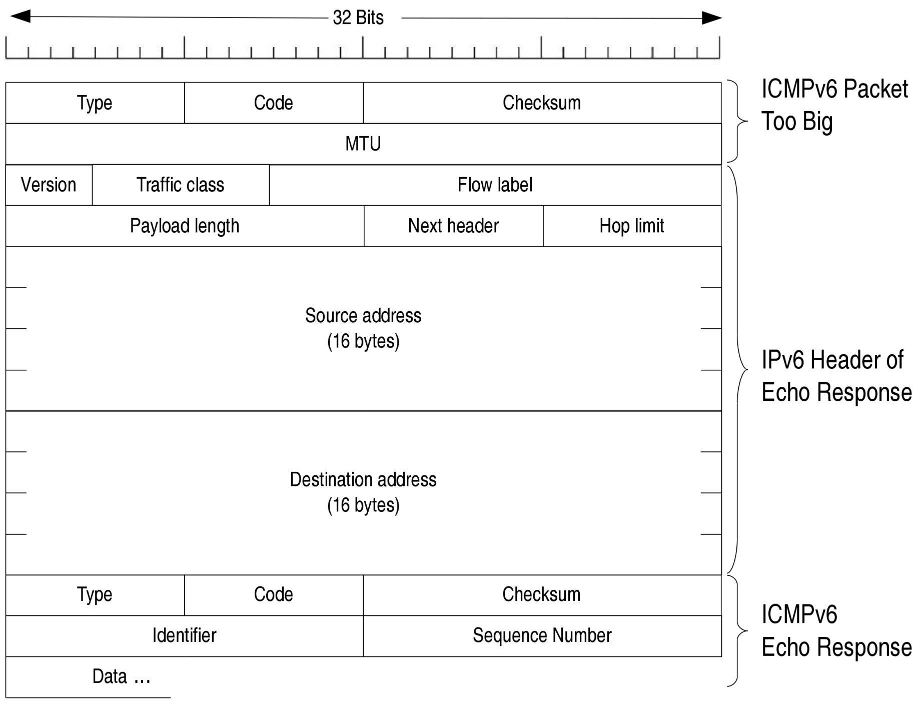


Figure 7: ICMPv6 Packet Too Big Message to spoof

The message consists of the ICMPv6 Packet Too Big header, which piggybacks as much as possible of the original message which caused the ICMPv6 Packet Too Big message to be sent, which is in this case the ICMPv6 Echo Response and its IPv6 header. In order to spoof the message, first the ICMPv6 Packet Too Big header with an MTU smaller than the minimum IPv6 MTU is created. The payload of this message needs to be filled with as much of the ICMPv6 Echo Response from the idle host to the target as possible without exceeding the minimum IPv6 MTU of 1280 bytes.

Although the attacker is not able to see the ICMPv6 Echo Response , its contents can be assumed: On the IPv6 layer, the *version* will state "IPv6", and with the exception of special cases, the *traffic class* and the *flow label* will be zero [10]. The *payload length* can be calculated according to RFC 2460 [10], and the *next header* field will state "ICMPv6". For the *hop limit*, the only values established while testing Linux and Windows hosts were 64 and 128, this value can therefore be guessed. The remaining fields the *source address* and *destination address*, which are known by the attacker as well.

At the ICMPv6 layer for the old ICMPv6 Echo Response, the fields for *type* and *code* are known, and the *checksum* can be calculated [8]. The *identifier* as well as the *sequence number* were both zero with Windows 7 and Windows 8, and the field for the data contains the values being created in the spoofed ICMPv6 Echo Request previously.

Knowing how to reconstruct the ICMPv6 Echo Response from the idle host to the target, an attacker is able to create a spoofed ICMPv6 Packet Too Big message from the target and send it to the idle host, causing it to use atomic fragments in all its IPv6 packets addressed to the target.

3.3. TCP Idle Scan in IPv6

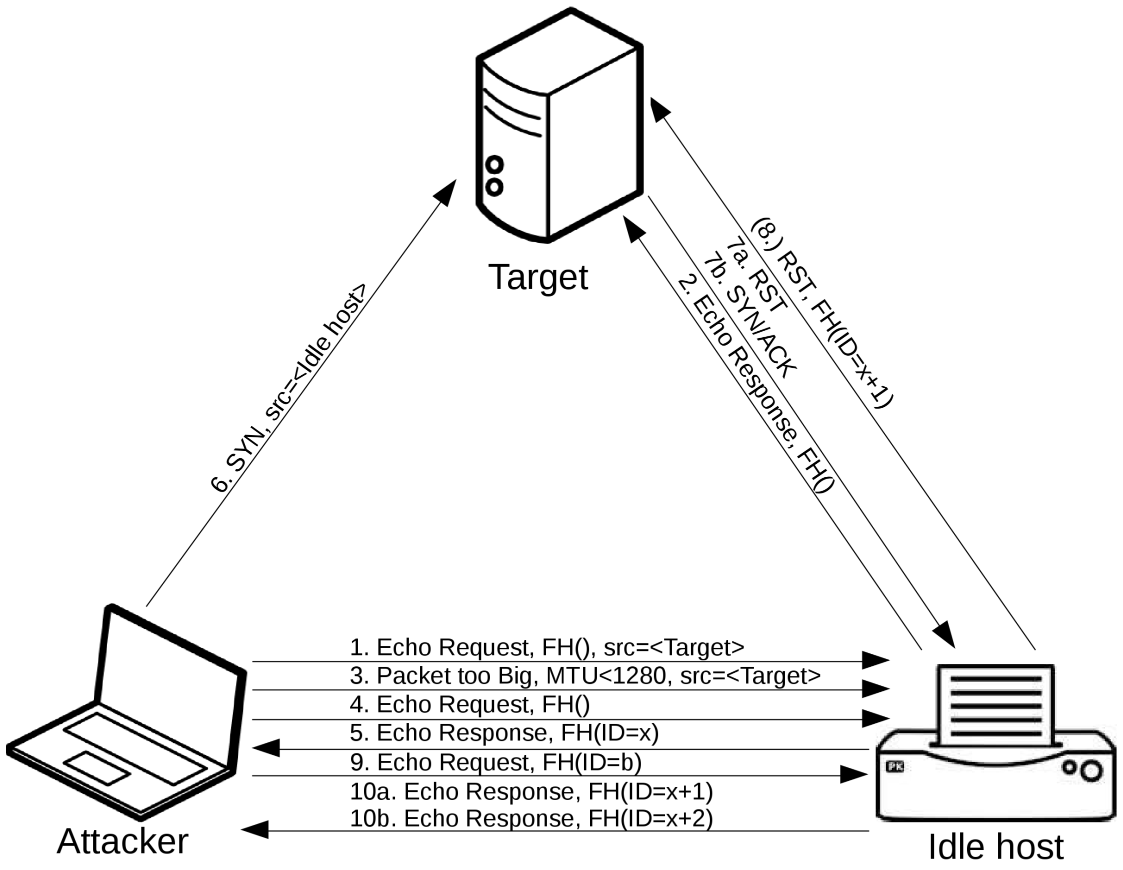
With the knowledge gained in the previous sections, it is now possible to update the TCP Idle Scan in IPv4 described in Section 2.3 so that it can be used in IPv6. The procedure includes the full message sequence mentioned in Section 3.2.3, and therefore can be used for idle hosts only accepting ICMPv6 Packet Too Big messages with prior traffic as well as for those accepting the messages without on prior traffic. Figure 8 gives an overview over the attack.

1. At first, the attacker sends a spoofed ICMPv6 Echo Request ("ping") to the idle host, with the source address of the target, which causes the idle host to answer to the target. This Echo Request contains enough data that the request needs to be fragmented, therefore the idle host will use an extension header for fragmentation (FH) in its response. The *identification* value in the extension header used by the attacker is not relevant.

2. The idle host will answer with an ICMPv6 Echo Response directed to the target. This response contains all the data received in the request, and therefore also needs to be fragmented. The *identification* value in the extension header in this message is not relevant for the scan.

3. Despite receiving the ICMPv6 Echo Response, the target will not answer to the idle host, as responses are not answered [8]. Instead, the attacker spoofs an ICMPv6 Packet Too Big message with a MTU smaller than the IPv6 minimum MTU with the source address of the target (as described in Section 3.2.3) and sends it to the idle host. Therefore, the idle host will now append an extension header for fragmentation to all IPv6 packets sent to the target, even if there is no need to fragment the data.

4. Similar to the TCP Idle Scan in IPv4, the purpose of the next two steps is to obtain the idle hosts currently used *identification* value in the extension header for



Target
6. SYN, src=<Idle host>
(8.) RST, FH(ID=x+1)
7a. RST
7b. SYN/ACK
2. Echo Response, FH()
1. Echo Request, FH(), src=<Target>
3. Packet too Big, MTU<1280, src=<Target>
4. Echo Request, FH()
5. Echo Response, FH(ID=x)
9. Echo Request, FH(ID=b)
10a. Echo Response, FH(ID=x+1)
10b. Echo Response, FH(ID=x+2)
Attacker
Idle host

Figure 8: TCP Idle Scan in IPv6

fragmentation. For this, the attacker sends an ICMPv6 Echo Request to the idle host which contains enough data that it needs to be fragmented. The *identification* value in the extension header used by the attacker is not relevant for the scan.

5. After receiving the ICMPv6 Echo Request, the idle host will answer with an ICMPv6 Echo Response, which contains all the data received in the ICMPv6 Echo Request, and therefore will also be fragmented. The *identification* value in the extension header is stored by the attacker.

6. Now, equally to the TCP Idle Scan in IPv4, the attacker sends a spoofed TCP segment with the SYN-flag to the target on the port he wants to scan, using as source address the address of the idle host. This causes the target to answer to the idle host.

7. If the port on the target is closed, a segment with a RST-flag will be sent to the idle host (7a). If the scanned port on the target is open, the host will send a segment containing a SYN- and an ACK-flag (7b).

8. In case of receiving a segment with a RST-flag, the idle host will not execute further actions. But if a segment with the SYN- and ACK-flag is received, it will answer by sending a segment with the RST-flag (8), as the received segment with the SYN- and ACK-flag was not expected. As the idle host appends an extension header for fragmentation to all packets being sent to the target because of step 3, it will append an empty extension header for fragmentation, using its next available value for the *identification* field.

9. In order to request the result of the scan, the attacker now repeats step 4, sending an ICMPv6 Echo Request with enough data so that it needs to be fragmented, to the idle host.

10. The received ICMPv6 Echo Response will now be analyzed by the attacker regarding its *identification* value in the extension header for fragmentation. If it incremented once compared to the one stored in step 5, it can be reasoned that the idle host did not send a segment with an RST-flag, therefore the scanned port on the target is closed (10a). If the value incremented twice compared to the one stored in step 5, the idle host had to send a segment, and therefore the port on the target is open (10b).

3.4. Requirements for the idle host

When using the TCP Idle Scan in IPv4, the requirements for the idle host are clearly defined [33]:

1. The IPID values assigned by the idle host need to be predictable. Also, the generation of the value should be done globally, and not on a per-host-basis. Otherwise, the attacker will not be able to recognize via the IPID if a packet was sent to a different host.

2. The idle host should not create traffic by itself. This would disturb the scanning process. (Hence the name idle host.)

Another rule for the idle host in IPv4, which was not mentioned in [33], depends on the network architecture. In order to execute the TCP Idle Scan in IPv4, no node on the networking path between the idle host and the attacker should exchange the IPv4 header and with it its IPID. If this is done for example by a router, an attacker is not able to access the idle host's IPID, making the scan unfeasible. Therefore, a third rule for the TCP Idle Scan in IPv4 should be declared as follows:

3. No node on the path between the idle host and the attacker should exchange the IPv4 header. This would prevent the attacker from accessing the idle host's IPID value.

When the TCP Idle Scan is used in IPv6, the first requirement remains the same. In order for an attacker to see if the idle host answered the target with a TCP segment containing the RST-flag, the *identification* value needs to be assigned in a global and predictable way also in IPv6. What changes is the second requirement, which requires the idle host to be idle.

In IPv4, every sent packet contains the IPID value, whether it is actually fragmented or not. Compared to this, the *identification* value in IPv6 is only used and increased when a package with an extension header for fragmentation is sent, as stated by RFC 2460 [10, Page 17]:

> For every packet that is to be fragmented, the source node generates an Identification value.

Gilad and Herzberg [15] present a statistic according to which only 0.11% of the IPv6 packets they analyzed were fragmented, and 0.15% of all IPv4 packets. This statistic shows that the amount of fragmented packets, which would disturb the scanning process of the TCP Idle Scan in IPv6, is nearly negligible for average network traffic.

On the other hand, it should not be forgotten that the idle host attaches an IPv6 extension header for fragmentation to every IPv6 packet which is sent to the target after once receiving the spoofed ICMPv6 Packet Too Big message with an MTU smaller than the minimum IPv6 MTU. This includes for example ICMPv6 Neighbor Discovery messages, which will be exchanged between the idle host and the target if both are in the same network to announce their presence [35]. Besides those messages, also other IPv6 traffic from the idle host to the target will influence the *identification* value and therefore disturb the scanning process.

To summarize, it can be said that the percentage of traffic which would affect the TCP Idle Scan in IPv6 remains rather limited. Most of all, no IPv6 communication besides the port scan should be done from the idle host to the target. Reaching these results, the second requirement for the idle host should be changed in IPv6 as follows:

2. The idle host should neither create fragmented IPv6 traffic nor IPv6 traffic to the target by itself. This would disturb the scanning process.

Compared to IPv4, this second requirement for the idle host is less limiting. In IPv4, the idle host was not allowed to create traffic at all due to the IPID value being a static part of the IPv4 header, and being included for each single packet. In IPv6, the limitations are mostly on communication only between the idle host and the target because the *identification* value is not used for each single IPv6 packet, but only when needed. While executing the TCP Idle Scan in IPv6, an idle host communicating with a fourth party via IPv6 would not disturb the scanning process, as long as the IPv6 packets being sent from the idle host to the fourth party do not use the IPv6 extension header for fragmentation. IPv6 packets from the fourth party to the idle host can use the extension header though, as this does not influence the counter for the *identification* value of the idle host.

Another requirement which changes by using the TCP Idle Scan in IPv6 is requirement 3, which does not allow nodes on the path to change the IP header. In IPv6, the *identification* value is not located in the main IPv6 header, but in the IPv6 extension header for fragmentation. Along the delivery path of an IPv6 packet, this extension header will not be accessed or changed by nodes on the path [10]. Therefore, requirement number 3 does not apply for the TCP Idle Scan in IPv6.

To summarize, the requirements for the idle host when executing the TCP Idle Scan in IPv6 are the following:

1. The *identification* values assigned by the idle host need to be predictable. Also, the generation of the value should be done globally, and not on a per-host-basis.
2. The idle host should neither create fragmented IPv6 traffic nor IPv6 traffic to the target by itself.

3.5. Increasing stealth

In order to increase the stealth of the TCP Idle Scan in IPv6, certain steps can be modified or even omitted in some scenarios.

- The first detail which can be further improved is located in step 3, sending the spoofed ICMPv6 Packet Too Big message. This message contains the MTU for this path, which will be applied by the idle host. RFC 1981 states the following about the PMTU:

  > The PMTU of a path may change over time, due to changes in the routing topology. Reductions of the PMTU are detected by Packet Too Big messages. To detect increases in a path's PMTU, a node periodically increases its assumed PMTU. This will almost always result in packets being discarded and Packet Too Big messages being generated, because in most cases the PMTU of the path will not have changed. Therefore, attempts to detect increases in a path's PMTU should be done infrequently. [30, Page 3]

  Because of this, the decision on which MTU to be set in the ICMPv6 Packet Too Big message in step 3 should be made carefully. Useful details to be aware of in this step are the number of ports being scanned, the duration of scanning a single port as well as the time span in which the idle host attempts to increase the PMTU and by which value the PMTU is increased. Choosing this value well, steps 1 to 3 in the TCP Idle Scan in IPv6 may only be necessary to be executed once while scanning multiple ports. Ideally, the idle host even attempts to increment the PMTU shortly after the scan to a value higher or equal to the IPv6 minimum MTU, causing the idle host to stop appending the extension header for fragmentation on the path and leaving less evidence of the occurred scan. By carefully choosing the MTU value and omitting steps 1 to 3 for scanning multiple ports, the number of messages necessary can be reduced from ten to seven, the same as for the TCP Idle Scan in IPv4. On Linux and Windows, the period for trying to increase the PMTU is set to 10 minutes by default [9] [42].

