IMPROVING THE FUNCTIONALITY OF SYN COOKIES

André Zúquete
IST / INESC-ID Lisboa, Lisboa, Portugal
andre.zuquete@gsd.inesc-id.pt

Abstract Current Linux kernels include a facility called TCP SYN cookies, conceived to face SYN flooding attacks. However, the current implementation of SYN cookies does not support the negotiation of TCP options, although some of them are relevant for throughput performance, such as large windows or selective acknowledgment. In this paper we present an improvement of the SYN cookie protocol, using all the current mechanisms for generating and validating cookies while allowing connections negotiated with SYN cookies to set up and use any TCP options. The key idea is to exploit a kind of TCP connection called "*simultaneous connection initiation*" in order to lead client hosts to send together TCP options and SYN cookies to a server being attacked.

Keywords: SYN flooding attacks, SYN cookies, TCP options, simultaneous connection initiation.

1. INTRODUCTION

Current Linux kernels include a facility called TCP SYN cookies, designed and first implemented by D. J. Bernstein and Eric Schenk [1, 2]. This facility was conceived to face a specific attack on normal TCP/IP networking, known as "*SYN flooding attack*". This is a denial-of-service attack that floods a server host with long-lasting half-open TCP connections. Since kernels usually restrict the number of half-open connections, a SYN flooding attack prevents real client hosts from connecting to a server host being attacked. Furthermore, such attacks are easy to deploy, can be launched anywhere on the Internet and are difficult to avoid by well-known servers.

SYN cookies provide protection against this type of attack and their rationale is straightforward: prevent the denial-of-service scenario by not keeping state about pending connection requests, or half-open connections. The cookies allow a server host to maintain the state of half-open

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35612-9_23 connections outside its memory: such state is (partially) stored inside a cryptographic challenge, the SYN cookie, that is returned to the client within SYN-ACK segments as the server's TCP Initial Sequence Number (ISN). Since TCP requires the client to send back that ISN on the subsequent ACK, the server will be able to restore a half-open connection from a cookie and, consequently, create a final connection descriptor.

Although the rationale behind SYN cookies is straightforward, its implementation is more complicated. First, cookies must fit in the space defined for the ISN field of a TCP header (32 bits). Second, the generation of cookies should respect the TCP recommendations for ISN values being monotonically increasing over time, possibly using some sort of time-based counter, to reduce the probability of delayed segments being accepted by new incarnations of similar connections [3]. Third, cookies must be unpredictable by attackers, to prevent the forgery of valid cookies, which could be used to launch TCP hijacking attacks [4]. And fourth, cookies should contain all TCP options sent by clients on SYN segments and usually kept in half-open connection's descriptors (if supported by servers).

The current implementation of the SYN Cookie Protocol (SynCP hereafter) in Linux systems deals differently with these issues, handling completely or partially some of them, but, unfortunately, ignoring most of them [1]. For instance, some TCP options that are relevant for throughput performance, like large windows or selective acknowledgment, are simply not supported when using SYN cookies. Even the choice of suitable server's Message Size (MSS) is limited. Therefore, a SYN flooding attack, by triggering the use of SYN cookies, can reduce the quality of service provided by a server host (besides forcing the waste of CPU cycles and bandwidth to deal with bogus connection requests). This is clearly an undesirable side effect of using SYN cookies.

In this paper we present an improvement of the current SynCP. This improvement uses most of the current mechanisms for generating and validating cookies; however, it allows connections negotiated with SYN cookies to set up and use TCP options that are relevant for performance but currently ignored. The key idea is to explore a kind of TCP connection called "*simultaneous connection initiation*". But this approach, although fully compatible with standard TCP rules [3], faces two major problems. First, some systems do not deal correctly with simultaneous connection initiations (e.g. Windows systems). Second, client-side firewalls may interfere with the action taken by the server. To overcome these problems we propose a mixed protocol, combining the current and the new SynCPs. Problematic clients are detected and handled differently by the server using a simple cache with their IP addresses.

This paper is organized as follows. Section 2 presents some related work regarding the countermeasures for SYN flooding attacks. Section 3 briefly describes how SYN cookies are currently handled by Linux kernels. Section 4 presents our proposal, starting with the basic protocol, presenting some problems it faces with problematic clients and firewalls, and concluding with the description of the final protocol. Section 5 sketches some implementation details. Finally, in Section 6 we draw some conclusions.

2. RELATED WORK

The generic goal of any solution to SYN flooding attacks is to continue to accept connection requests even when being under attack. There are several ways to achieve this goal, but none of them is perfect. In this section we will shortly describe the approach followed by several proposed solutions, presenting their advantages and drawbacks. The description will focus only on solutions that change the way server kernels deal with the current IPv4 TCP protocol specifications. We will not address any solutions requiring either (i) a modification of the TCP protocol, (ii) a modification of the kernel of client hosts or (iii) filtering policies applied to client hosts in their access to the Internet (e.g. [5]).

One obvious solution, proposed by several vendors, is to use larger queues for pending connection requests. A high bound for the queue's length can be computed from the bandwidth of the server's network connection and the timeout used by the server to discard pending requests. This is a sort of brute-force solution that may waste lots of kernel memory and slow down the server's response time, but it can be effective in public servers serving large communities of clients, since such hosts usually have extensive hardware resources.

A more crafty solution is called Random Drop [6]. The principle is simple: a server always accepts a new connection request and uses the queue of pending requests as a cache, with a random substitution policy to get space for new requests. For each dropped request the server sends an RST to the source host, enabling real clients to react to the server action. This solution allows a flexible trade-off of defense effectiveness with resource requirements, but it only guarantees service in a probabilistic manner. Thus, an attacker may still occasionally affect connections requested by real clients.