- The next steps which will be analyzed are 4 and 5, sending an ICMPv6 Echo Request to the idle host to receive its currently used *identification* value in the extension header for fragmentation. Those are executed in order to be able to compare the value later with the one received in the steps 9 and 10. If multiple ports are scanned by the attacker, steps 4 and 5 can be omitted for all but the first port, as the *identification* value is already known from the steps nine and ten while scanning the previous port. This reduces the number of messages necessary to execute the scan further from seven to five when multiple ports are scanned.

- Another improvement that can be made affects steps 9 and 10, as well as 4 and 5 if executed. In these steps, fragmented ICMPv6 Echo Requests and Echo Responses are sent to determine the *identification* value in the extension header for fragmentation. To ensure that the idle host answers with a fragmented Echo Request, a relatively high amount of data, which has no effective use, has to be sent. For example in a local Ethernet network, an IP packet of 1500 bytes can be sent without needing fragmentation [21], thus forcing the ICMPv6 Echo Request to have at least 1501 bytes.

  Compared to this, an IPv6 Packet with an extension header for fragmentation but no data in the ICMPv6 Echo Request produces an IPv6 Packet of 48 bytes, being 40 bytes for the IPv6 Header, and eight bytes for the ICMPv6 header [8]. This is a reduction of 97%. In order to still force the idle host to use the extension header for fragmentation, the attacker can spoof an ICMPv6 Packet Too Big message as explained in Section 3.2.2 so that the idle host also uses atomic fragments on the path to the attacker. This step can be executed after step 5, and the message of step 5 can be piggybacked by the ICMPv6 Packet Too Big message. The size of the reduced ICMPv6 Echo Response is 56 bytes, as it also includes eight bytes for the extension header for fragmentation. This is still a reduction of 96%. It is also to mention that after applying the procedure to receive atomic fragments, every other message using IPv6 Packets can be chosen, as for example a Neighbor Solicitation message [35].

While this procedure might not be advisable if only one port should be scanned since it requires sending additional messages, it is clearly advisable if multiple ports are scanned as it decreases the network traffic drastically, and peaks in network traffic are conspicuous [52].

Figures 9 and 10 show the TCP Idle Scan in IPv6 with all three improvements of this section applied. In Figure 9, which shows scanning of the first port on a target, one step was added after step 5, the sending of the ICMPv6 Packet Too Big message with the modified MTU. Therefore, the ICMPv6 Echo Request in step 10 does not need to be fragmented anymore, because the idle host uses atomic fragments at this point. This leads to a decrease in the size of the messages in step 10 and 11.

For scanning another port, the steps 1 to 6 can be omitted, as shown by Figure 10. In total, although while scanning the first port the amount of messages increased by one to eleven, it decreased by half for scanning further ports in Figure 10. Another achievement was a drastic reduction of the network traffic.

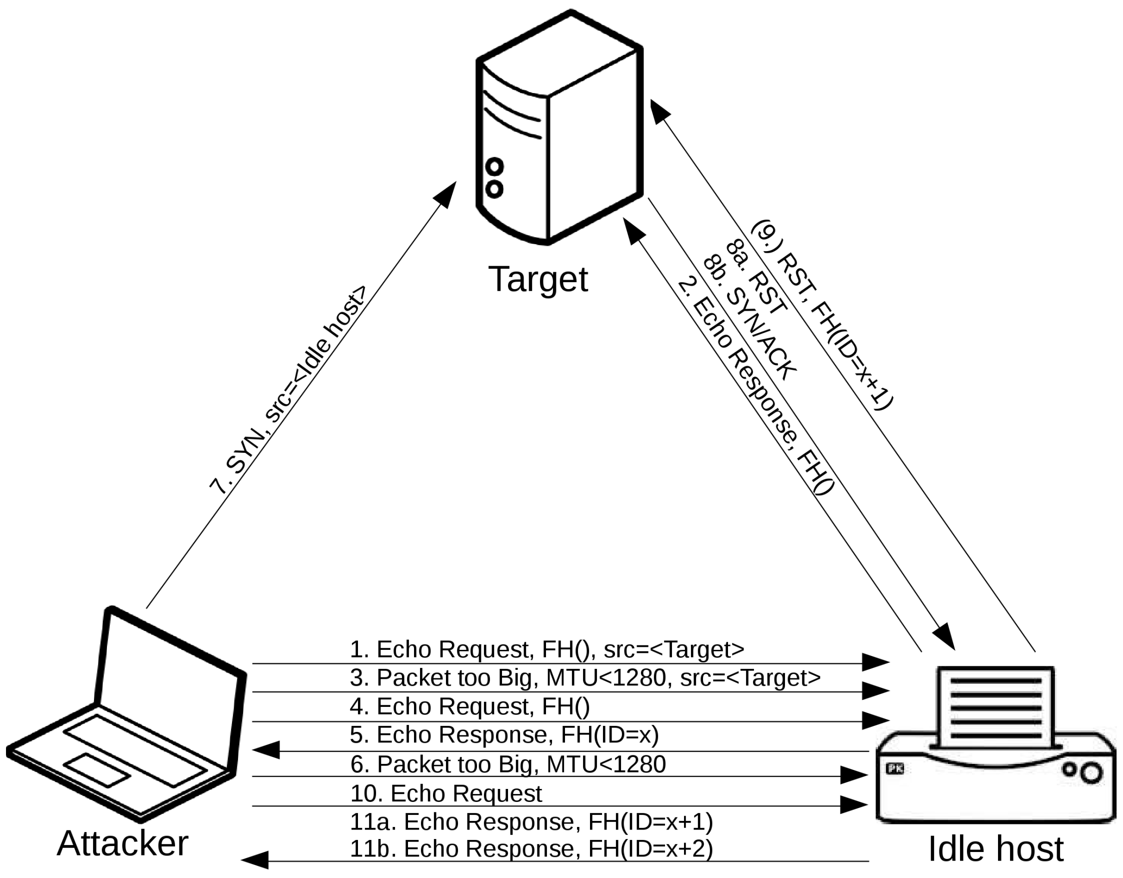

Target
7. SYN, src=<Idle host>
(9.) RST, FH(ID=x+1)
8a. RST
8b. SYN/ACK
2. Echo Response, FH()
1. Echo Request, FH(), src=<Target>
3. Packet too Big, MTU<1280, src=<Target>
4. Echo Request, FH()
5. Echo Response, FH(ID=x)
6. Packet too Big, MTU<1280
10. Echo Request
11a. Echo Response, FH(ID=x+1)
11b. Echo Response, FH(ID=x+2)
Attacker
Idle host

Figure 9: Scanning the first port with the improved TCP Idle Scan in IPv6

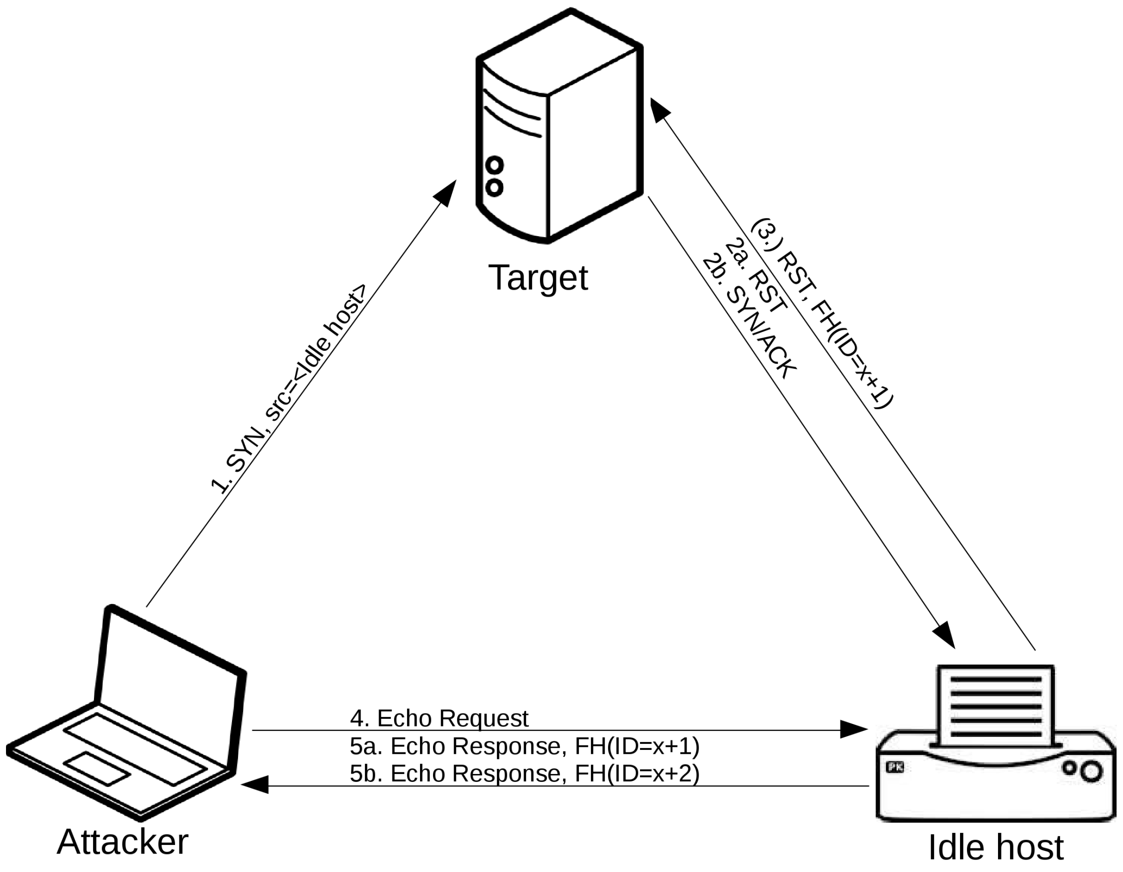


Figure 10: Scanning another port with the improved TCP Idle Scan in IPv6

4. Conducting the TCP Idle Scan in IPv6

This section deals with the characteristics of the TCP Idle Scan in IPv6. Compared to IPv4, where most modern operating systems use protection mechanisms against the scan, it is novel to conduct the scan in IPv6. Therefore, not all operating systems use the same protection mechanisms as in IPv4. To give an overview of the behavior from various operating systems, tests have been conducted with 21 different systems, and the results are shown and discussed in the first part of this section. Afterwards, the feasibility for an attacker to execute the TCP Idle Scan in IPv6 in IPv4 networks is discussed.

4.1. Behavior of various systems

A crucial factor for the TCP Idle Scan in IPv6 to be feasible is the predictable assignment of the *identification* value in the IPv6 extension header for fragmentation by the idle host. If the value is assigned in an unpredictable way, an attacker is not able to determine if the idle host sent a TCP segment with the RST-flag to the target, which makes it impossible to perform the scan as presented. Another necessity is the assignment of the value on a global basis. This means that for all IPv6 packets which use the extension header for fragmentation, the same counter should be accessed to determine its *identification* value, independent from the destination address of the packet. Such a behavior is necessary for the attacker to detect an increase in the counter in case the TCP segment with the RST-flag was sent in step 8 of the TCP Idle Scan in IPv6. If the idle host would maintain a separate counter for every destination, this increase could not be detected.

To determine which operating systems form appropriate idle hosts, 21 different operating systems and versions have been tested to establish their method of assigning the *identification* value. The results have been gathered by using the *IPv6 Toolkit* [45] and have been verified with ICMPv6 Echo Requests. Those requests contained enough data so that they needed to be fragmented and were sent from different hosts to the tested systems. In the received ICMPv6 Echo Responses, the *identification* values were analyzed with a network sniffer. The reason for sending the messages from different hosts was to determine if the values were assigned globally, or on a per host basis. Table 1 shows the results of the tests.

Among all the tested systems, six assigned the *identification* value on a random basis and are therefore not suitable as idle host. Out of the remaining 15, five assigned their values on a per host basis. Compared to the *identification* value in IPv4, where only Solaris 11 assigned values on a per host basis [29], this is a big increase. Another system which can not be used as idle host is OS X 10.6.7, which does not support atomic fragments. The details for this system will be discussed later in this section.

This leaves nine out of the 21 tested systems, which can be used as idle host for the TCP Idle Scan in IPv6, which is every tested Windows operating system. All those nine systems have a host-based firewall, which is enabled by default [9]. The firewall blocks ICMPv6 Echo Requests as well as TCP segments with a SYN- and ACK-flag without a prior segment using the ACK-flag. Therefore, the TCP Idle Scan in IPv6 is impossible by using one of the tested Windows operating systems as long as the firewall is active. A

#	**System**	**Assignment of Identification**
1	Android 4.1 (Linux 3.0.15)	Per host, incremental (1)
2	FreeBSD 7.4	Random
3	FreeBSD 9.1	Random
4	iOS 6.1.2	Random
5	Linux 2.6.32	Per host, incremental (2)
6	Linux 3.2	Per host, incremental (1)
7	Linux 3.8	Per host, incremental
8	OpenBSD 4.6	Random
9	OpenBSD 5.2	Random
10	OS X 10.6.7	Global, incremental (3)
11	OS X 10.8.3	Random
12	Solaris 11	Per host, incremental
13	Windows Server 2003 R2 Standard 64bit, SP2	Global, incremental
14	Windows Server 2008 Standard 32bit, SP1	Global, incremental
15	Windows Server 2008 R2 Standard 64bit, SP1	Global, incremental by 2
16	Windows Server 2012 Standard 64bit	Global, incremental by 2
17	Windows XP Professional 32bit, SP3	Global, incremental (4)
18	Windows Vista Business 64bit, SP1	Global, incremental
19	Windows 7 Home Premium 32bit, SP1	Global, incremental by 2
20	Windows 7 Ultimate 32bit, SP1	Global, incremental by 2
21	Windows 8 Enterprise 32 bit	Global, incremental by 2

(1) Hosts calculates wrong TCP checksum for routes with PMTU $<$ 1280
(2) PMTU $<$ 1280 results in DoS
(3) Does not accept PMTU $<$ 1280
(4) IPv6 disabled by default

Table 1: List of tested systems

deactivated firewall though makes all tested hosts running a Windows operating system suitable as idle host.

A special behavior occurred when testing Windows 8 and Windows Server 2012. A first analysis of the *identification* values sent to different hosts gives the impression that the values are assigned on a per-host-basis and start at a random initialization value. Closer investigation though revealed that the values being assigned for one system are also incremented if messages are sent to another system. This leads to the conclusion that those operating systems use a global counter, but also a random offset for each host, which is added to the counter to create the *identification* value. However, the global counter is increased each time a message is sent to a host. For the TCP Idle Scan in IPv6, this means that the systems are still suitable as idle hosts, as from the view of the attacker, the *identification* value received from the idle host increases each time the idle host sends a message to the target. It is not clear to the author what should be achieved with this behavior.

4.2. Incorrect behavior of systems

Besides the analysis of the *identification* values, incorrect behavior occurred with certain systems, which did not behave as defined in the RFCs. This can be either traced back to bugs in the networking stack of the kernel or to conscious ignoring of the RFCs.

Denial of Service This concerns Linux 2.6.32. When an ICMPv6 Packet Too Big message is received, the PMTU is adjusted accordingly without requiring any previous traffic. But if the MTU value in the ICMPv6 message is lower than the IPv6 minimum MTU of 1280 bytes, this results in a Denial of Service (DoS) for the path. Until the PMTU is increased again (See point 1 of Section 3.5) and reaches again a value smaller or equal to the IPv6 minimum MTU, no packets are sent on the path. However, this bug is fixed in Linux 3.2, which makes further investigation unnecessary.

Wrong TCP checksums While the bug regarding DoS has been fixed in Linux 3.2, another problem was encountered while testing Linux 3.2 as well as Android 4.1, running Linux 3.0.15. When these systems were used as idle host, the TCP segments with the RST flag were sent, but not accepted by the target. This resulted in the target repeating to send the segment with the SYN- and ACK-flag, and the idle host therefore repeating its answer with the RST-flag, which was again not accepted by the target, and so on. A detailed investigation of the problem revealed that the kernel calculated the wrong TCP checksum if the PMTU was set to a value smaller than the IPv6 minimum MTU. Similar to the DoS bug, this has been fixed in a later kernel version, namely Linux 3.8. Again, this made further investigation and bug reporting unnecessary.

Unsupported atomic fragments This behavior occurred while testing Mac OS X 10.6.7, which is running the kernel *xnu-1504.9.37*. While ICMPv6 Packet Too Big messages with a MTU bigger or equal to 1280 bytes were accepted and the PMTU was adjusted accordingly, values below were ignored instead of appending an empty IPv6 extension header for fragmentation to each IPv6 packet. An investigation of the kernel sources confirmed this behavior [2]. The kernel used by the newest OS X version 10.8.2 fixed this problem, which has been confirmed by investigation.

4.3. Dual Stacking

The version of the TCP Idle Scan introduced in this thesis as well as the properties of various systems described in Section 4.1 only apply to IPv6, not to IPv4. However, each of the tested systems in Table 1, except Windows XP, works out of the box in dual stack mode. This means that although being connected to an IPv4 network, each interface is automatically provided with a link-local IPv6 address which can be used for communication within the local network [18]. Therefore, despite being connected to an IPv4 network, an attacker might still be able to execute the TCP Idle Scan in IPv6 within this network by using the link-local addresses of the target and the idle host.

One requirement for the idle host in the TCP Idle Scan in IPv6 is not to create fragmented IPv6 traffic. A host which is connected to an IPv4 network will most likely not create IPv6 traffic at all, which increases the probability that it is suitable as idle host for the TCP Idle Scan in IPv6, since this requirement can be considered fulfilled.

Additionally, dual stack systems listen for ICMPv6 Router Advertisements. Those are sent periodically by routers on the network as well as on request of hosts, and contain routing information and information for hosts which IPv6 address they can assign for their interfaces [35]. In case of receiving an ICMPv6 Router Advertisement for a specific subnet, a host will assign itself an IPv6 address within this subnet. The header of the ICMPv6 Router Advertisement is shown in Figure 11.

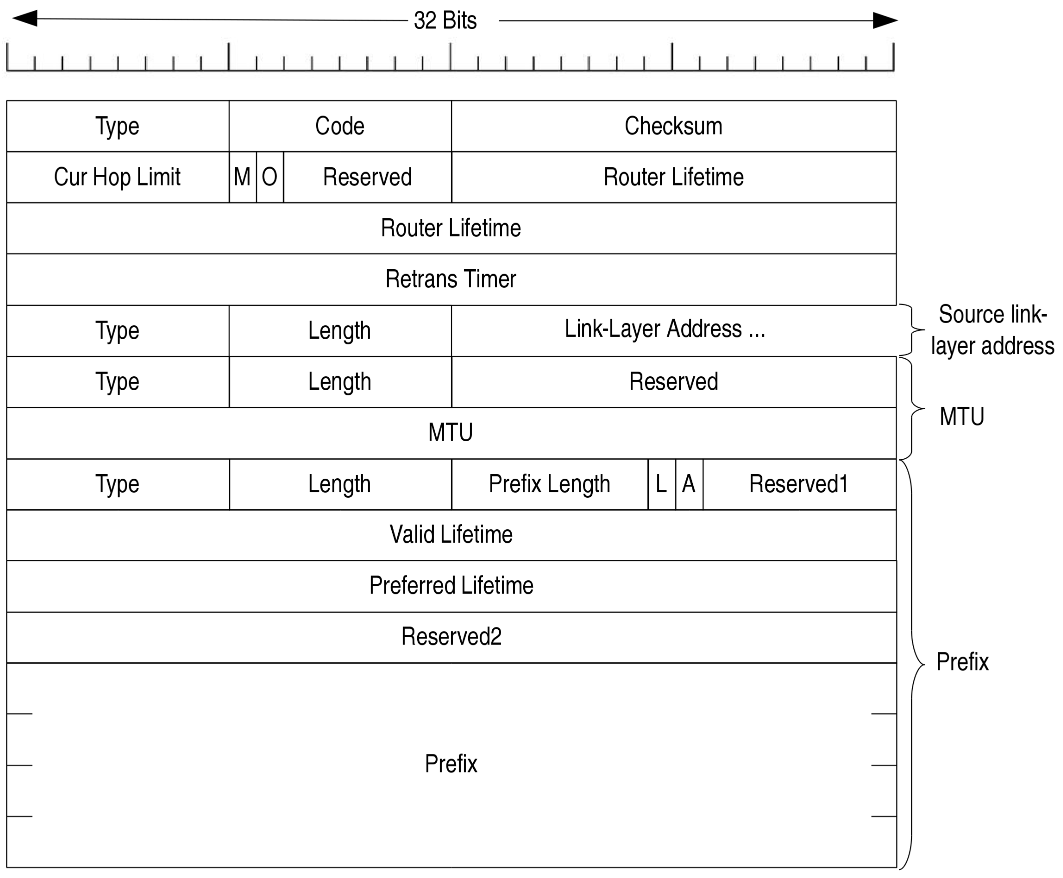


Figure 11: ICMPv6 Router Advertisement (based on RFC 4861 [35])

In the figure, the basic header as well as the three optional headers for the source link-layer address, the MTU and the prefix are shown. Within the context of this work, one interesting detail is the optional header for the prefix, which gives information for the stateless autoconfiguration [35]. After receiving an ICMPv6 Router Advertisement with this optional header, the host will by default assign itself an IPv6 address using the received prefix. This can be used by an attacker to assign a global IPv6 address to other hosts where afterwards port scans can be executed with this IPv6 address [20].

By using spoofed ICMPv6 Router Advertisements, an attacker can also claim to be an IPv6 router. A host receiving this information will add the route with the attacker as router to its IPv6 routing table. As IPv6 routes are preferred over IPv4 routes, this can be used to execute a Man in the Middle (MitM) attack [20] [50].

Another thing which attracts attention is the optional header for the MTU. This one is intended to be used to specify the MTU for routes, in case nodes behind the router do not generate ICMPv6 Packet Too Big messages [35]. The coherent approach for an attacker would be therefore to specify a MTU smaller than the IPv6 minimum MTU, so that the victim appends an extension header for fragmentation to each IPv6 packet in order to execute step 5 of the TCP Idle Scan in IPv6, similar to the approach of sending the ICMPv6 Packet Too Big message as described in Section 3.2.2. However, RFC 4861 states the following about the MTU value:

> If the MTU option is present, hosts SHOULD copy the option's value into LinkMTU so long as the value is greater than or equal to the minimum link MTU [IPv6] and does not exceed the maximum LinkMTU value specified in the link-type-specific document [35, Page 53].

The fact that only values for the MTU greater or equal to the IPv6 minimum MTU are accepted by the host makes it impossible for an attacker to force the idle host to append the IPv6 extension header for fragmentation to packets sent to the target by using ICMPv6 Router Advertisements. However, as described in this subsection, spoofing such messages can still be useful for an attacker, for example to execute a MitM attack or to assign IPv6 addresses to hosts.

5. Alternatives to the TCP Idle Scan in IPv6

One requirement for the idle host in the TCP Idle Scan in IPv4 as well as in IPv6 is the predictability of the *identification* value. To avoid this requirement, Ensafi et al. [12] describe two alternatives for the TCP Idle Scan, which do not rely on the predictable assignment of the *identification* value by the idle host. Instead, one method makes use of the RST rate limit of the idle host, while another one utilizes the limited amount of half-open TCP connections the idle host can remember.

5.1. RST Rate Limit Scan

This section deals with the RST Rate Limit Scan. At first, the concept itself, as described by Ensafi et al. [12], is introduced. Afterwards, the advantages and disadvantages of the scan compared to the TCP Idle Scan in IPv4 and IPv6 are discussed. In the end, an overview over various operating systems and their RST rate limits is provided.

5.1.1. Concept of the RST Rate Limit Scan

Similar to the TCP Idle Scan, the three participating parties are the attacker, the idle host and the target. The idle host has a limit of how many TCP segments with the RST-flag it sends per second, the so-called RST rate limit. Such a limit is for instance enforced to slow down port scans [29]. In the following example, the limit is a maximum of 100 TCP segments with the RST-flag per second. Figure 12 shows the message sequence for conducting the scan.

1. First, the attacker sends a TCP segment with the SYN-flag to the idle host, and states as destination a closed port. In order to remain undetected, the source address of the IP packet is spoofed and substituted with one that is not in use.

2. As the target port is closed, the idle host will respond with a TCP segment with the RST-flag. Due to the spoofed IP address, the reply will not reach any destination.

   These first two steps are repeated until the idle host uses all of its RST rate limit but one. In Figure 12, in which the idle host has an RST rate limit of 100, the steps would be executed 99 times, until the idle host can only send one more TCP segment with the RST-flag within the same second. It is worthwhile to mention that the TCP segment with the SYN-flag of step 1 can also be substituted by every other segment which leads to the idle host sending a TCP segment with the RST-flag, for example an unexpected TCP segment with the SYN- and ACK-flag.

3. Afterwards, the attacker creates a TCP segment with the SYN-flag, and sends it to the target. Similar to the TCP Idle Scan, the source IP address is spoofed, and substituted with the address of the idle host.

4. Because the source IP address in the last step was spoofed, the target will not answer to the attacker, but to the idle host. If the port on the target is closed, it will send a TCP segment with the RST-flag to the idle host (4a). But if the port is

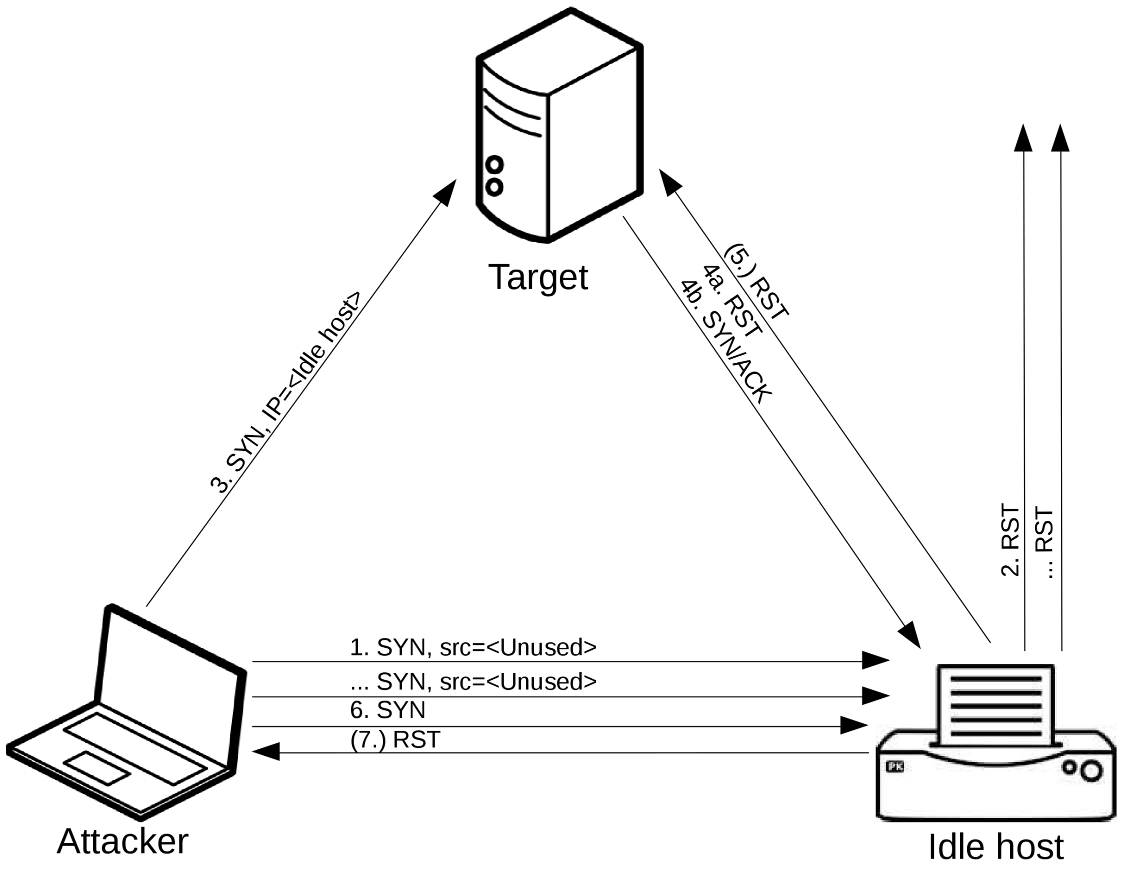


Figure 12: RST Rate Limit Scan

open, the target will send a segment with the SYN- and ACK-flag (4b) to continue the TCP three way handshake.

5. In case of receiving a TCP segment with the RST-flag, no further actions are carried out by the idle host. But in case of receiving a segment with the SYN- and ACK-flag, the idle host will answer with a TCP segment having set the RST-flag, using its last available TCP segment with the RST-flag within its RST rate limit.

6. To receive the result of the scan, the attacker sends a TCP segment with the SYN-flag to the idle host to a closed port, and uses his real address in order to receive the answer.

7. As the port on the idle host is closed, it will try to answer with a TCP segment with the RST-flag. But if the port on the target in step 3 was open, the idle host already reached its RST rate limit by sending the last TCP segment with the RST-flag in step 5. Therefore, if the limit is reached, the idle host will not be able to answer the TCP segment with the SYN-flag sent by the attacker in step 6. This

lets the attacker conclude that the scanned port on the target is open. In case of a closed port on the target, the idle host did not have to execute step 5, and is still able to send one more TCP segment with the RST-flag without reaching its RST rate limit. Therefore, it will reply to the attacker by sending a TCP segment with the RST-flag. The attacker can conclude that the scanned port on the target was closed.

5.1.2. Characteristics of the RST Rate Limit Scan

Compared to the traditional TCP Idle Scan in IPv4 and IPv6, the RST Rate Limit Scan has advantages as well as disadvantages. Most important, it does not rely on the predictable assignment of the *identification* value in the IP header by the idle host. Instead, it relies on the idle host to use a RST rate limit.

A big disadvantage of the scan is that the attacker needs to be aware of the idle hosts RST rate limit in order to force the borderline cases necessary for the scan. The RST rate limit can often be found in the documentation of the operating system. An alternative to identify the RST rate limit would be a test, in which a big amount of TCP segments, which will cause a segment with the RST-flag as response, will be sent to the idle host. Afterwards, the amount of packets with the RST-flag received from the idle host, which complies to its RST rate limit, is determined.

Another issue with the RST Rate Limit Scan is that it requires a lot more messages than the TCP Idle Scan, at which the exact amount of messages depends on the idle host. This is because the attacker has to fill the idle hosts RST rate limit but one when he wants to scan a port. Also, after having scanned one port, the maximum amount of TCP segments with the RST-flag which the idle host can send is reached. This requires the attacker to wait until the RST rate limit is reset before scanning another port, which slows down the scan dramatically.

A positive aspect from the RST Rate Limit Scan is that it works only with the TCP layer, independent from the IP layer. While the TCP Idle Scan has to be adapted by changing from IPv4 to IPv6, the RST Rate Limit Scan is unaffected by this exchange. Finally, it also needs to be considered that the stricter a host is configured, and the less TCP segments with the RST-flag it allows per second, the easier it is for an attacker to reach this limit and utilize the host as idle host.

The described advantages and disadvantages lead to the following requirements for the idle host regarding the RST Rate Limit Scan:

1. The idle host should not create TCP segments containing the RST-flag without influence of the attacker. This would disturb the scanning-process.

2. The idle host needs to limit the amount of TCP segments containing the RST-flag it sends within a certain period.

For regular traffic, it should not be necessary for the idle host to send TCP segments with the RST-flag, as those are only used to indicate connection problems [48]. Therefore, the first requirement can be considered fulfilled in regular networks. On the contrary, the

#	System	RST rate limit per second
1	FreeBSD 9.1	200
2	iOS 6.1.2	50
3	Linux 3.8	-
4	OS X 10.8.3	250
5	OpenBSD 5.2	100 (1)
6	Solaris 11	40
7	Windows XP Professional 32bit, SP3	-
8	Windows 7 Ultimate 32bit, SP1	-
9	Windows Server 2008 R2 Standard 64bit, SP1	-
10	Windows 8 Enterprise 32 bit	-

(1) Considers as well outgoing as incoming TCP segments with RST-flag

Table 2: RST rate limits of various systems

RST rate limit in the second requirement is not enforced by all operating systems. To give an overview which operating systems use RST rate limits, tests have been conducted with various operating systems, as only FreeBSD has been tested by Ensafi et al. [12]. Table 2 shows which of the tested operating systems limit the amount of TCP segments with RST-flags, and for those which do, the limit is indicated.