Another policy for dropping connection requests is called Reset Cookies [7, 6], using security associations[1] for improving correctness. When under attack, the server host checks a cache of security associations prior to accept and queue each new connection request. If the client is not listed in the cache the server replies with an illegal SYN-ACK with a cookie as the server's ISN. Legitimate clients will reply with an RST containing that cookie, which can then be checked by the server and trigger the creation of a security association with the client. This solution implies the storage and management of a cache of security associations in the server and increases significantly the latency of connections for clients not listed in the cache. Therefore, heavily used public servers should use large caches for reducing the impact of using Reset Cookies, but that is exactly the opposite of what defenses against SYN flooding attacks should do.

3. SYN COOKIES IN CURRENT LINUX KERNELS

The most recent Linux kernel (version 2.4.14), as well as many other previous versions, allows kernel builders to include the generation and analysis of SYN cookies in the kernel functionality, and allows administrators to dynamically activate their use. We will now explain how SYN cookies are used on a kernel were they are enabled, in order to introduce all the problems currently raised by their use. For simplicity, hereafter we will use the term cookie to refer to a SYN cookie.

3.1. Generation and validation of SYN cookies

Cookies are not used when the kernel operates in normal conditions, but only when it suspects of being under a SYN flooding attack[2]. The suspicion is simply derived from the length of queue of pending connection requests: if it reaches a given threshold length, the kernel emits a SYN flood warning and starts using cookies to handle new connection requests.

Cookies have a limited lifetime. When the kernel receives an ACK from a client (the third segment of the three-way handshake), first it checks the segment against the queue of connection requests, and upon failure it may check whether the segment carries a valid cookie. Such

[1]A security association is the IP address of a real client host that initiated a TCP connection to the server in the past.

[2]This is not true if the kernel operates as a so called "*SYN cookie firewall*". In this case, all incoming SYN segments get a SYN cookie reply from the firewall kernel, and only upon the correct reception of the cookie within an ACK the server is contacted by the firewall.

checking takes place only within a short time frame, starting when the last cookie was sent and ending a few seconds later (currently 3 seconds), and cookies are accepted only if their value is among a set of acceptable values.

3.2. Transparent use of SYN cookies

Cookies were designed to tackle SYN flooding attacks without changing TCP implementations used by client hosts; only servers using them must be modified in order to produce and validate them, and their use should be as transparent as possible to clients. Therefore, the use of cookies must conform with all TCP mandatory rules, but may impose restrictions on the use of TCP optional behaviours. These restrictions, however, should be minimized in order to reduce the side effects of using cookies.

Cookies are 32-bit values stored as ISN values in the sequence field of SYN-ACK segments sent by servers, and are retrieved from the sequence numbers acknowledged in ACK segments sent by clients. Therefore, cookies should respect the recommendations for the generation of ISN values [3, 4].

Cookies also carry some TCP options negotiated exclusively on SYN segments. Currently they only carry a 3-bit encoding of 8 predefined MSS values. All other options are simply ignored[3], including window scaling, selective acknowledgment, and time stamping for Round-Trip Time Measurement (RTTM) or Protection Against Wrapped Sequence numbers (PAWS). Unfortunately, all these options were introduced to improve the performance of TCP connections [8, 9]. Thus, we can conclude that a SYN flooding attack, by triggering the use of cookies, has the potential side effect of reducing the performance of some server's TCP connections.

3.3. SYN cookies algorithms

The algorithm for generating cookies should try to reconcile two different goals. On one hand, cookies should be hard to guess by clients, in order to defeat attacks using ACK segments with forged, valid cookies. This implies that cookies should contain a large number of bits generated using servers secrets and functions not easily invertible by clients. On the other hand, cookies cannot be fully random and still respect the TCP rule of slowly growing over time. This implies that some part a

[3]SYN cookies cannot also handle correctly T/TCP connection requests. However, since this is still an experimental protocol, we will not address it further.

cookie should be generated without using values produced by cryptographic functions.

However, since cookies are only 32 bit long, it is difficult, if not impossible, to accomplish both goals simultaneously. Therefore, the algorithm to produce cookies cares only about security, and is completely independent of the algorithm to produce ordinary ISN values.

The algorithms currently used to generate and validate cookies are fully explained in Appendix A. In short, cookies are computed using constant secret values, TCP/IP addresses and ports of the client's SYN segment, a time counter and a 3-bit encoding of the server's MSS value (see Table A.1 in Appendix A). The validation of a cookie involves retrieving and testing the last two – time counter and MSS encoding – using the same secrets and the same fields of the client's ACK segment.

3.4. Risk analysis

Cookies were devised to solve a problem, and not to create new ones. Thus, they should not allow attackers to launch other kinds of attacks using them. This means that attackers should not be able to produce valid cookies, since that would allow them to create fictitious TCP connections on a victim server.

The reality is that valid cookies are relatively hard to forge[4]. On a given instant, a valid cookie for a given pair of TCP addresses can only take 32 values out of 2^{32} possible ones. The value of 32 comes out of multiplying the 4 acceptable values for the time counter with the 8 possible values of the MSS encoding (see Appendix A for more details). Any increment in the range of either one of these values would naturally improve the probability of guessing valid cookies.

4. OUR PROPOSAL

As previously mentioned, the generation of cookies is not a trivial task because one has to trade-off several different requirements. In this section we will show how the current SynCP can be improved in order to better deal with one of those requirements, namely the support of TCP options, without reducing its current functionality or its security against guessing attacks. Furthermore, we want to keep the basic approach of the current SynCP of not storing any state on servers, namely TCP options, for ongoing connection handshakes requested during a SYN flooding attack.

[4]Assuming that no better strategy exists for producing valid cookies besides random guessing.

The rationale for the new approach is the following: as TCP options cannot be fully embedded in cookies, for both practical and security reasons, then one has to force the client host to send again the TCP options together with the segment that carries the cookie. This means that the client must send another segment containing both the cookie and the SYN bit set, as only such segments may contain the TCP options that we are concerned with. This requirement can be met using the "*simultaneous connection initiation*" described in the seminal TCP documentation [3].

4.1. Basic approach

Figure 1 shows the diagram presented in [3] (and corrected in [10]) describing the steps followed in one simultaneous connection initiation. The new SynCP will explore this particular way of negotiation; in particular it will conduct the client socket through the same state transition of socket A of Figure 1.

	socket A state		segment		socket B state
1	CLOSED				CLOSED
2	SYN-SENT	→	(SEQ=x)(CTL=SYN)	...	
3	SYN-RCVD	←	(SEQ=y)(CTL=SYN)	←	SYN-SENT
4		...	(SEQ=x)(CTL=SYN)	→	SYN-RCVD
5		→	(SEQ=x)(ACK=$y+1$)(CTL=SYN,ACK)	...	
6	ESTABL.	←	(SEQ=y)(ACK=$x+1$)(CTL=SYN,ACK)	←	SYN-RCVD
7		...	(SEQ=x)(ACK=$y+1$)(CTL=SYN,ACK)	→	ESTABL.

Figure 1. Steps followed by TCP sockets in a simultaneous connection initiation.

The new SynCP works as follows (see Figure 2-II). When the server receives a SYN, it computes a normal SYN-ACK reply, gets a cookie for the server's ISN, and sends it as a pure SYN (with the ACK bit disabled). A genuine client socket for the requested connection is in SYN-SENT state, will move to SYN-RCVD and reply to the server's SYN with a SYN-ACK, repeating its ISN number and all the TCP options sent by the server. When the server receives a SYN-ACK for a socket in LISTEN state it checks if the acknowledged sequence number is a valid cookie. If it is valid, the server creates a new connection with the client, and sends back a SYN-ACK; otherwise, it sends back an RST.

This new way of using cookies is more complex (and thus slower) than the original one, but has the advantage of allowing both client and server to negotiate and agree on TCP options that are relevant for performance. Thus, the performance penalty imposed by this 4-way handshake can be blurred by the performance gain in the subsequent data transfer. A similar 4-way handshake was adopted by the Stream

	client socket state		segment		server socket state
1	CLOSED				LISTEN
2	SYN-SENT	→	(SEQ=s)(CTL=SYN)(tentative TCP options)	→	
3	ESTABL.	←	(SEQ=cookie)(ACK=s + 1)(CTL=SYN,ACK)	←	
4		→	(SEQ=s + 1)(ACK=cookie+1)(CTL=ACK)	→	ESTABL.

(I)

	client socket state		segment		server socket state
1	CLOSED				LISTEN
2	SYN-SENT	→	(SEQ=s)(CTL=SYN)(tentative TCP options)	→	
3	SYN-RCVD	←	(SEQ=cookie)(CTL=SYN)(final options)	←	
4		→	(SEQ=s)(ACK=cookie+1)(CTL=SYN,ACK)(final options)	→	ESTABL.
5	ESTABL.	←	(SEQ=cookie)(ACK=s + 1)(CTL=SYN,ACK)(final options)	←	

(II)

Figure 2. Steps followed by TCP sockets using (I) the current SynCP implementation and (II) the basic approach of the new SynCP.

Control Transmission Protocol (SCTP [11]) to tackle the same security problem.

The TCP options are initially presented in the SYN of the client (step 2), and the final set of agreed options is returned in the server's SYN reply containing the cookie (step 3). The client's SYN-ACK (step 4) will simply reproduce the options presented by the server, as they already result from an agreement process; the same happens in step 5.

4.2. Simplification of the basic approach

This basic approach can be further simplified: the final SYN-ACK sent by the server may be a simple ACK. Since client sockets are in a SYN-RCVD state, all they need to move to ESTABLISHED is an ACK. Consequently, we changed the protocol, replacing the SYN-ACK of step 5 by a simple ACK (see Figure 3).

	client socket state		segment		server socket state
1	CLOSED				LISTEN
2	SYN-SENT	→	(SEQ=s)(CTL=SYN)(tentative TCP options)	→	
3	SYN-RCVD	←	(SEQ=cookie)(CTL=SYN)(final options)	←	
4		→	(SEQ=s)(ACK=cookie+1)(CTL=SYN,ACK)(final options)	→	ESTABL.
5	ESTABL.	←	(SEQ=cookie+1)(ACK=s + 1)(CTL=ACK)	←	

Figure 3. New SynCP with a final ACK instead of a SYN-ACK.

Early experiences showed that this simplification is not only possible but also critical. In fact, some TCP implementations follow a simplified state diagram where a socket in the SYN-RCVD state only changes to ESTABLISHED after receiving an ACK (Figure 6 of [3]), though they accept the SYN-ACK as a valid segment. This is a clear violation of TCP rules (c.f. [10], §4.2.2.10).

4.3. Initial assessment of problems

This new way of using cookies is a sort of Pandora box, since the exploitation of simultaneous connection initiations is rare, although valid and imposed by the seminal paper defining the TCP standard. Therefore, this protocol was tested and evaluated with several client operating systems to better assess its suitability. We tried to use both old and new systems, and also Unix/Linux, Windows and other proprietary systems (e.g. Cisco IOS).

Table 1. Client operating systems used to test the new SynCP and the result of a preliminary evaluation of their support for simultaneous connection initiations.

Operating System	*OS or Kernel version*	*Supports the simultaneous connection initiation*
Windows	CE 3.0 (PocketPC) 95, 98 SE, Millennium NT 4 Workstation/Advanced Server 2000 Professional/Server XP Professional	No
Cisco IOS	C4500-I-M V 11.1(7) C7200-DS-M V 12.0(7)	No
SunOS HP-UX Linux FreeBSD OpenBSD MacOS Digital UNIX OSF1 SGI IRIX	4.1.3, 5.6, 5.7, 5.8 A.09.05 2.2.x, 2.4.x 3.3 2.8 9.2.2 V4.0 5.2	Yes

Table 1 shows the exact systems that we experimented with and the preliminary results of using the protocols of Figures 2-II and 3. These tests showed two facts concerning the simultaneous connection initiation forced by the server:

1 Some operating systems apparently support it, but after a certain point they fail.

2 Some client operating systems support it, but react differently to the segments received.

Windows and Cisco IOS systems exemplify the first kind of systems. All the Windows systems tested fail the same way. After accepting the SYN-ACK reply, the client socket changes from SYN-SENT to SYN-RCVD, but thereafter it stays stuck in that state (repeatedly sending SYN-ACK segments to the server until giving up, sending then an RST; see Figure 4). We tried several possible replies to make it change state, including RST, but without any success.