Out of the ten tested operating systems, five limit the amount of TCP segments with the RST-flag. OpenBSD 5.2 limits by default the amount of outgoing TCP segments containing the RST-flag to 100, according to the manual pages [36]. Different from the description in the manual pages, the diagnosed behavior was that the limit included outgoing TCP segments as well as incoming segments. While executing the RST Rate Limit Scan, the idle host will either receive a TCP segment with the RST-flag in step 4a, or send the segment on its own by executing step 5. Either way, the message is considered by OpenBSD for the RST rate limit. Step 7, sending the last TCP segment with the RST-flag, will therefore never be executed by the idle host, as its RST rate limit is reached, regardless of the port on the target being open or closed. This makes the RST Rate Limit Scan impossible by using OpenBSD 5.2 as idle host.

Reaching these results, four out of the ten tested operating systems remain suitable as idle host with the RST Rate Limit Scan, namely FreeBSD, iOS, OS X and Solaris.

5.2. SYN Cache Scan

The second alternative to the traditional TCP Idle Scan in IPv4 and IPv6 is the SYN Cache Scan, which will be described in this section. At the beginning, SYN-Flooding attacks, which are a necessary background to understand the concept of the scan, are explained. Then, the concept itself, as described by Ensafi et al. [12], is introduced. Afterwards, the advantages and disadvantages of the scan compared to the TCP Idle Scan in IPv4 and IPv6 are discussed. In the end, an overview of various operating systems and their limits for half-open connections is provided.

5.2.1. SYN-Flooding attacks and defenses

In order to understand why the SYN Cache Scan works as described in the following section, it is necessary to discuss SYN-Flooding attacks. When a TCP three way handshake is executed correctly, as shown in Figure 2a on page 10, the client first sends a TCP segment with the SYN-flag to the server [11]. The server answers with a TCP segment containing the SYN- and ACK-flag, and stores the half-open connection to remember related information, such as the sequence number. This number is sent back to the client to uniquely identify the message. To fully establish the connection, the client answers with a TCP segment with the ACK-flag. This TCP segment also contains the sequence number received from the server, incremented by one. With this number, the server can determine to which previously sent TCP segment the answer belongs to. After receiving the TCP segment with the ACK-flag and the correct sequence number, the connection is fully established. Now, data can be sent, and the server removes the connection from the list of half-open connection, the so-called backlog queue.

When executing a SYN-Flooding attack, an attacker repeatedly sends TCP segments with the SYN-flag to a server. While the server will respond with a TCP segment containing the SYN- and ACK-flag, the attacker will drop the message from the server and not send the necessary TCP segment with the ACK-flag to complete the TCP three way handshake [26]. This causes the server to keep the half-open connection saved in its backlog queue until the server decides that it waited long enough for an answer and removes the entry from the queue. If the attacker requests enough connections within a certain period, the backlog queue of the server will reach its limit. In this state, the server can not remember additional half-open connections, and new connection requests are rejected. This leads to a denial of service, as legitimate clients are not able to establish a connection with the server.

To prevent this attack, various methods have been developed. One of them are SYN-Cookies [4]. When creating a sequence number for the TCP segment with the SYN- and ACK-flag, the server makes specific choices and calculates the sequence number based on a time counter, the Maximum Segment Size (MSS) [40] chosen by the server and a secret function. This sequence number is the so-called SYN-Cookie. After the TCP segment is sent, no information is stored in the backlog queue. In case the client responds with the sequence number incremented by one, the server can calculate if this number complies to a SYN-Cookie that was created previously without requiring any additional information besides the received sequence number. If the received sequence number complies to a valid SYN-Cookie, the connection is fully established. SYN-Cookies are for example used by Linux. By default, the operating system is using a traditional backlog queue. If the system is under attack and the backlog queue is full, it falls back to sending SYN-Cookies until place in the backlog queue is freed, for example due to a timeout of a connection or because of receiving a TCP segment with the RST-flag [26].

Another method for protecting against SYN-Flooding is a SYN cache [26]. Different from SYN-Cookies, information about the half-open connection is still stored on the server. But compared to the traditional backlog queue, the amount of memory necessary to remember the connection is minimized. This is done by saving a hash value containing the source and destination address of the connection as well as the source and destination port and a randomly chosen secret instead of the original values. Due to this memory saving approach, the server is able to remember more connections using the same amount of memory as with a traditional backlog queue. Those hash values are stored in different hash-tables. In case a hash-table is full, the oldest entry is removed.

Both SYN-Cookies and SYN Cache have advantages and disadvantages, which are discussed by Lemon [26]. Besides those two methods, there are also other defense mechanisms, such as filtering and increasing the size of the backlog queue, which are discussed in RFC 4987 [11].

5.2.2. Concept of the SYN Cache Scan

The second alternative to the TCP Idle Scan is the SYN Cache Scan. Similar to the other port scanning methods introduced so far, the three participating parties are the attacker, the idle host and the target. In the example, the idle host has a limited backlog queue of the size 100. Afterwards, it falls back to sending SYN-Cookies until place in the backlog queue is freed again. Additionally, the target shows different behavior depending on if an unexpected TCP segment with the SYN- and ACK-flag is received on an open or on a closed port. If received on an open port, the target will answer with a TCP segment containing the RST-flag, as the message was unexpected. But in case of receiving the segment on a closed port, it will be dropped, and no response will be given. Figure 13 shows the sequence of messages for conducting the scan.

1. First, the attacker sends a TCP segment with the SYN-flag to the idle host, and states as destination an open port. In order to remain undetected, the source address of the message is spoofed and substituted with one that is not in use.

2. As the port on the idle host is open, it will respond with a TCP segment containing the SYN- and ACK-flag. Because the source address is not in use, the message will reach no destination, and therefore no answer will be received. As long as no answer is received or the connection times out, the entry will remain in the backlog queue of the idle host.

   These first two steps are repeated until the idle host fills all but one entries in its backlog queue. In the example, in which the size of the idle host's backlog queue is 100, the steps would be executed 99 times, until the idle host can only store one more half-open connection.

3. Now, the attacker sends again a TCP segment with the SYN-flag to the idle host, targeted to an open port. As source port, he specifies the port he wants to scan on the target. The source address is spoofed, and substituted with the target's address.

4. As the port on the idle host is open, it will respond with a TCP segment containing the SYN- and ACK-flag in order to continue the TCP three way handshake, and fill its backlog queue to the maximum. This reply will be sent to the target, because

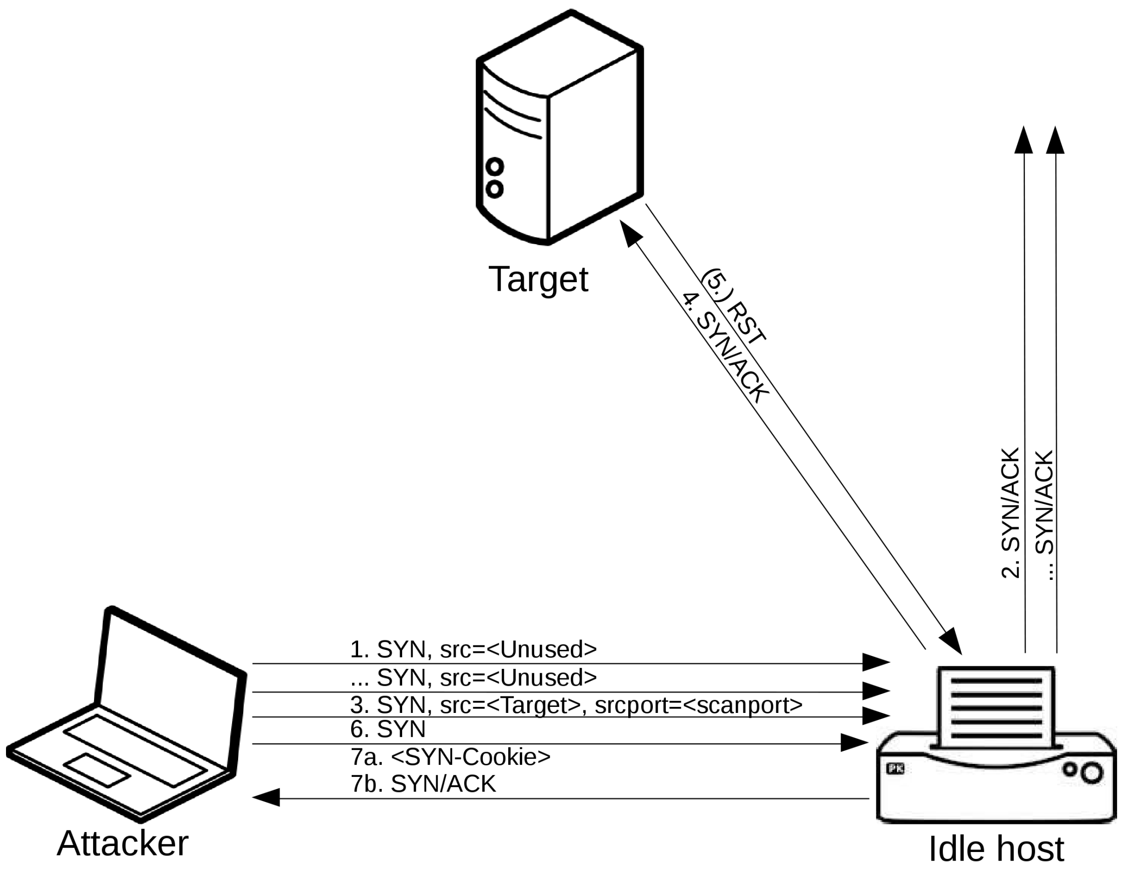


Figure 13: SYN Cache Scan

the source address in the previous step was spoofed. Also, the reply will be sent to the source port specified in the previous message, which is the one the attacker wants to scan.

5. If the port on the target is open, it will answer by sending a TCP segment with the RST-flag. Receiving this segment causes the idle host to close the half-open connection, and remove the entry from its backlog queue. Therefore, the idle host can again accept one more half-open connection before having to fall back to sending SYN-Cookies. But if the port is closed, the message will be dropped by the target and the half-open connection on the idle host will remain in the full backlog queue.

6. To get the results of the scan, the attacker sends a TCP segment with the SYN-flag to the idle host on an open port, using his real source address.

7. Because the message is received on an open port, the idle host will try to reply with a TCP segment with the SYN- and ACK-flag. If the port on the target was open and a TCP segment with the RST-flag was received in step 5, this is possible, as

there is place for one more half-open connection in the backlog queue. In this case, a normal TCP segment with the SYN- and ACK-flag will be sent. But if no TCP segment with the RST-flag was received in step 5 because the port was closed, the idle host is not able to create another half-open connection due to its full backlog queue. Instead, it will fall back to sending a SYN-Cookie. By receiving this cookie, the attacker knows that the limit of the backlog queue on the idle host is reached, which leads to the conclusion that the port on the target is closed.

5.2.3. Characteristics of the SYN Cache Scan

Like the other scans introduced in this work, the SYN Cache Scan has advantages and disadvantages. Compared to the TCP Idle Scan in IPv4 and IPv6, it does not rely on the predictable assignment of the *identification* value in the IP header by the idle host. Instead, it relies on the idle host to use a defense mechanism against SYN-Flooding attacks, such as SYN-Cookies. An overview of how different operating systems behave in case of being exposed to SYN-Flooding attacks can be found in Section 5.2.4.

An advantage compared to the RST Rate Limit Scan is that it is slightly easier to scan multiple ports. Entries in the backlog queue will remain for a certain amount of time. While the backlog queue is nearly filled, the attacker might be able to scan multiple ports without having to refill the backlog queue for each port which is to be scanned. The details for scanning multiple ports depend on the target. In case of FreeBSD's *blackhole* [41], which drops messages received on closed ports, an attacker will be able to scan multiple open ports. If he scans a closed port, the target will not answer to the idle host with a TCP-segment with the RST-flag, and the entry for the connection will remain in the idle host's backlog queue. In this situation, the attacker has to wait until place in the backlog queue is freed to continue scanning.

Alternatively, the attacker might not spoof the source addresses of all TCP segments with the SYN-flag used in step 1 to fill the idle hosts backlog queue, but send some segments with his real source address. Using this method, the answer given from the idle host is received and provides the sequence number necessary to free an entry in the backlog queue. In case an open port was scanned and the backlog queue is full, the received sequence number can be used in a TCP segment with the RST-flag sent to the idle host to free one entry from the backlog queue to enable further scanning.

Like the RST Rate Limit Scan, the SYN Cache Scan is independent from the IP layer. Defense mechanisms like SYN-Cookies fully work on the TCP layer, which makes a change of the IP protocol easily possible.

A disadvantage of this port scanning method is that the attacker is required to know details about the idle host, such as the size of its backlog queue and the defense mechanism against SYN-Flooding attacks. As they differ between operating systems, the attacker will also have to structure the attack differently depending on the operating system used by the idle host, which complicates an automated execution. Additionally, and overflowing backlog queue might create warning messages on the idle host, which will suspect a SYN-Flooding attack. Such messages are likely to raise someones attention, which drastically decreases the stealthiness of the SYN Cache Scan.

Compared to the other port scans presented in this thesis, the SYN Cache Scan also requires the idle host to have at least one open port. Otherwise it would not be possible for the attacker to store half-open connections in the backlog queue of the idle host. Additionally, it is advisable that no other clients try to establish TCP connections with the idle host. An unknown amount of entries in the backlog queue which are not from the attacker and will be removed at an unknown time will make it harder for the attacker to fill the backlog queue with exactly all but one entries. This leads to the following requirements for the idle host in the SYN Cache Scan:

1. The idle host needs to have at least one open TCP port.
2. The idle host should not receive TCP connection requests from other hosts. This would disturb the scanning-process.
3. The idle host needs to have a protection mechanism against SYN-Flooding, to which it falls back after its backlog queue is full.

Additionally, the SYN Cache Scan also has one requirement for the target. The host needs to behave differently depending on if a TCP segment with the SYN- and ACK-flag is received on an open, or on a closed port. Such a behavior could for example be caused by FreeBSD's *blackhole* [41], which drops messages received on closed ports, or by a firewall.

Considering all those requirements for the SYN Cache Scan, it does not seem to be a very tempting approach for an attacker. But compared to all other port scans introduced in this paper, the SYN Cache Scan has one very interesting characteristic: A closer look at Figure 13 reveals that not a single packet is sent from the attacker to the target, not even one with a spoofed source address. This means that the attacker is not required to be able to send packets to the target, but only to the idle host. This can enable the attacker to scan internal networks to which he should not have access.

An example for this could be the DeMilitarized Zone (DMZ) [48] of a company network. The idle host could be a web server in the DMZ, which is reachable from the outside. Also, the DMZ contains a database server, which is not accessible from external networks. This database server represents the target. Using the SYN Cache Scan, an attacker is able to execute a port scan on the database server from outside the DMZ, even though being separated by a firewall. Figure 14 shows the SYN Cache Scan in the described scenario.