The two Cisco IOS systems also fail but differently from the Windows systems. The client socket changes to SYN-RECV after receiving the

SYN, but replies with a simple ACK, instead of a SYN-ACK. Thus, from our point of view, these systems fail in handling the simultaneous connection initiation because we need to get a SYN-ACK segment from clients.

```
C.1711 > S.ssh: S 3711264047:3711264047(0) win 64240 <mss 1460,sackOK> (DF)
S.ssh > C.1711: SP 416925441:416925441(0) win 5840 <mss 1460,sackOK> (DF)
S.ssh > C.1711: S 3878041854:3878041854(0) ack 3711264048 win 5840 <mss 1460> (DF)
C.1711 > S.ssh: S 3711264047:3711264047(0) ack 416925442 win 64240 <mss 1460,sackOK> (DF)
S.ssh > C.1711: . 1:1(0) ack 1 win 5840 (DF)
S.ssh > C.1711: P 1:26(25) ack 1 win 5840 (DF)
C.1711 > S.ssh: S 3711264047:3711264047(0) ack 416925442 win 64240 <mss 1460,sackOK> (DF)
S.ssh > C.1711: P 1:26(25) ack 1 win 5840 (DF)
C.1711 > S.ssh: S 3711264047:3711264047(0) ack 416925442 win 64240 <mss 1460,sackOK> (DF)
S.ssh > C.1711: P 1:26(25) ack 1 win 5840 (DF)
C.1711 > S.ssh: R 3711264048:3711264048(0) win 0
```

Figure 4. Output produced by the `tcpdump` tool showing the segments exchanged using **ssh** in a Windows XP system to connect to the modified server using always the new SynCP. The PUSH flag in the second segment is explained in §4.6.3) and, for clarity, all NOPs of TCP options were removed.

In the second kind of systems we can distinguish two different reactions:

- Some only change to ESTABLISHED after receiving a pure ACK; getting a SYN-ACK only make them repeat their own SYN-ACK (e.g. SunOS 4.1.3). This behaviour, already referred to in §4.2, goes against TCP rules.
- Some acknowledge the SYN-ACK sent by the server, if using the protocol of Figure 2-II (e.g. SunOS 5.8). This is a legal behaviour.

These different behaviours show that our new SynCP is more sensitive to differences in TCP implementations of client operating systems than the current one. Though it may predict and accommodate, as much as reasonable, some known problems of client systems, there is always a possibility of failing with some of them.

4.4. Overcoming problems raised by firewalls: mixed approach

Firewalls usually refuse TCP connections, initiated outside, to inside ports other than well-known service ports. This means that if the client of Figure 3 is behind a firewall, and the server is outside the defense perimeter of that firewall, the segment sent in step 3 will probably not reach the client. In that case, the client would continue to send SYN segments just like in step 2, until giving up. Therefore, the new SynCP will probably fail if the client socket is behind a firewall.

The solution that we devised for this problem is a best effort modification of the protocol presented in Figure 3. The modification consists of mixing both SynCPs, the current and the new, and thus the server replies to a SYN request with both a SYN and a SYN-ACK containing cookies. The format of the SYN-ACK is just like in the current SynCP, i.e. without any TCP options other than MSS. The server will try to deduce, from future segments sent by the client, which of the SynCPs it engaged to. Basically, if it receives a SYN-ACK, the client received the SYN and is using the new protocol; if it only receives an ACK, the client probably did not receive the SYN and is using the current protocol. If the segments with cookies sent by the server arrive in a different order to the client (the SYN-ACK first and the SYN next), the client will react to the SYN-ACK just like in the current SynCP (c.f. Figure 2-I). The delayed SYN will make the client reply with a harmless ACK (from Figure 10 of [3]).

Such mixing has a key issue, which is the compatibility between the SYN and SYN-ACK segments sent by the server. In practice this means that we have to decide if the cookies of these segments are the same or produced differently. Both solutions have advantages and drawbacks: equal cookies are natural to clients but may confuse the server; different cookies may be awkward to clients but allow the server to decide correctly. We chose the second approach, which is described below; for the sake of completeness, the problems raised by other approach are described in Appendix B.

4.4.1 Mixed SynCPs with two different cookies.

This approach simplifies the task of the server when dealing with probable segment losses because it knows exactly, from the acknowledged sequence numbers, which segments the client saw. The two cookies can be easily computed one from the other using a simple and fast invertible function, like a one's complement.

Its problem is that clients may react differently to the strange scenario of receiving two segments slightly incompatible between themselves. The main issue here is how should a client socket react when it receives a partially incorrect SYN-ACK, i.e. with an incorrect sequence number (server's ISN) and a correct acknowledged sequence number (client's ISN plus one).

According to [3], an RST should only be sent by a socket in any non-synchronized state (SYN-RCVD in this case) if "*the incoming segment acknowledges something not yet sent (the segment carries an unaccept-*

able ACK)". Since that is not the case, the normal reaction should be to either (i) ignore the segment, or (ii) send an ACK with the actual sequence numbers known by the client. In fact, our tests show that clients systems do react differently, but most of them send the expected ACK. Two systems, unfortunately, send RST segments: OpenBSD 2.8 and MacOS 9.2.2. This problem will be analysed further below.

So, the mixture of SynCPs using two different cookies – $\mathbf{cookie}_1$ and $\mathbf{cookie}_2$ (see Figure 5) – works this way:

I If the client receives the server's SYN, then its socket, after changing to SYN-RCVD, waits only for an ACK to change to ESTABLISHED. The SYN-ACK can be received in the meanwhile, but because its sequence number ($\mathbf{cookie}_2$) is different from the sequence number of the previously received SYN ($\mathbf{cookie}_1$), it is invalid and an ACK is sent back to the server.

II If the client misses the server's SYN because of a firewall, then it falls back to the current SynCP, as it only gets the server's SYN-ACK without any TCP options.

The server can easily check which of these alternative scenarios is the real one for each negotiation using cookies. If it gets a SYN-ACK with a $\mathbf{cookie}_1$, then the client saw the SYN and engaged in the new SynCP. If it gets a simple ACK with a cookie, two scenarios are possible:

- **ACK acknowledges $\mathbf{cookie}_1$:** the client saw the SYN and engaged in the new SynCP. The server simply drops the segment, as it should get a SYN-ACK with that cookie; the client will keep sending SYN-ACK segments until giving up or until getting a reply from the server.
- **ACK acknowledges $\mathbf{cookie}_2$:** the client did not see the SYN and engaged in the current SynCP.

This mixed protocol using two cookies fails in two systems – OpenBSD 2.8 and MacOS 9.2.2 – because these, after accepting the SYN with $\mathbf{cookie}_1$, do not reply with an ACK to the following SYN-ACK carrying $\mathbf{cookie}_2$. Instead, they reply with an RST, which terminates the connection just established on the server side (see Figure 6).

However, such RST segments have some unusual properties that can help the server to detect and, possibly, overcome the problem. OpenBSD sends an RST but keeps the connection in the same SYN-RCVD state, which is an illogical reaction: if the RST is meaningful, it will eventually terminate the connection just created, so there is no point in keeping it. Fortunately, the RST is unusual and can easily be spotted and ignored

	client socket state		segment		server socket state
1	CLOSED				LISTEN
2	SYN-SENT	→	(SEQ=x)(CTL=SYN)(tentative TCP options)	→	
3	SYN-RCVD	←	(SEQ=cookie$_1$)(CTL=SYN)(final options)	←	
4		→	(SEQ=x)(ACK=cookie$_1$ + 1)(CTL=SYN,ACK)(final options)	···	
5		←	(SEQ=cookie$_2$)(ACK=x + 1)(CTL=SYN,ACK)	←	
6		···	(SEQ=x)(ACK=cookie$_1$ + 1)(CTL=SYN,ACK)(final options)	→	ESTABL.
7	ESTABL.	←	(SEQ=cookie$_1$ + 1)(ACK=x + 1)(CTL=ACK)	←	

(I)

	client socket state		segment		server socket state
1	CLOSED				LISTEN
2	SYN-SENT	→	(SEQ=x)(CTL=SYN)(tentative TCP options)	→	
3		×	(SEQ=cookie$_1$)(CTL=SYN)(final options)	←	
4	ESTABL.	←	(SEQ=cookie$_2$)(ACK=x + 1)(CTL=SYN,ACK)	←	
5		→	(SEQ=x + 1)(ACK=cookie$_2$ + 1)(CTL=ACK)	→	ESTABL.

(II)

Figure 5. Steps followed by TCP sockets using an improved version of the new SynCP, capable of handling correctly clients behind a firewall. Scenario I shows the negotiation steps with a client not protected by a firewall; scenario II, on the contrary, shows the negotiation steps with a client protected by a firewall dropping pure out-in SYN segments. The values of **cookie$_1$** and **cookie$_2$** must be different and can be computed from each other using an invertible function.

I – OpenBSD 2.8

```
C.911 > S.ssh: S 1804448176:1804448176(0) win 16384 <mss 1460,sackOK,wscale 0,timestamp 12250862 0>
S.ssh > C.911: SP 3628984603:3628984603(0) win 5792 <mss 1460,sackOK,timestamp 1656630 12250862,wscale 0> (DF)
S.ssh > C.911: S 665982692:665982692(0) ack 1804448177 win 5840 <mss 1460> (DF)
C.911 > S.ssh: S 1804448176:1804448176(0) ack 3628984604 win 17376 <mss 1460,sackOK,wscale 0,timestamp 12250862 1656630>
S.ssh > C.911: . 1:1(0) ack 1 win 5840 <timestamp 1656637 12250862> (DF)
S.ssh > C.911: P 1:26(25) ack 1 win 5840 <timestamp 1656638 12250862> (DF)
C.911 > S.ssh: R 1804448177:1804448177(0) win 17376
C.911 > S.ssh: . 1:1(0) ack 26 win 17352 <timestamp 12250863 1656638>
S.ssh > C.911: R 3628984629:3628984629(0) win 0 (DF)
C.911 > S.ssh: P 1:23(22) ack 26 win 17376 <timestamp 12250863 1656638>
S.ssh > C.911: R 3628984629:3628984629(0) win 0 (DF)
```

II – MacOS 9.2.2

```
C.62884 > S.ssh: S 3091633285:3091633285(0) win 32768 <mss 1460,wscale 0> (DF)
S.ssh > C.62884: SP 2343418210:2343418210(0) win 5840 <mss 1460,wscale 0> (DF)
S.ssh > C.62884: S 1951549085:1951549085(0) ack 3091633286 win 5840 <mss 1460> (DF)
C.62884 > S.ssh: S 3091633285:3091633285(0) ack 2343418211 win 32768 <mss 1460,wscale 0> (DF)
C.62884 > S.ssh: R 3091633286:3091633292(6) win 0 (DF)
146.193.7.2.ssh > C.62884: . 1:1(0) ack 1 win 5840 (DF)
C.62884 > S.ssh: R 3091633286:3091633304(18) win 0 (DF)
```

Figure 6. Output produced by the `tcpdump` tool showing the segments exchanged using `ssh` in a OpenBSD 2.8 or a MacOS 9.2.2 systems to connect to the modified server always using the new SynCP with different cookies. The data in the RST segments sent by the MacOS consists of the following textual messages: "`TH_SYN`" and "`No TCP/No listener`". The PUSH flag in the second segment of each dump is explained in §4.6.3 and, for clarity sake, all NOPs of TCP options were removed.

by the server, enabling the mixed SynCP to be used with OpenBSD clients: from the dump of Figure 6-I we see that the RST includes a non-null window size, although this system usually sends RST segments like any other, i.e. with a null window size.