In case the target in this scenario does not fulfill the requirement of different behavior depending on receiving the TCP segment with the SYN- and ACK-flag on a closed or on an open port, the SYN Cache Scan can instead also be used to perform host discovery in the DMZ. While alive hosts will respond with a TCP segment containing the RST-flag after receiving the TCP segment with the SYN- and ACK flag, no response will be created by non-existing hosts. Due to this behavior, an attacker can spoof different source addresses in step 3 of the SYN Cache Scan. If those addresses exist, step 5, sending the TCP segment with the RST-flag will be executed, and the entry in the backlog queue of the idle host will be freed. Therefore, in step 7, the attacker will receive a normal TCP

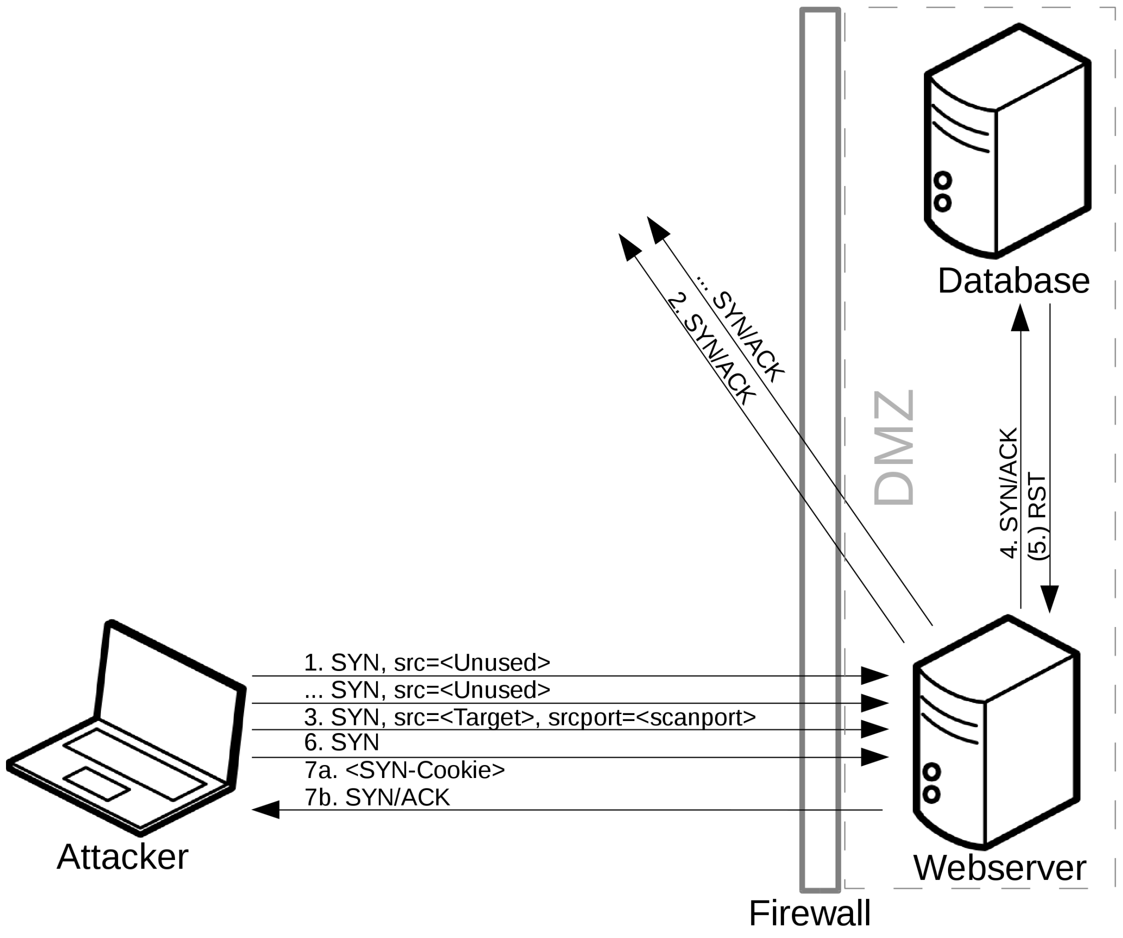


Figure 14: SYN Cache Scan into DMZ

segment with the SYN- and ACK-flag, and no SYN-Cookie. In case the spoofed source address does not exist, step 5 will not be executed, and the backlog queue of the idle host will remain full. Because of this, the attacker will receive a SYN-Cookie in step 7. Using this method, an attacker is able to detect which addresses are used within the DMZ.

5.2.4. Behavior of various systems

To determine what operating systems make use of which defense mechanisms against SYN-Flooding, and which of those operating systems can be used as idle host for the SYN Cache Scan, tests have been conducted with various operating systems. The results are shown in Table 3.

For each tested operating system, Table 3 also indicates the amount of RAM the system was using, as the amount of RAM has influence on the size of the backlog queue for some operating systems [12] [31]. In the next column, the countermeasure used against SYN-Flooding is indicated, followed by the amount of half-open connections the server stores before it falls back to using the countermeasure. The last column states how long an half-open connection stays in the backlog queue before it is removed when no matching TCP segment is received.

Out of the ten tested operating systems, two use SYN-Cookies as defense mechanisms against SYN-Flooding attacks, namely FreeBSD and Linux. FreeBSD uses SYN-Cookies by default, as soon as the first connection request is received. This makes it unfeasible for an attacker to use FreeBSD as idle host for the SYN Cache Scan, as the behavior does not change after a certain amount of half-open connections. The same applies to OpenBSD, which uses a different defense mechanism, a SYN Cache, but also uses the mechanism already for the first half-open connection. Additionally for OpenBSD, it would not be possible for the attacker to determine if the system makes use of the SYN-Cache or not, as the data being sent on the network does not change compared to using a traditional backlog queue.

Linux, which falls back to using SYN-Cookies after 256 half-open connections, represents a suitable idle host for the SYN Cache Scan. What is left for the attacker is to determine if the idle host sent a SYN-Cookie or a normal sequence number in its TCP segment with the SYN- and ACK-flag. For this, Ensafi et al. [12] suggest the detection of SYN-Cookies by statistical analysis of the received sequence numbers. An approach which turned out to be easier for the author was to analyze how often the TCP segment with the SYN- and ACK-flag is resent by the idle host. Most tested operating systems resend TCP segments between three and eight times if no answer is received, where the exact amount depends on the operating system. If SYN-Cookies are used, no information regarding the connection is stored on the system. Therefore, the operating system is only able to send the TCP segment with the SYN- and ACK-flag once when SYN-Cookies are used, instead of replaying multiple times. This way of determining if the idle host sent a SYN-Cookie turned out to be the easier approach compared to statistical analysis.

Solaris was simply dropping requests if no place was left in the backlog queue. Because of this behavior, an idle host running Solaris can be used similarly to one sending SYN-Cookies: If the attacker does not receive a TCP segment from the idle host in step 7 of Figure 13, he can conclude that the backlog queue of the host is full. Being aware of this behavior, SYN Cache Scans could be executed successfully using an idle host running Solaris as operating system.

Compared to this, iOS and OS X did not drop new requests in case of a full backlog queue, but dropped old requests. In general, this behavior seems to be more advisable, as new connection requests still have a chance of establishing the TCP three way handshake instead of getting instantly dropped. However, both systems can be used as idle hosts for the SYN Cache Scan: When sending the first TCP segment with the SYN-flag to the idle host to fill its backlog queue in step 1 of Figure 13, the attacker does not spoof the source address in order to receive the reply. The idle host will respond with a TCP segment with the SYN-and ACK-flag, and will resend this segment multiple times if no answer is received. Meanwhile, the attacker continues executing the SYN Cache Scan as described, all other TCP segments with the SYN-flag of step 1 will have a spoofed source address in order for the attacker to remain undetected. If now step 6 in Figure 13 causes the backlog queue on the idle host to overflow, the oldest connection in the queue will be dropped, which is the one being made by the attacker from his own address in step 1. By analyzing how often the TCP segment with the SYN- and ACK-flag in step 2 has been sent back to the attacker, an attacker can determine if the backlog queue was overflown or not, and therefore conclude if the port on the target is open or closed.

Also four systems from the Windows family have been tested, which showed different behavior in case of SYN-Flooding attacks. While Windows XP rejected new connection requests by sending a TCP segment with the RST-flag in case the backlog queue was full, no limit of half-open connections could be determined for Windows 7 and Windows Server 2008 R2. Windows 8 normally replays the TCP segment with the SYN- and ACK-flag multiple times, and afterwards sends a TCP segment with the RST-flag if no answer is received. In case the backlog queue is full, the final TCP segment with RST-flag is not sent. It is assumed that the reason for this behavior is that those connections are dropped. However, documentation on this behavior leaves the details unclear [31]. Apart from showing different behavior with a full backlog queue, the tested operating systems from the Windows family all showed the same behavior when resuming normal behavior after an entry of the backlog queue being removed. Regardless of whether a TCP segment with RST-flag was received in step 5 of Figure 13 or not, the answer to the attacker in step 7 remained the same. The experienced behavior leads to the conclusion that the amount of half-open connections is limited per period, not considering if place in the backlog queue is freed within this period or not. This makes the tested Windows operating systems unsuitable as idle hosts for the SYN Cache Scan.

To sum up, four out of the ten tested operating systems are suitable to be used as idle host for the SYN Cache Scan, namely iOS, Linux, OS X and Solaris. However, those four systems show three different kinds of behavior in case of a full backlog queue, which need to be considered to successfully execute the attack.

#	System	RAM	Countermeasure	Backlog queue	Period per entry
1	FreeBSD 9.1	256 MB	SYN-Cookies	0	35s
2	iOS 6.1.2	2048 MB	Overwrite old entries	128	86s
3	Linux 3.8	1024 MB	SYN-Cookies	256	180s
4	OS X 10.8.3	2048 MB	Overwrite old entries	128	75s
5	OpenBSD 5.2	256 MB	SYN-Cache	0	93s
6	Solaris 11	1536 MB	Dropping Requests	1024	190s
7	Windows XP Professional 32bit, SP3	512 MB	Dropping Requests	50	21s (1)
8	Windows 7 Ultimate 32bit, SP1	1024 MB	-	-	21s (1)
9	Windows Server 2008 R2 Standard 64bit, SP1	1024 MB	-	-	21s (1)
10	Windows 8 Enterprise 32 bit	1024 MB	No RST	4094	21s (1)

(1) Limit of half-open connections per period

Table 3: SYN protection mechanisms of various systems

6. Implementations

In the previous sections, three alternatives for the TCP Idle Scan in IPv4 have been presented. To prove besides the theoretical also the practical feasibility of those scanning methods, all three port scans have been implemented using the python program *scapy* [5]. Scapy is a powerful packet manipulation tool, which allows the easy creation of network packets. Additionally, the TCP Idle Scan in IPv6 was also implemented as extension to the security scanner *Nmap* [28].

6.1. TCP Idle Scan in IPv6 using scapy

Listing 1 shows the python [5] source code which implements the TCP Idle Scan in IPv6 using *scapy*. The source code follows the TCP Idle Scan in IPv6 as shown by Figure 8 on page 21.

Listing 1: The TCP Idle Scan in IPv6 using scapy

```python
#!/usr/bin/python
from scapy.all import *

#the addresses of the three participants
idlehost="<IPv6-address>"
attacker="<IPv6-address>"
target="<IPv6-address>"

# MTU which will be announced in the ICMPv6 PTB message
newmtu=1278

# Checksum which the ICMPv6 PTB message will have
checksum=0x6fb0

# the port which is to scan
port=22

# configure scapy's routes and interfaces
conf.iface6="eth0"
conf.route6.ifadd("eth0","::/64")

# create a fragmented ping from the target to the idle host
ping_target=fragment6(IPv6(dst=idlehost,src=target)\
/IPv6ExtHdrFragment()\
/ICMPv6EchoRequest(id=123,data="A"*1800),1400)
# send the ping
send(ping_target[0])
send(ping_target[1])

# we do not get the response, so we have to make our own one
response=IPv6(plen=1248,nh=0x3a,hlim=64,src=idlehost,dst=target)\
/ICMPv6EchoReply(id=123,cksum=checksum,data="A"*1800)
```

```
33 # take the IPv6 layer of the response
34 ipv6response=response[IPv6]
35 # reduce the amount of data being sent in the reply
36 # (a ICMPv6 PTB message will only have a maximum of 1280 bytes)
37 ipv6response[IPv6][ICMPv6EchoReply].data="A"*(newmtu-69)
38 
39 # wait a second, so that the target has
40 # enough time to answer the idle host
41 time.sleep(1)
42 
43 # tell the idle host that his reply was too big, the MTU is smaller
44 mtu_idlehost_to_target=IPv6(dst=idlehost, src=target)\
45 /ICMPv6PacketTooBig(mtu=newmtu)/ipv6response
46 # send the ICMPv6 PTB message
47 send(mtu_idlehost_to_target)
48 
49 # create a huge, fragmented ping to the idle host
50 # to get its fragmentation identification
51 fragments=fragment6(IPv6(dst=idlehost, src=attacker, nh=0x2c)\
52 /IPv6ExtHdrFragment()/ICMPv6EchoRequest(data="A"*1800),100)
53 
54 # send the huge ping
55 send(fragments[0])
56 send(fragments[1])
57 
58 # send a spoofed syn to the target in the name of the idle host
59 syn=IPv6(dst=target, src=idlehost)\
60 /TCP(dport=port, sport=RandNum(1,8000), flags="S")
61 send(syn)
62 
63 # give the idle host some time to send an rst
64 time.sleep(1)
65 
66 # send the huge ping again
67 send(f0)
68 send(f1)
```

At first, all necessary parameters are configured. In the lines 23-28, an ICMPv6 Echo Request with enough data that it needs to be fragmented is sent to the idle host, with the spoofed source address of the target (Step 1 in Figure 8). Line 31 and 32 recreate the ICMPv6 Echo Response the idle host will send to the target (Step 2). After the data in the response is reduced, the IPv6 layer of this message is used for the ICMPv6 Packet Too Big message. Reduction of the data from the ICMPv6 Echo Response is necessary as the message is not allowed to be bigger than the IPv6 minimum MTU of 1280 bytes. Now, the ICMPv6 Packet Too Big message is created and sent in the lines 44-46 (Step 3).

Lines 51-55 create and send another ICMPv6 Echo Request with enough data that it needs to be fragmented to the idle host, but this time the source address is not spoofed

(Step 4). In the lines 59-61, a TCP segment with the SYN-flag and a spoofed source address of the idle host is created, and sent to the target (Step 6). Afterwards, the script waits for one second in order to give the target enough time to answer to the idle host, and if necessary, the idle host to reply to the target (Step 7 and 8). Finally, lines 67 and 68 re-send the fragmented ICMPv6 Echo Request (Step 9).

By observing the described steps with a network sniffer such as *Wireshark* [51], the *identification* value used in the IPv6 extension header for fragmentation in step 5 and 10 can be analyzed. Depending on the difference between those two values, it can be concluded if the scanned port on the target is open or closed.

6.2. RST Rate Limit Scan using scapy

The next implementation which was done with scapy is the RST Rate Limit Scan. The source code acts according to Figure 12 on page 34.

Listing 2: The RST Rate Limit Scan using scapy

```python
#!/usr/bin/python
import sys
from scapy.all import *

idle_host="<address>"
attacker="<address>"
target="<address>"
target_mac="<mac-address>"
idle_host_mac="<mac-address>"

# the port which is to scan
port=22

# port on the idle host which create a RST when receiving a SYN/ACK
rstport=1

# RST rate limit of the idle host
rstlimit=50

# configure scapy's routes and interfaces
conf.iface6="eth0"
conf.route6.ifadd("eth0","::/64")

# create a tcp segment to which the idle host will respond
# with a RST
getrst=Ether(src=RandMAC('*:*:*:*:*:*'),dst=idle_host_mac)\
/IPv6(src=RandIP6(),dst=idle_host_mac)\
/TCP(flags="S",dport=rstport,sport=RandNum(1,8000))

# send the segments fast by using sendpfast()
sendpfast(getrst,iface="eth0",loop=rstlimit-1)

```

```
33 # send a spoofed syn to the target in the name of the idle_host
34 syn=IPv6(dst=target,src=idle_host)\
35 /TCP(dport=port,sport=RandNum(1,8000),flags="S")
36 send(syn)
37
38
39 # send again a message where a rst should come back
40 # - this time with our own address so that we receive the answer
41 getrst=IPv6(src=attacker,dst=idle_host)\
42 /TCP(flags="S",sport=RandNum(1,8000),dport=rsport)
43 sr1(getrst)
```

Similar to the TCP Idle Scan in IPv6, the first lines configure the necessary parameters. The implementation uses IPv6 addresses, but changing to IPv4 would only require a minimum amount of work. In the lines 26-31, a TCP segment with the SYN-flag and a random, spoofed source address is created and addressed to a closed port on the idle host. Afterwards, this segment is sent to the idle host one time less than its RST rate limit (Step 1 of Figure 12). Line 34-36 create a TCP segment with the SYN-flag and the spoofed source address of the idle host and send it to the target (Step 3). As the RST rate limit of the idle host is only valid within one second, the script does not pause at this point. Finally, another TCP segment with the SYN-flag is sent to the idle host on a closed port and the answer is recorded (Step 6 and 7). In case the idle host does not respond with a TCP segment with the RST-flag, the port on the target was open. If the segment is received, the port was closed.