MacOS sends an RST and terminates immediately its ongoing connection. This RST is also unusual, because it includes data (see Figure 6-II), but that can only help the server to identify a client that does not support the mixed SynCP with different cookies.

4.5. Final protocol: cache of problematic clients

Its now time to summarize all the problems faced by a mixed SynCP using different cookies in order to present a common solution for all of them. In short, the major problems are the following three:

- Some systems do not support the simultaneous connection initiation (e.g. Windows systems);
- Some systems do not react as required to the SYN sent by the server (e.g. Cisco IOS); and
- Some systems do not react well to a mixed protocol using different cookies (e.g. MacOS).

To handle all these cases we need to (i) maintain in the server a cache with the IP of problematic clients and to (ii) use only the current SynCP with hosts referred in that cache. Such cache should be updated whenever the server suspects a problem with the client. Furthermore, the cache should be managed in a conservative way, i.e. always assuming the worst case. This is advised because client systems may belong to private networks, using a gateway and masquerading to access Internet servers. In such cases, the server always sees the IP of gateways, but the protocol is sensitive to particular TCP implementations of client hosts behind them. Therefore, we should never remove hosts from the cache once they get there for some reason (except for getting a free entry).

The hints for inserting a client's IP in the cache are the following:

- The server receives a SYN-ACK, with a cookie, to a socket in the ESTABLISHED state. In this case we are probably dealing with a Windows client: we put its IP in the cache, but we don't abort the connection (first, because we may be wrong; and, second, because that is useless, as explained in §4.3); instead, the segment is processed normally by the TCP.

- The server receives an ACK, with a **cookie**$_1$, to a socket in the LISTEN state. In this case we may be dealing with a Cisco IOS client: we put its IP in the cache and we drop the packet.
- The server receives an RST, with the "TH_SYN" message, to a socket in the ESTABLISHED state. In this case we are probably dealing with a MacOS client: we put its IP in the cache and we let the RST be processed normally.

Note that in the first two cases the hint may be a false positive caused by: (i) a delayed reception of the server's ACK, in the first case, or (ii) a delayed client's SYN-ACK, in the second case. But, as previously explained, we should always assume the worst case; therefore we assume that such segments reveal a problematic client.

This cache is different from the one used by the Reset Cookies protocol to store security associations (c.f. §2). Both store the IP of real systems that tried to access the server, but our cache stores only the IP of problematic clients, while the other stores all the IPs. Thus, we are likely to get a better hit-rate with a cache of equal length. Furthermore, we only delay connections initiated by problematic hosts, while Reset Cookies delays the connections of all hosts not in the security association's cache.

The use of a cache of problematic clients is not a perfect solution, because the server reacts when it believes there could be a problem, instead of anticipating the problem. One possibility for an earlier detection of problematic clients could be to apply fingerprinting techniques, such as the ones used in active recognition tools (e.g. **nmap** [12]) or in passive IDS systems [13, 14], to the contents of SYN segments (either at TCP or IP level). This approach is not 100% accurate, may work better for some operating systems and may even be disturbed by fingerprint scrubbers [15]. Nevertheless, it may be explored in the future for some particular cases without interfering with the cache update policy previously described.

4.6. Security evaluation

4.6.1 Guessing SYN-ACK segments with valid cookies.

The mixed SynCP is as secure as the current one. Cookies are generated and validated the same way; they only appear in different TCP segments – in ACK segments in the current implementation and in ACK and SYN-ACK segments in the new one. The cookie of the SYN is computed from the one of the SYN-ACK using an simple and fast invertible function, like the one's complement. The fact of using two cookies instead of one does not reduce the resistance against guessing attacks, because at a given instant the set of cookies that is valid for a given type of segment, ACK or SYN-ACK, remains equal to that of the current SynCP.

4.6.2 Forged SYN segments with spoofed source addresses.

Another relevant concern with security is the impact of SYN segments sent by servers when replying to forged SYN segments sent by attackers. Unlike the current SynCP, that uses a normal reply, a SYN-ACK, the new SynCP uses a typical request segment (a SYN) as a reply to a client. This means that an attacker can lead a server under a SYN flooding attack to initiate connections with other servers. However, the algorithms to generate and validate cookies are enough to detect and avoid such problem.

Imagine the following scenario, illustrated in Figure 7: an attacker sends a forged SYN to a server A, which is using the new SynCP, and the forged segment says that the sender is an existing server B. The result of such attack is that A and B will exchange some segments and abort the connection, because the SYN-ACK from B has a cookie that was generated with x as ISN, and not with the ISN y provided by B in step 4. Furthermore, server B will also abort the connection by replying with an RST to any SYN-ACK segments sent by A to a socket in the LISTEN state (as in the current SynCP); such RST is produced by the normal operation of the TCP.

	server A socket state	segment	server B socket state
1	LISTEN		LISTEN
2		← (SEQ= x)(CTL=SYN) (apparently from B)	
3		→ (SEQ=$cookie(x)$)(CTL=SYN) →	SYN-RCVD
4		← (SEQ= y)(ACK=$cookie(x)$ + 1)(CTL=SYN,ACK) ←	
5		→ (RST) →	CLOSE

Figure 7. Segments exchanged resulting from a forged SYN referring an existing server B as the sender. The SYN-ACK segments with cookies that are also sent by A are not shown for the sake of simplicity, but they also abort the connection, because B replies with an RST to a SYN-ACK sent to a socket in LISTEN state (as in the current SynCP).

Note that the issue here is to avoid the creation of a useless TCP connection between A and B (between two sockets in LISTEN state) from a spoofed SYN segment sent by a attacker. Without using host authentication we cannot protect B from getting replies from A caused by spoofed segments. Neither can B prove that those segments were in fact sent by A.

4.6.3 Identification of SYN segments with cookies.

The diagram of Figure 7 is not valid if both hosts A and B are servers acting similarly, i.e. responding to SYN segments with other SYN segments carrying a cookie. In such a scenario, both hosts enter into an endless ping-pong of SYN segments, since they do not (intentionally) keep any record about past replies containing cookies.

This problem can be solved only if SYN segments containing cookies could be clearly distinguished from other SYN segments with ordinary ISN numbers. Two possible solutions for this problem are:

- to use one of the flags in the base TCP header not used in SYN segments (URG, PUSH, etc.); or

- to use a new TCP option.

The first solution is a sort of a hack that may work in most cases since TCP implementations are not sensitive to the state of such header bits in SYN segments. The second solution is more standard, all TCP implementations should be immune to it (see [10]) but it implies the reservation of a new option value.

Note that the clear identification of SYN segments is only needed for servers using the new SynCP, and not by any other hosts. Furthermore, such identification helps modified server hosts to further reduce the problem presented in Figure 7. In fact, as the host of server B can see that the segment from A is a SYN segment with a cookie, it may simply drop the segment and thus prevent all the following exchange of segments.

5. IMPLEMENTATION

The new SynCP, described in §4.4, Figure 5-I, was implemented on a Linux kernel (2.4.2-2). The implementation involved a minor modification of the TCP modules: three files (tcp_ipv4.c, tcp_input.c and syncookies.c) and about 300 new lines of code.

The implementation uses the following strategy for choosing SynCPs: if the client does not require any TCP options, or if the client belongs to our cache of problematic clients, the current SynCP is used; otherwise, we use the new mixed SynCP, described in Figure 5. To simplify the protocol tests, the kernel was also modified to behave as if under a SYN flooding attack.

The SYN segments used by the new SynCP are identified with the PUSH TCP header flag, as explained in §4.6.3. This flag was used in all the tests of the new protocol without any noticeable problems, but it should be replaced in the future by a proper, standard TCP option.

6. CONCLUSIONS

In this document we presented a new strategy for using SYN cookies by a server under a SYN flooding attack. This new strategy overcomes a limitation of the current SynCP – it does not allow clients to negotiate any TCP options within SYN segments (it only allows clients to get the server's MSS). The solution that we propose relies on the fact that TCP allows a scenario called "*simultaneous connection initiation*", that we use to force client hosts to repeat their SYN requests. This way, the server can get together, in a single SYN-ACK, a cookie and all the TCP options initially requested by the client and already agreed to by the server.

This simple approach, fully compatible with standard TCP rules, faces two major problems. First, some systems do not deal correctly with the simultaneous connection initiation (e.g. Windows systems). Second, client-side firewalls may transparently interfere with the connection initiation started by the server, thus preventing the client from connecting to the server. To overcome these problems we did two complementary actions: (i) changed the protocol, in order to simultaneously use the current and the new SynCPs, creating a mixed SynCP, and (ii) added to the server TCP implementation a cache for storing the IP of problematic client hosts. This cache is updated whenever the server gets a hint, from the TCP segments received, that the client may not deal properly with the new mixed SynCP.

Concerning the security of the new protocol, we did not change the algorithms for generating and validating cookies, so they are as secure as they were before. We also showed that, due to the current algorithm to validate cookies, spoofed connection requests cannot drive a server to establish a connection with another victim server. Finally, we justified why SYN segments sent by the server must be properly identified to detect equal reactions of two hosts trying to connect with each other, both being under a SYN flooding attack. For simplicity we used the PUSH flag of the TCP header for such identification, without any noticeable problems, but a more correct implementation should use a proper, standard TCP option.

The new SynCP was implemented in a Linux kernel and tested with a large set of client operating systems. From the tests, we concluded that some systems do not tolerate it (Windows, Cisco IOS and MacOS), that some systems react strangely but in a way that can be detected and masqueraded by the server (OpenBSD), and that all the other systems behave as expected. The problems raised by the first kind of systems are solved with the cache of problematic systems.

In conclusion, we believe this new mixed SynCP, using both the current one and a new one faking a simultaneous connection initiation, is a valid and powerful improvement of the current SynCP. The resulting protocol supports the negotiation of any TCP options, is flexible enough to deal with firewalls and can be downgraded, on an as-needed basis, to the current one in order to attend to special problematic clients. In future implementations the late discovery of such clients may be partially anticipated by applying fingerprinting techniques to SYN segments.

Acknowledgments

I would like to thank Paulo Ferreira for reviewing draft versions of this paper and David Matos for reviewing the final version.

Appendix A: SYN cookies algorithms

Linux kernels use the folowing algorithms to generate and validate cookies:

$$\begin{aligned}
H_1 &= hash_{32-61}\,(S_{addr}|S_{port}|D_{addr}|D_{port}|K_1) \\
H_2 &= hash_{32-61}\,(S_{addr}|S_{port}|D_{addr}|D_{port}|counter|K_2)
\end{aligned}$$

Generation:

$$cookie = H_1 + ISN_{client} + (counter \times 2^{24}) + (H_2 + data) \bmod 2^{24}$$

Validation:

$$\begin{aligned}
counter_{cookie} &= (cookie - H_1 - ISN_{client}) \div 2^{24} \\
\Delta counter &= counter_{current} - counter_{cookie} \\
data &= (cookie - H_1 - ISN_{client}) \bmod 2^{24} - H_2 \bmod 2^{24}
\end{aligned}$$

$hash_n\,(x)$	n bit range, starting from *lsb*0, produced from x using the compression function of a digest algorithm (MD5 or SHA-1)
S_{addr}, S_{port}	source TCP/IP address
D_{addr}, D_{port}	destination TCP/IP address
K_1, K_2	secret keys
ISN_{client}	ISN provided by the client in the SYN segment
counter	minute counter
data	24-bit value

Cookies are generated and validated using two constant secret values, K_1 and K_2, which are long enough to completely fill the input buffer of the hash function used (64 bytes for both MD5 and SHA-1, so K_1 has 52 bytes and K_2 has 48 bytes). The data value is a server-defined value currently used for storing a 3-bit encoding of 8 predefined MSS values, presented in Table A.1.