6.3. SYN Cache Scan using scapy

Also the SYN Cache Scan has been implemented with scapy. Listing 3 shows the source code, which acts according to Figure 13 on page 39.

Listing 3: The SYN Cache Scan using scapy

```
#!/usr/bin/python
import time
import random
import sys
from scapy.all import *

target="<address>"
attacker="<address>"
zombie="<address>"

# the port which is to scan
port=22

# an open port on the idle host
openport=22

# the size of the idle host's backlog queue
```

```
18 syncache=100
19
20 # configure scapy's routes and interfaces
21 conf.iface6="eth0"
22 conf.route6.ifadd("eth0","::/64")
23
24 # fill the backlog queue -1 by sending requests
25 for i in xrange(0,syncache-1):
26   # create SYNs with random source addresses to fill backlog queue
27   fillsyn=IPv6(dst=zombie,src=RandIP6("2001::*:*")ip6src)\
28   /TCP(sport=RandNum(1,8000),dport=openport,flags="S")
29   send(fillsyn,inter=0.0,verbose=0)
30
31 # send a spoofed syn to the zombie in the name of the target
32 syn=IPv6(dst=zombie,src=target)/TCP(dport=openport,sport=port)
33 send(syn)
34
35 # wait a bit, so that the target can answer to the idle host
36 time.sleep(1)
37
38 # send again a message where a syn/ack should come back
39 # - this time with our own address so that we receive the answer
40 getsyn=IPv6(dst=zombie)\
41 /TCP(sport=RandNum(1,8000),dport=openport,flags="S")
42 ans=sr1(getsyn)
43
44 # show the received answer to analyze if it is a SYN-Cookie
45 ans[TCP].show()
```

As in the previous scripts, the first lines of code are used for initialization. Line 25-29 are used to fill the backlog queue of the idle host with all but one entries by using TCP segments with the SYN-flag and a spoofed source address (Step 1 in Figure 13). Afterwards, another TCP segment with the SYN-flag, the spoofed source address of the target and the source port which should be scanned is created in line 32 and 33, and sent to the idle host (Step 3). Then the script waits for one second in order to give the idle host enough time to answer to the target, and if necessary, the target to reply (Steps 4 and 5). Finally, the last TCP segment with the SYN-flag is created and sent to the idle host in lines 40-42, and the answer is displayed (Step 6 and 7). By analyzing the received sequence number, the attacker can determine if the port is open or closed. Alternatively, a network sniffer can be used to analyze how often step 7, sending the TCP segment with the SYN- and ACK flag, is executed by the idle host.

6.4. TCP Idle Scan in IPv6 with Nmap

After creating a proof of concept with scapy, the TCP Idle Scan in IPv6 was implemented in Nmap [28] to provide a more elaborated scanning environment. Nmap was already capable of executing the TCP Idle Scan in IPv4. This functionality could be used as a basis for implementing the TCP Idle Scan in IPv6, as both scans rely on the same basic concept. At first, this section discusses how the TCP Idle Scan in IPv4 is implemented in Nmap to understand how it is possible to achieve a much more efficient scan in comparison to the book-version. Afterwards, it is shown which adjustments had to be made to transfer the scan to IPv6, and finally the performance of the implementation is discussed.

6.4.1. TCP Idle Scan in IPv4 with Nmap

While Figure 3 on Page 12 shows the basic concept of the TCP Idle Scan in IPv4, the Nmap implementation has been optimized in order to decrease the time needed to scan multiple ports as well as to increase reliability [29].

First, the sequence in which the IPIDs are assigned is tested. This is done by sending six TCP segments with the SYN- and ACK-flag to the idle host, to which the host will respond with six TCP segments with the RST-flag. From these responses, it is determined by how much the IPID is increased for each packet sent. In case the idle host does not respond to the TCP segments, it is assumed that the host is behind a firewall, which makes it impossible to use it as idle host in the TCP Idle Scan, and the scan is aborted.

When the IPID sequence has shown to be predictable, the next step is to detect if the IPIDs are assigned on a global, or on a per-host basis. For this, four TCP segments with the SYN- and ACK-flag are sent to the idle host with the spoofed source address of the target. The responses to these messages will be sent to the target, and therefore will not be received. Instead, after giving the idle host enough time to answer to the target, another TCP segment with the SYN- and ACK-flag and the real source address is sent to the idle host, and the IPID in the response is analyzed. If the IPID increased five times compared to the last response received from the idle host, it is assumed that the last four IPIDs have been used to reply to the spoofed TCP segments. Therefore, it can be concluded that the IPIDs are assigned on a global basis.

In case the IPID only incremented once, independent IPIDs have been used for the answer to the target, which allows the conclusion that the IPIDs are assigned on a per-host basis. An alternative explanation would be that the attacker is not able to spoof IP-addresses, for example due to protection mechanisms of the Internet Service Provider (ISP), or that the spoofed segments are recognized and dropped by the target. Anyhow, the host is not suitable as idle host, and the scan is aborted.

After assuring the reliability of the idle host, the actual scan is started. This is done by sending up to 100 TCP segments with the SYN-flag to the target, addressed at ports which should be scanned, and spoofing the source address of the target. When receiving these segments, the target will answer with a certain amount of TCP segments with the RST-flag, which represents the number of closed ports, as well as with a number of TCP segments with the SYN- and ACK-flag representing the number of open ports. To discover how many ports are open, a TCP segment with the SYN- and ACK-flag is sent to the idle host, which will answer with a TCP segment with the RST-flag and its IPID. By analyzing how often the IPID increased compared to the last IPID received from the idle host, Nmap is able to determine how many of the scanned ports are open.

At this point, it is known how many of the scanned ports are open, but not which ones. In case the received IPID incremented only once, it can be concluded that no TCP segment with the RST-flag was sent by the idle host, so none of the scanned ports are open, and all tested ports are marked as closed. If the IPID incremented further, a binary search is executed to determine which of the scanned ports are open. For this, the amount of the previously scanned ports is split in half, and the scan is repeated for the first half of the ports. In case the IPID incremented by the same amount as in the previous step, it can be concluded that all open ports are on the first half, and all ports on the second half are marked as closed. If one part shows open ports, this part is again split into half, and the process is repeated until all open ports are discovered.

While adding a huge amount of complexity as well as additional messages required to determine the open ports, this method decreases the time needed for a port scan by the order of magnitude [29]. This is due to the fact that when scanning a big number of ports, only some of them will be open. Figure 15 shows the scanning method for the TCP Idle Scan being used in Nmap.

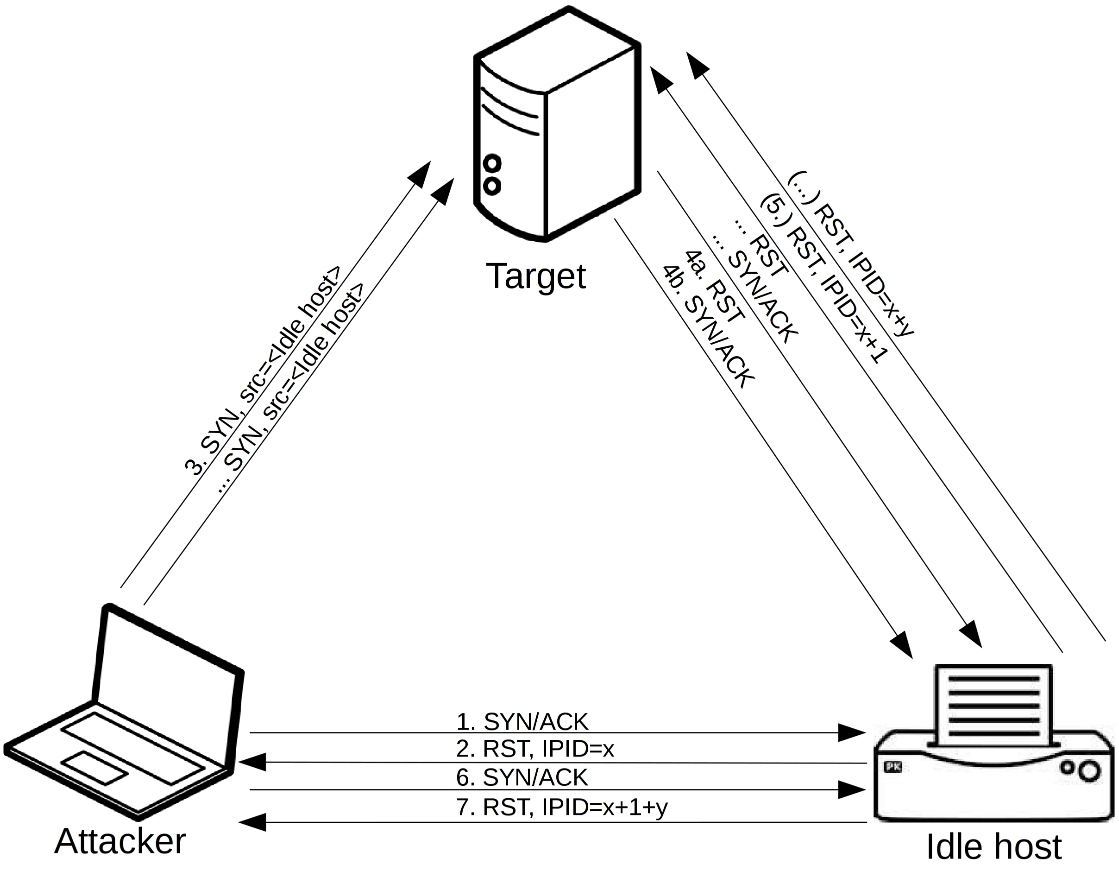


Figure 15: TCP Idle Scan in IPv4 with Nmap

6.4.2. Changing existing process

When adding the functionality for the TCP Idle Scan in IPv6 to Nmap, the basic functionality provided for IPv4 could be reused. Still, all differences between the TCP Idle Scan in IPv4 and IPv6 had to be considered and the original implementation had to be adapted to be able to perform both types of port scans.

- The biggest change was to force the use of the IPv6 extension header for fragmentation before any other task could be performed. This was not only done between the idle host and the target, but also between the idle host and the attacker. The reason for this was to be able to use the existing code, which made use of a TCP segment with the SYN- and ACK-flag to receive an answer containing an IPID, instead of using a fragmented ICMPv6 Echo Request as this was shown in Figure 8 on page 21. Additionally, the use of TCP segments with the SYN- and ACK-flag decreases the amount of bytes being sent between the idle host and the attacker compared to using ICMPv6 Echo Requests with a big amount of data, as discussed in Section 3.5.

  At first, an ICMPv6 Echo Request with a total size of 1280 bytes is created and sent to the idle host. After receiving the response, an ICMPv6 Packet Too Big message is sent to the idle host, informing it that the MTU for the path is lower than the minimum IPv6 MTU. This will force the idle host to append the extension header for fragmentation to every IPv6 packet sent to the attacker.

  Next, the same ICMPv6 Echo Request is sent to the idle host again, but the source address is spoofed to be the one from the target. Afterwards, an ICMPv6 Packet Too Big message with the source address of the target and an MTU smaller than the IPv6 minimum MTU is spoofed, as described in Section 3.2.3, and sent to the idle host.

  The issue in this step is the timing. It is difficult to evaluate how long it should be paused between sending the ICMPv6 Echo Request and the ICMPv6 Packet Too Big message. If the time span is too short, the idle host will not have enough time to send an ICMPv6 Echo Response to the target, and will therefore not yet expect the ICMPv6 Packet Too Big message. If the time span is too long, the scan will be needlessly delayed.

  However, testing with different Windows operating systems as idle host and different waiting periods has shown that even if the ICMPv6 Packet Too Big message is sent immediately after the ICMPv6 Echo Request, the idle host still switches into the desired behavior of appending the IPv6 extension header for fragmentation to every IPv6 packet. This is probably due to the fact that incoming messages are handled one after each other in the order they arrive. Therefore, as long the ICMPv6 Echo Request is sent before the ICMPv6 Packet Too Big message, the probability for it to arrive first is high. But it still needs to be considered that the IP protocol does not guarantee that both ICMPv6 packets will take the same route and therefore arrive in the same order as they were sent [48]. To increase the probability of the messages to arrive in the desired order, a latency of 10 milliseconds was introduced before sending the ICMPv6 Packet Too Big message.

  After sending the ICMPv6 Packet Too Big messages, the idle host will append the extension header for fragmentation to every IPv6 packet sent to the target as well as to the attacker. This allows to execute the actual scan by using the same sequence of messages as in IPv4, which was shown in Figure 15, and keep the necessary changes to a minimum. The function which executes the described steps is shown in Appendix A.

- Another major chance was the access of the *identification* value. In IPv4, the IPID used for the TCP Idle Scan is located at a defined place in the IPv4 header and can therefore easily be accessed. In IPv6, the *identification* value used for the TCP Idle Scan is located in the extension header for fragmentation, which might have various previous and successive extension headers, which are of a variable size. To locate the *identification* value within this chain of extension headers, a function was created which investigates the type and size of the IPv6 extension headers in a loop. In case the extension header is the one for fragmentation, the *identification* value is returned, and otherwise the length of the extension header is determined and the next extension header in the chain is accessed. This process is repeated until the extension header for fragmentation is found or the end of the extension header chain is reached. Appendix B shows the function which finds the *identification* value within an IPv6 packet.

- It also had to be considered that while the IPID in IPv4 has a size of 16 bits, the *identification* value in the IPv6 extension header for fragmentation is 32 bits long. Mostly, the TCP Idle Scan implementation in Nmap used variables of the type *integer* to store the IPID. The C++ standard defines that variables from this type have to have a size of at least 32 bits, which is enough for the IPID in IPv4, and also enough to store the *identification* value in IPv6. But on some occasions, the data type *u16* was used to store the IPID value of IPv4. Code using this data type was changed into using the data type *u32* which has a size of 32 bits and is therefore able to store the IPID of the IPv4 header as well as the *identification* value of the IPv6 extension header for fragmentation [24].

- Additionally, a new detection mechanism for sequential increments was added. Previously, Nmap was only able to detect predictable IPID values if those were incremented either by one or by 256. With five of the tested operating systems incrementing by two, it was necessary to also create detection for such a behavior. During the implementation, the method being used to detect a sequential increment of 256 could be used, and only needed slight adaptation.

After applying all the described changes, Nmap is able to execute the TCP Idle Scan in IPv6. The full patch as well as further developments are available at [32].

6.4.3. Performance

With Nmap being able to execute the TCP Idle Scan in IPv6, tests have been conducted to ensure the correct behavior as well as to determine the performance of the implementation. For this purpose, three virtual machines running on the same physical system have been used. For the attacker, a Debian Linux system running kernel 3.2 was used, and for the target another Debian Linux running kernel 3.8. The target had one open port on IPv4 as well as on IPv6, which was the SSH-daemon[1] on port 22. For the idle host, different operating systems have been tested, which all led to the desired output stating either the correct result or a reason why the scan could not be executed, such as a per-host assignment of the *identification* value from the idle host.

To test the performance of the implementation, Windows 7 has been chosen as operating system for the idle host. The time spans needed to scan 1000 ports were slightly fluctuating. When executing the scan 50 times with pauses of one minute in between, the shortest time span was 7.69 seconds to execute the whole scan, while the longest was 9.20 seconds. On average, the TCP Idle Scan in IPv6 took 8.13 seconds by using the developed patch.

In order to create a valid comparison, the TCP Idle Scan in IPv4 has also been executed 50 times by using the same virtual machines. The fastest of these 50 scans was finished within 7.46 seconds, while the slowest took 10.35 seconds to finish. On average, an execution time of 8.06 seconds was reached for the TCP Idle Scan in IPv4.

Compared to the TCP Idle Scan in IPv4, the implementation created along with this thesis was on average 0.07 seconds slower while delivering the same results. The delay is caused by the message sequence in the beginning, which is used to force the idle host to append the extension header for fragmentation to every IPv6 packet sent to the target and to the attacker. Before being able to send the ICMPv6 Packet Too Big message to the idle host, the attacker once has to wait until the answer for the ICMPv6 Echo Request is received. For forcing the extension header for fragmentation between the idle host and the target, another delay had to be placed to ensure that the ICMPv6 Packet Too Big message will be received by the idle host after the ICMPv6 Echo Request.