The secrets K_1 and K_2 are produced using the kernel random number generator, the same used to generate the random part of ordinary IPv4 ISN values. These values are produced the first time SYN cookies are used after a system reboot and remain constant in kernel memory. The difficulty of guessing K_1 and K_2 from cookies is out of the scope of this document.

The kernel checks for suitable cookies only within a short time frame, starting when the last one was sent and ending a few seconds later. During that time gap cookies

Table A.1. MSS predefined values encoded in the data value of SYN cookies.

SYN Cookie data	0	1	2	3	4	5	6	7
MSS value	64	256	512	536	1024	1440	1460	4312

may be checked, and are accepted only after some integrity control validations. There are two integrity controls of cookie contents, and if any of them fails the cookie is rejected. The first integrity control test checks if it is valid (acceptable) in terms of a temporal criteria: if $\Delta counter$ is lower than a given threshold (currently a hard-coded value of 4 minutes), it is acceptable. The second integrity control test checks if the data value is a valid one, i.e. a value between 0 and 7.

Appendix B: Problems raised by mixing SynCPs with two equal cookies

The main advantage of using equal cookies in both SYN and SYN-ACK segments used in a mixed SynCP is that clients always see segments that do not look strange. The instant chosen to send the server's SYN-ACK reply is irrelevant for the correction of the protocol from the client's point of view.

However, the premature sending of the server's SYN-ACK reply may be problematic for the server since it is not keeping state about ongoing connections. There are two particular scenarios that could lead to problems:

- The client socket receives the SYN and the SYN-ACK segments, sends replies whenever it decides to and moves to ESTABLISHED. If all the client's replies get lost, the client stays with a TCP connection that will be destroyed as soon as it sends some data or probes the server.
- The client socket receives the SYN and the SYN-ACK segments, sends replies whenever it decides to and moves to ESTABLISHED. If the server misses the SYN-ACK reply, but it sees one ACK reply, it will conclude that the client did not receive the SYN and, therefore, it uses the current SynCP. The result is that the server will establish a connection with the client, but will assume that the client will not use any TCP options, which is not true. This scenario can occur with client sockets that acknowledge SYN-ACK segments in the SYN-RCVD state, like SunOS 5.8.

The first scenario is annoying but not dramatic, being similar to a temporary server failure. The second scenario is more critical, since it can lead to future problems during the client-server interaction. However, it may be detected and avoided in some cases, namely when both client and server could agree on using TCP timestamps. In this case, the server could activate the time stamping in its SYN, but not in the SYN-ACK, and latter detect only from ACK segments if the client saw its SYN (if they carry a timestamp). This way ACK segments with both a cookie and a timestamp could not be used to create a connection.

Concerning the use of the TCP timestamp mechanism, the document describing it [9] says nothing about a segment not carrying a timestamp when the receiver is expecting it; it only says that timestamps may be sent only when the sender got one in the initial SYN of the connection. Therefore, we assume that it is legal to receive a SYN with a timestamp and a SYN-ACK without timestamp. Furthermore, such lack of timestamp in the SYN-ACK should not also affect the PAWS mechanism, also described in [9], because apparently it is only used for "*open connections*".

References

[1] D. J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`.

[2] Syn cookies mailing list `syncookies-archive@koobera.math.uic.edu`. `http://cr.yp.to/syncookies/archive`.

[3] J. Postel. Transmission Control Protocol. RFC 793, September 1981. available via DDN Network Center.

[4] S. Bellovin. Defending Against Sequence Number Attacks. RFC 1948, May 1996. available via DDN Network Center.

[5] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2267, January 1998. available via DDN Network Center.

[6] Livio Ricciulli, Patrick Lincoln, and Pankaj Kakkar. TCP SYN Flooding Defense. In *Comm. Net. and Dist. Systems Modeling and Simulation Conf. (CNDS' 99), 1999 Western MultiConf. (WMC' 99),*, San Francisco, CAL, USA, January 1999.

[7] Eric Schenk. Another new thought on TCP SYN attacks, 1996. `http://www.wcug.wwu.edu/lists/netdev/199609/msg00115.html`.

[8] V. Jacobson and R. Braden. TCP Extensions for Long-Delay Paths. RFC 1072, October 1988. available via DDN Network Center.

[9] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, May 1992. available via DDN Network Center.

[10] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, October 1989. available via DDN Network Center.

[11] Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, October 2000. available via DDN Network Center.

[12] Fyodor. Remote OS detection via TCP/IP Stack FingerPrinting, October 1998. `http://www.insecure.org/nmap/nmap-fingerprinting-article.html`.

[13] Burak Dayıoğlu and Attila Özgit. Use of Passive Network Mapping to Enhance Signature Quality of Misuse Network Intrusion Detection Systems. In *16th Int. Symp. on Computer and Information Sciences*, November 2001.

[14] Honeynet Project. Know Your Enemy: Passive Fingerprinting. White Paper, January 2002. `http://project.honeynet.org`.

[15] Matthew Smart, G. Robert Malan, and Farnam Jahanian. Defeating TCP/IP Stack Fingerprinting. In *Proc. of the 9th USENIX Security Symp.*, 2000.