However, the delay of 0.07 seconds compared to the TCP Idle Scan in IPv4 illustrates a decrease in performance by less than 1%. Considering the less restricting requirements on the idle host, this can be seen as a negligible decrease in performance.

[1]Secure SHell

7. Defense mechanisms

After discussing various port scanning methods and their characteristics, this section deals with defense mechanisms against those. Short-term defenses for system administrators are as well discussed as long-term defenses which need to be applied by vendors.

7.1. General defense mechanisms

- All four scans introduced in this work require the attacker to be able to spoof the source addresses of some packets. Mechanisms against IP address spoofing are therefore a first defense mechanism. To prevent IP spoofing within a network, administrators can use techniques such as *Reverse Path Forwarding*. This technique checks for the source address of each received packet if the interface on which the packet was received equals the interface which would be used for forwarding a packet to this address [7]. Systems which offer such a defense against IP source address spoofing are for instance Cisco, Linux and OpenBSD [7] [22] [37]. Outside of internal networks, an approach to prevent IP source address spoofing is networking ingress filtering, which should be done by the Internet Service Provider (ISP), as suggested in RFC 2267 [14].

- Another defense which applies to all introduced scans is related to accepting TCP segments with the SYN- and ACK-flag without precedent traffic. In the TCP Idle Scan in IPv4 and IPv6 as well as in the RST Rate Limit Scan, the idle host is expected to reply to such received TCP segments with a segment containing the RST-flag, as those messages where unexpected. If the idle host would drop TCP segments with the SYN- and ACK-flag without precedent traffic instead of answering with a TCP segment with the RST-flag, an attacker would not be able to conclude if the idle host received a TCP segment with the RST-, or with the SYN- and ACK-flag from the target.

  A similar behavior of the target is required while executing the SYN Cache Scan: In reply to a received TCP segment with the SYN- and ACK-flag, it is supposed to either answer with a segment containing the RST-flag in case of an open port, or to drop the message in case of a closed port. To anticipate such a behavior, stateful inspection firewalls can be used in both cases, as described by Stallings [46]. This type of firewall saves the state of each connection, and will therefore only accept a TCP segment with the SYN- and ACK-flag from one party if a segment with the SYN-flag was sent previously from the other party.

7.2. Defense mechanisms against the TCP Idle Scan in IPv6

- Regarding the TCP Idle Scan in IPv6, the most effective defense is a random assignment of the *identification* value in the IPv6 extension header for fragmentation. Using this method, an attacker will not be able to predict the upcoming values of the *identification* values assigned by the idle host. Being unable to predict those values, it is impossible for the attacker to determine if the idle host sent a TCP segment with the RST-flag to the target, as this is done in step 8 of Figure 8. However, this is a long-term defense mechanism, where the responsibility of implementation relies on the vendor instead of the administrator.

7.3. Defense mechanisms against the RST Rate Limit Scan

- To protect against the RST Rate Limit Scan, one method would be to remove the RST rate limit from operating systems. But before executing this step, it should be understood why an operating system makes use of such a limit in the first place. If a host is exposed to a traditional port scan [29], an attacker tries to create TCP connections directly from his source address with TCP ports on the idle host. For each connection attempt received on a closed port, the idle host will reply with a TCP segment with the RST-flag, which lets the attacker conclude that this port is closed. By using an RST rate limit, the idle host will only give information about a certain amount of closed ports per second, making the port scan more time-consuming.

  For this reason, it is not advisable to remove the RST rate limit, which protects from traditional port scans, in order to protect from the RST Rate Limit Scan. Instead, a better defense would be a combined RST rate limit, which considers as well received as sent TCP segments with the RST flag, as done for example by OpenBSD. This behavior makes it impossible to execute the RST Rate Limit Scan with this idle host. Independent if the scanned port on the target is open or closed, the idle host will each time either receive or send a TCP segment with the RST flag, see Figure 12, and increase its counter for the RST rate limit. Therefore, the idle hosts final answer to the attacker in step 7 will always remain the same, independent from the behavior of the target, which makes the RST Rate Limit Scan ineffective.

  Additionally, a RST rate limit which also considers incoming TCP segments will also protect against other attacks which utilize the RST-flag, such as RST Hijacking [13]. As stated by Tanenbaum [48], TCP segments with the RST-flag should not occur in normal TCP traffic, as those TCP segments indicate either networking problems or attacks. Therefore, such a combined RST rate limit can be kept relatively low to handicap attacks using TCP segments with the RST-flag.

7.4. Defense mechanisms against the SYN Cache Scan

- There are various mechanisms in order to defend against the SYN Cache Scan,. What enables an attacker to execute the port scan is that the the data being sent on the network changes after the idle host's backlog queue is full because of the host falling back to sending SYN-Cookies. One method to change this behavior is not to use a backlog queue at all, as done for instance by FreeBSD and OpenBSD. Both use different defense mechanisms against SYN-Flooding, but use those for all received connection requests, which means that the behavior does not change after a certain amount of half-open connections.

- Another defense against the SYN Cache Scan is to maintain the behavior which the idle host shows on the network after falling back to using the protection mechanism. For the scan to work, the attacker needs to be able to detect in the last step if the idle host answered with a SYN-Cookie or not, see Figure 13. By using alternatives to SYN-Cookies, such as a SYN Cache, which do not influence the data being sent on the network, an attacker will not be able to detect if the idle hosts backlog queue is full and therefore if the port on the target is open.
- To avoid an external attacker scanning internal networks, as shown in Figure 14 on page 42, it is necessary to analyze the interface as well as the source address of received IP packages. An example for this is the already mentioned *Reverse Path Forwarding*, which checks if the interface on which an IP packet is received is the same as the one which would be used to send the reply. Taken as example the firewall in Figure 14, the system should be able to detect that it is not possible that an IP packet which is received from the external network has an internal source address. Therefore, to avoid an external attacker to be able to scan an internal network, elaborated configuration of the gateway to the external network is necessary.
- The last defense against the SYN Cache Scan does not need to be applied on the idle host, but on the target. In order to be able to execute the scan, the target is required to show different behavior compared to if a TCP segments with the SYN- and ACK-flag is received on an open, or on a closed port. Examples for this are FreeBSD's *blackhole* [41] and poorly configured firewalls. To prevent an attacker from executing the SYN Cache Scan, it is essential that the target shows the same behavior independent from if the TCP segment from the idle host was received on an open, or on a closed port.

To sum up, there are various defense mechanisms against the different types of port scans described. On the short term, it is advisable for system administrators to take measures against IP source address spoofing, use stateful inspection firewalls, review the used mechanisms against SYN-Flooding attacks and analyze the behavior of hosts when receiving TCP segments with the SYN- and ACK-flag on open and closed ports. Additionally, to prevent from external intrusion into internal networks, the forwarding policy of the gateway to the external network should be audited.

On the long term, vendors are advised to design the *identification* values in the IPv6 extension header for fragmentation in a way which is unpredictable for an attacker. Besides that, a combined RST rate limit for incoming and outgoing TCP segments as well as defense mechanisms against SYN-Flooding which do not influence the data being sent on the network would be desirable.

8. Conclusion

This thesis has shown that by clever use of some IPv6 features, the TCP Idle Scan can successfully be transferred from IPv4 to IPv6. Therefore, this type of port scan remains also in IPv6 a powerful tool in the hands of an attacker who wants to cover his tracks, and a challenge for anybody who tries to trace back the scan to its origin. The fact that major operating systems assign the *identification* value in the extension header for fragmentation in a predictable way also drastically increases the chances for an attacker to find a suitable idle host for executing the TCP Idle Scan in IPv6. Because the idle host is also not required to be completely idle, but only expected not to create IPv6 traffic using the extension header for fragmentation, these chances are increased additionally.

What remains is the question why it is still a common practice to utilize predictable *identification* values. The danger of predictable sequence numbers has already been disclosed by Morris [34] in 1985. Although his article covered TCP, the vulnerabilities were caused by the same problem: a predictable assignment of the sequence number. For this reason, he advised to use random sequence numbers. With the TCP Idle Scan in IPv4 being first discovered in 1998, it has been shown that the necessity of unpredictable *identification* values also applies to IPv4. This article has shown that also in IPv6, predictable *identification* values facilitate attacks and should be substituted with random values.

Besides the TCP Idle Scan in IPv4 and IPv6, two alternative scanning methods have been presented, namely the RST Rate Limit Scan and the SYN Cache Scan. Table 4 compares the properties of all four scanning types. While the TCP Idle Scan in IPv4 and IPv6 utilizes the predictability of the IPID or *identification* value intended for fragmentation, the RST Rate Limit Scan requires the idle host to enforce an RST Rate Limit, and the SYN Cache Scan requires the usage of the backlog queue with a limited size.

Regarding idleness of the idle host, the port scan methods have different requirements. With the TCP Idle Scan in IPv4, the idle host is required to remain completely idle, as each sent IPv4 packet influences its IPID. If traffic is exchanged with a third party, the scan will return unexpected results. Compared to this, the TCP Idle Scan in IPv6 only requires the idle host not to create IPv6 traffic which makes use of the IPv6 extension header for fragmentation. When using the RST Rate Limit Scan, the idle host is not required to remain idle at all, as it can be assumed that casual traffic does not require to send TCP segments with the RST-flag [48]. Within the SYN Cache Scan, the idle host is not allowed to receive incoming TCP connection requests, as those would influence the available place in the backlog queue. However, outgoing traffic does not affect the scanning results.

The port scans also differ by the number of message which are necessary to scan the first port. For the TCP Idle Scan in IPv4, those are seven messages, whereas the TCP Idle Scan in IPv6 requires a total of 11 messages. The total amount of messages needed to scan the first port with the RST Rate Limit Scan depends on the RST Rate Limit of the idle host. To force the idle host to send all but one TCP segments with the RST-flag, an amount of the RST Rate Limit but one messages, which will be replied by the idle host, is necessary. After this is done, another five messages are required to execute the actual scan. For the SYN Cache Scan, a total amount of the size of the idle host's backlog queue but one messages is required to fill the backlog queue, to which the idle host will reply with TCP segments with the SYN- and ACK-flag. Afterwards, five more messages are required to execute the actual scan. Additionally, it needs to be considered that the idle host might resend the TCP segments with the SYN- and ACK-flag, which will increase the amount of messages additionally.

To scan further ports, the amount of messages can be decreased to five for the TCP Idle Scan in IPv4 as well as for the TCP Idle Scan in IPv6. This also applies to the SYN Cache Scan as long as the backlog queue was not filled by previously scanning a port which did not return a TCP segment with the RST-flag from the target to free an entry in the idle host's backlog queue. For the RST Rate Limit Scan, it is necessary to send the same amount of messages as when scanning the first port, since the RST rate limit will be reset at all tested operating systems after one second. Additionally, one will have to wait until the RST rate limit is reset before scanning a second port, as the idle host will have sent the maximum number of TCP segments with the RST-flag after scanning the first port.

Among all port scanning methods, the stealthiest is the TCP Idle Scan in IPv4. The TCP Idle Scan in IPv6 reaches similar results regarding stealthiness, but has the problem that it requires to send the unusual ICMPv6 Packet Too Big messages with an MTU smaller than 1280, which might raise the attention of an administrator. On the third position regarding stealth is the RST Rate Limit Scan, as the unusually high amount of TCP segments with the RST-flag is also likely to raise the attention of an administrator. However, the least stealthy scan is the SYN Cache Scan, as a full backlog queue will be reported by most operating systems with the warning of a suspected SYN-Flooding attack, and is therefore highly likely to raise the attention of the administrator.

When analyzing the difficulty of tracing back a scan to its origin, all four scans provide the same level of protection for an attacker. For all of the scans, not a single IP-packet containing the attacker's IP-address is sent to the target. To be able to discover the attacker's IP-address, access to the idle host is required, and even then, most of the used messages such as TCP segments with the SYN- and ACK-flag are unlikely to be logged by the target system. Therefore, to trace back the scan to its origin, one would be required to run for example a network sniffer on the idle host while the scan is executed in order to be able to determine its source.

Compared to the other three scans, the SYN Cache Scan is the only one which requires the idle host to have at least one open TCP-port in order to be able to fill the backlog queue of the host. Also, it is the only scanning method that requires a specific behavior from the target, which is to respond differently depending on if a TCP segment with the SYN- and ACK flag is received on an open or on a closed port. Additionally, the SYN Cache Scan is also the only one from the four presented scans which needs to be adapted depending on the operating system the idle host is running and the defense mechanism against SYN-Flooding it is using. However, although requiring all those additional properties, the SYN Cache Scan has the advantage that the attacker is not required to be able to send packets to the target, which might enable an attacker to scan internal networks. Summed up, it can be said that all port scans shown in this thesis provide advantages as well as disadvantages. It will depend on the situation which method should be preferred after considering variables such as the operating system and behavior of the target and the idle host as well as details such as the network architecture.

With implementing proofs of concept for the TCP Idle Scan in IPv6, the RST Rate Limit Scan and the SYN Cache scan, it has been shown that the scanning methods are also feasible in practice. Additionally, the patch created for the security scanner Nmap along with this thesis has shown that the newly discovered TCP Idle Scan in IPv6 is only slightly slower than the TCP Idle Scan in IPv4 by having less requirements for the idle host.

Until vendors are able to implement the long term defense mechanisms described in Section 7, administrators are advised to make use of the short-term protection mechanisms to ensure a full protection against the port scanning methods discussed.

In the future, one might consider an update of RFC 1981, which forces a host to append an empty fragmentation header to every IPv6 packet after receiving an ICMPv6 Packet Too Big message with an MTU smaller than the IPv6 minimum MTU. Likewise, updating RFC 2460 towards an obligatory random assignment of the *identification* value in the extension header for fragmentation should be considered as well.

	TCP Idle Scan IPv4	**TCP Idle Scan IPv6**	**RST Rate Limit Scan**	**SYN Cache Scan**
Utilizes	IPID	*identification* value	RRL	BQL
Idle host required to be idle?	✓	~	×	~
No. of messages to scan first port	7	11	(RRL - 1) * 2 + 5	>= (BQL - 1) * 2 + 5
No. of messages to scan second port	5	5	(RRL - 1) * 2 + 5	5
Stealth Level	+	o	-	- -
Attacker remains untraceable?	✓	✓	✓	✓
Open port on idle host required?	×	×	×	✓
Specific behavior of target required?	×	×	×	✓
Depends on OS of the idle host?	×	×	×	✓
Attacker needs to reach target?	✓	✓	✓	×

Legend:
RRL: RST rate limit
BQL: Backlog Queue Limit

Table 4: Overview over the characteristics of the discussed scans

8.1. Future work

The RST Rate Limit Scan and the SYN Cache Scan have shown that the TCP Idle Scan is not the only possibility for an attacker to execute a stealthy port scan. In the future, it would be interesting to research if similar scanning methods, which make use of a third host to hide the real IP address of an attacker are feasible. What would be needed is an idle host that sets a limit or a counter for a specific action and is influenced differently by receiving TCP segments with the SYN- and ACK-flag or segments with the RST-flag.

For this purpose, an investigation of other IPv6 extension headers to analyze if one of those provides alternatives to the *identification* value in the extension header for fragmentation would be a first approach. Also, extension headers could be used in order to hide port scans from Intrusion Detection Systems (IDS) such as snort [6] or to infiltrate into firewalls [17], which could also be investigated in the future and might even be successfully merged with the TCP Idle Scan in IPv6.

Acronyms

ACK	Acknowledge
DoS	Denial of Service
FH	Extension header for fragmentation
ICMP	Internet Control Message Protocol
ICMPv6	Internet Control Message Protocol version 6
ID	Identification
IP	Internet Protocol
IPID	Identification value for fragmentation
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
MTU	Maximum Transfer Unit
PMTU	Path Maximum Transfer Unit
RFC	Request for Comments
RST	Reset
SYN	Synchronize
TCP	Transmission Control Protocol

References

[1] Filipe Almeida. idlescan (ip.id portscanner). `http://seclists.org/bugtraq/1999/Dec/58`, 1999. [Online; Request on July, 22nd of 2013].

[2] Apple Inc. xnu-1504.9.37 - bsd/netinet6/icmp6.c, 2008.

[3] Steven M. Bellovin. Security Problems in the TCP/IP Protocol Suite. *Computer Communications Review*, pages 32–48, 1989.

[4] Daniel J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`. [Online; Request on April, 23rd of 2013].

[5] Bryan Burns, Eric Markham, Chris Iezzoni, Philippe Biondi, Jennifer Stisa Granick, Steve Manzuik, Paul Guersch, Dave Killion, Nicolas Beauchesne, Eric Moret, Julien Sobrier, and Michael Lynn. *Security power tools*. O'Reilly, first edition, 2007.

[6] Jon Christmas. Topera the sneaky IPv6 Port Scanner. `http://www.soleranetworks.com/blogs/topera-the-sneaky-ipv6-port-scanner/`, 2013. [Online; Request on July, 31st of 2013].

[7] Cisco Systems, Inc. Understanding Unicast Reverse Path Forwarding. `http://www.cisco.com/web/about/security/intelligence/unicast-rpf.html`, 2013. [Online; Request on May, 10th of 2013].

[8] Alex Conta, Stephen Deering, and Mukesh Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (Draft Standard), March 2006. Updated by RFC 4884.

[9] Joseph G. Davies. *Understanding IPv6*. Microsoft Press, Redmond, WA, USA, third edition, 2012.

[10] Stephen Deering and Robert Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998.

[11] Wesley Eddy. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Draft Standard), August 2007.

[12] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. Idle Port Scanning and Non-Interference Analysis of Network Protocol Stacks Using Model Checking. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 257–272, Berkeley, CA, USA, 2010. USENIX Association.

[13] Jon Erickson. *Hacking: the art of exploitation, 2nd edition*. No Starch Press, San Francisco, CA, USA, second edition, 2008.

[14] Paul Ferguson and Daniel Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2267 (Draft Standard), January 1998.

[15] Yossi Gilad and Amir Herzberg. Fragmentation considered vulnerable: blindly intercepting and discarding fragments. In *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, pages 9–18, Berkeley, CA, USA, 2011. USENIX Association.

[16] Fernando Gont. Processing of IPv6 "Atomic" Fragments. RFC 6946 (Draft Standard), May 2013.

[17] Fernando Gont and Marc Heuse. Security Assessment of IPv6 Networks and Firewalls. `http://www.si6networks.com/presentations/ipv6kongress/mhfg-ipv6-kongress-ipv6-security-assessment.pdf`, 2013. [Online; Request on July, 31st of 2013].

[18] Silvia Hagen. *IPv6 Essentials.* O'Reilly Media, 2009.

[19] heise Netze. IPv6-Kongress, 2013.

[20] Marc Heuse. Recent advances in IPv6 insecurities. `http://media.ccc.de/browse/congress/2010/27c3-3957-en-ipv6_insecurities.html`, 2010. [Online; Request on April, 5th of 2013].

[21] Charles Hornig. A Standard for the Transmission of IP Datagrams over Ethernet Networks. RFC 894 (Draft Standard), April 1984.

[22] Bert Hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B. Schroeder, and Jasper Spaans. Reverse Path Filtering. `http://tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.kernel.rpf.html`, 2013. [Online; Request on May, 10th of 2013].

[23] Informa UK. IPv6 World Congress, 2013.

[24] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++.* International Organization for Standardization, Geneva, Switzerland, February 2012.

[25] Erik J. Kamerling. The Hping2 Idle Host Scan . `http://www.ouah.org/hping2idle.htm`, 2001. [Online; Request on June, 27th of 2012].

[26] Jonathan Lemon. Resisting SYN flood DoS attacks with a SYN cache. In *Proceedings of the BSD Conference 2002 on BSD Conference*, BSDC'02, pages 89–97, Berkeley, CA, USA, 2002. USENIX Association.

[27] Gordon Lyon. *The Official Nmap Project Guide to Network Discovery and Security Scanning.* Nmap Project, 2012.

[28] Gordon Lyon. Nmap - Free Security Scanner For Network Exploration and Security Audits. `http://www.nmap.org/`, 2013. [Online; Request on August, 1st of 2013].

[29] Gordon Lyon. Nmap Reference Guide. http://nmap.org/book/man.html, 2013. [Online; Request on August, 1st of 2013].

[30] Jack McCann, Stephen Deering, and Jeffrey Mogul. Path MTU Discovery for IP version 6. RFC 1981 (Draft Standard), August 1996.

[31] Microsoft Corporation. Syn attack protection on Windows Vista, Windows 2008, Windows 7 and Windows 2008 R2. http://blogs.technet.com/b/nettracer/archive/2010/06/01/syn-attack-protection-on-windows-vista-windows-2008-windows-7-and-windows-2008-r2.aspx, 2010. [Online; Request on May, 7th of 2013].

[32] Mathias Morbitzer. Nmap Development: [PATCH] TCP Idle Scan in IPv6. http://seclists.org/nmap-dev/2013/q2/394 and http://seclists.org/nmap-dev/2013/q3/13, 2013. [Online; Request on August, 1st of 2013].

[33] Mathias Morbitzer. TCP Idle Scanning using network printers. http://www.researchgate.net/publication/239660144_TCP_Idle_Scanning_using_network_printers/file/3deec51c1c17e29b78.pdf, 2013. Research Paper, Radboud University of Nijmegen. [Online; Request on July, 7th of 2013].

[34] Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software, 1985.

[35] Thomas Narten, Erik Nordmark, William A. Simpson, and Hesham Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (Draft Standard), September 2007.

[36] OpenBSD. OpenBSD Programmer's Manual: sysctl(3). http://www.openbsd.org/cgi-bin/man.cgi?query=sysctl&sektion=3, 2013. [Online; Request on April, 25th of 2013].

[37] OpenBSD. PF: Packet Filtering. http://www.openbsd.org/faq/pf/filter.html#urpf, 2013. [Online; Request on May, 10th of 2013].

[38] Jon Postel. Internet Protocol. RFC 791 (Draft Standard), September 1981.

[39] Jon Postel. Transmission Control Protocol. RFC 793 (Draft Standard), September 1981.

[40] Jon Postel. The TCP Maximum Segment Size and Related Topics. RFC 879 (Draft Standard), March 1983. Updated by RFC 6691.

[41] Geoffrey M. Rehmet. FreeBSD Manual Pages for BLACKHOLE(4). http://www.unix.com/man-page/FreeBSD/4/BLACKHOLE/, 2007. [Online; Request on May, 7th of 2013].

[42] Pedro Roque. Linux 3.8 - net/ipv6/route.c, 2013.

[43] Salvatore Sanfilippo. New TCP scan method. http://seclists.org/bugtraq/1998/Dec/79, 1998. [Online; Request on July, 22nd of 2013].

[44] Salvatore Sanfilippo. Hping - Active Network Security Tool. `http://www.hping.org/`, 2006. [Online; Request on July, 31st of 2013].

[45] SI6 Networks. SI6 Networks' IPv6 Toolkit. `http://www.si6networks.com/tools/ipv6toolkit/`, 2013. [Online; Request on April, 3rd of 2013].

[46] William Stallings. *Network Security Essentials - Applications and Standards (4. ed., internat. ed.)*. Pearson Education, 2010.

[47] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1-2):105–136, July 2002.

[48] Andrew Tanenbaum. *Computer networks*. Prentice Hall PTR, 5th edition, 2011.

[49] Upperside Conferences. V6 World Congress, 2013.

[50] Johannes Weber. IPv6 Security Test Laboratory. Masterthesis, Ruhr-University Bochum, Germany, February 2013.

[51] Wireshark Foundation. Wireshark. Go deep. `https://www.wireshark.org/`, 2013. [Online; Request on August, 1st of 2013].

[52] Hai Zhang, Xuyang Zhu, and Wenming Guo. TCP portscan detection based on single packet flows and entropy. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, pages 1056–1060, New York, NY, USA, 2009. ACM.

A. Forcing atomic fragments

Listing 4: Function to force the use of the extension header for fragmentation

```c
/* Forces the permanent use of the IPv6 extension header for
 * fragmentation in each IPv6 packet sent from the idle host
 * to the target or the attacker. This is achieved by first
 * sending a ping, and afterwards an ICMPv6 Packet Too Big
 * message which states that the response from the ping was
 * too big, our MTU is smaller than the IPv6 minimum MTU */
static void ipv6_force_fragmentation(struct idle_proxy_info *proxy, Target *target) {
  int hardtimeout = 9000000; /* Generally don't wait more than 9 secs total */
  char filter[512]; /* Libpcap filter string */
  struct ip *ip;
  /* The maximum data size we can create without fragmenting, considering that the headers also need place */
  char data[IP6_MTU_MIN - IPv6_HEADER_LEN - ETH_HDR_LEN - ICMPv6_MIN_HEADER_LEN];
  unsigned int datalen, bytes;
  const unsigned int proxy_reply_timeout = 2000;
  const void *rdata; //the data received in the echo response
  struct timeval tmptv, rcvdtime, ipv6_packet_send_time;
  struct abstract_ip_hdr hdr;
  bool response_received = false;
  struct icmpv6_hdr *icmp6_header;
  u8 *ipv6_packet = NULL;
  u32 packetlen = 0;
  u16 pingid = 0;
  u16 seq = 0;
  struct sockaddr_storage ss;
  size_t sslen;
  int res;
  assert(proxy);

  /* First, we force the proxy to provide us with a
   * fragmentation header in each packet by sending a ping
   * and afterwards an ICMPv6 Packet Too Big */
  memset(data, 'A', sizeof(data));
  pingid = get_random_u16();
  seq = get_random_u16();

  /* pcap to get the answer. Max size here is the IPv6 minimum MTU */
  if ((proxy->pd = my_pcap_open_live(proxy->host.deviceName(), IP6_MTU_MIN, (o.spoofsource) ? 1 : 0, 50)) == NULL)
    fatal("%s", PCAP_OPEN_ERRMSG);

  Snprintf(filter, sizeof(filter), "icmp6 and src host %s and dst host %s", proxy->host.targetipstr(), proxy->host.sourceipstr());
  if (o.debugging)
    log_write(LOG_STDOUT, "Packet capture filter (device %s): %s\n", proxy->host.deviceFullName(), filter);

  /* Make a ping that is in total 1280 byte long and send it */
  proxy->host.TargetSockAddr(&ss, &sslen);
  ipv6_packet = build_icmpv6_raw(proxy->host.v6sourceip(), proxy->host.v6hostip(), 0x00, 0x0000, o.ttl, seq , pingid, ICMPV6_ECHO, 0x00, data, sizeof(data) , &packetlen);
  res = send_ip_packet(proxy->rawsd, proxy->ethptr, &ss, ipv6_packet, packetlen);
  if (res == -1)
    fatal("Error occured while trying to send ICMPv6 Echo Request to the idle host");
  free(ipv6_packet);
  gettimeofday(&ipv6_packet_send_time, NULL);

  /* Now let's wait for the answer */
  while (!response_received) {
    gettimeofday(&tmptv, NULL);
    ip = (struct ip *) readip_pcap(proxy->pd, &bytes, proxy_reply_timeout, &rcvdtime, NULL, true);
    if (!ip) {
      if (TIMEVAL_SUBTRACT(tmptv, ipv6_packet_send_time) >= hardtimeout) {
          fatal("Idle scan zombie %s (%s) port %hu cannot be used because it has not returned any of our ICMPv6 Echo Requests -- perhaps it is down or firewalled.",
             proxy->host.HostName(), proxy->host.targetipstr(),
             proxy->probe_port);
      }
      continue;
    }
    datalen = bytes;
    rdata = ip_get_data(ip, &datalen, &hdr);
    if (hdr.version == 6 && hdr.proto == IPPROTO_ICMPV6) {
      icmp6_header = (struct icmpv6_hdr *) rdata;
      if (icmp6_header->icmpv6_type == ICMPV6_ECHOREPLY) {
        const struct icmpv6_msg_echo *echo;
        echo = (struct icmpv6_msg_echo *) ((u8 *) icmp6_header + sizeof(*icmp6_header));
        if (ntohs(echo->icmpv6_id) == pingid && ntohs(echo->icmpv6_seq) == seq)
          response_received=true;
      }
    }
  }

  if (proxy->pd)
    pcap_close(proxy->pd);

  /* Now we can tell the idle host that its reply was too big,
   * we want it smaller than the IPV6 minimum MTU. The data
   * contains first the MTU we want, and then
   * the received IPv6 package */
  *(uint32_t *)&data = ntohl(IP6_MTU_MIN - 2);
  memcpy(&data[4], ip, sizeof(data)-4);

  ipv6_packet = build_icmpv6_raw(proxy->host.v6sourceip(), proxy->host.v6hostip(), 0x00, 0x0000, o.ttl, 0x00 , 0x00, 0x02, 0x00, data, sizeof(data) , &packetlen);
  res = send_ip_packet(proxy->rawsd, proxy->ethptr, &ss, ipv6_packet, packetlen);
  if (res == -1)
    fatal("Error occured while trying to send spoofed ICMPv6 Echo Request to the idle host");

  free(ipv6_packet);

  /* Now we do the same in the name of the target */
  /* No pcap this time, we won't receive the answer */
  memset(data, 'A', sizeof(data));
  pingid = get_random_u16();
  seq = get_random_u16();

  ipv6_packet = build_icmpv6_raw(target->v6hostip(), proxy->host.v6hostip(), 0x00, 0x0000, o.ttl, seq , pingid, ICMPV6_ECHO, 0x00, data, sizeof(data) , &packetlen);
  res = send_ip_packet(proxy->rawsd, proxy->ethptr, &ss, ipv6_packet, packetlen);
  if (res == -1)
    fatal("Error occured while trying to send ICMPv6 Echo Request to the idle host");

  free(ipv6_packet);

  /* Now we guess what answer the decoy host sent to the target,
   * so that we can piggyback this on the
   * ICMPV6 Packet too Big message */
  ipv6_packet = build_icmpv6_raw(proxy->host.v6hostip(), target->v6hostip(), 0x00, 0x0000, o.ttl, seq , pingid, ICMPV6_ECHOREPLY, 0x00, data, sizeof(data) , &packetlen);
  *(uint32_t *)&data = ntohl(IP6_MTU_MIN - 2);
  memcpy(&data[4], ipv6_packet, sizeof(data)-4);
  free(ipv6_packet);

  ipv6_packet = build_icmpv6_raw(target->v6hostip(), ↙
      ↳ proxy->host.v6hostip(), 0x00, 0x0000, o.ttl, 0x00 , 0x00, ↙
      ↳ 0x02, 0x00, data, sizeof(data) , &packetlen);
  /* give the decoy host time to reply to the target */
  usleep(10000);
  res = send_ip_packet(proxy->rawsd, proxy->ethptr, &ss, ↙
      ↳ ipv6_packet, packetlen);
  if (res == -1)
    fatal("Error occured while trying to send ICMPv6 PTB to the ↙
        ↳ idle host");
  free(ipv6_packet);
}
```

B. Finding the IPv6 Identification value

Listing 5: Search the IPv6 extension header chain for the *identification* value

```
/* Finds the IPv6 extension header for fragmentation in
 * an IPv6 packet, and returns the identification value
 * of the fragmentation header */
int ipv6_get_fragment_id(const struct ip6_hdr *ip6, unsigned int ↙
    ↳ len) {
  const unsigned char *p, *end;
  u8 hdr;
  struct ip6_ext_data_fragment *frag_header = NULL;

  if (len < sizeof(*ip6))
    return -1;

  p = (unsigned char *) ip6;
  end = p + len;

  hdr = ip6->ip6_nxt;
  p += sizeof(*ip6);

  /* If the first extension header is not the fragmentation,
   * we search our way through the extension headers until
   * we find the fragmentation header */
  while (p < end && hdr != IP_PROTO_FRAGMENT) {
    if (p + 2 > end)
      return -1;
    hdr = *p;
    p += (*(p + 1) + 1) * 8;
  }

  if ( hdr != IP_PROTO_FRAGMENT ||  (p + 2 + ↙
      ↳ sizeof(ip6_ext_data_fragment)) > end)
```

```
        return -1;

    frag_header = (struct ip6_ext_data_fragment *)( p + 2 );

    return (ntohl(frag_header->ident));

}
```