Macrothink Institute™

Network Protocols and Algorithms
ISSN 1943-3581
2010, Vol. 2, No. 1

The TCP Split Handshake: Practical Effects on Modern Network Equipment

Tod Beardsley (Corresponding author)

BreakingPoint Systems

3900 North Capital of Texas Highway, Austin, Texas 78746, United States

Tel: 1-512-821-6000 E-mail: todb@breakingpoint.com

Jin Qian

BreakingPoint Systems

3900 North Capital of Texas Highway, Austin, Texas 78746, United States

Tel: 1-512-821-6000 E-mail: jqian@breakingpoint.com

Abstract

Many network engineers might presume that the TCP three way handshake is the one, inviolate method of establishing TCP connections. A smaller percentage of engineers are also familiar with the little-used "simultaneous-open" connection method of establishing TCP connections. Researchers have discovered a third means to initiate TCP sessions, dubbed the "split-handshake" method, which blends features of both the three way handshake and the simultaneous-open connection. Popular TCP/IP networking stacks respect this novel handshaking method, including Microsoft, Apple, and Linux stacks, with no modification. Given the novelty of the split-handshake technique, session aware devices have had very little formal testing to determine their effectiveness in relation to sessions established in this way. The authors audit a number of intrusion detection devices, NAT gateways, port scanners, and firewalls, and unexpected behavior was observed within each class of device and application. This inconsistent behavior leads to the conclusion that such network-aware devices and applications should undergo more rigorous testing by their respective manufacturers in an effort to reliably detect malicious traffic, handle network address translation more effectively, and detect the presence of servers offering this form of session establishment.

Keywords: IDS, networking, security, simultaneous-open, split-handshake, TCP, TCP/IP, TCP evasions, TCP handshake, three way handshake

1. TCP, the Transport Control Protocol

The Transport Control Protocol, or TCP, is the essential underpinning of the vast majority of Internet data traffic on the Internet. In 2007, TCP accounted for over 90% of all IPv4 data by volume, with IPv4 itself accounting for over 99% of Internet traffic [1]. In 2009, the volume of UDP traffic did increase, but TCP still accounts for about 89 to 97% of the total traffic [2]. TCP is clearly a ubiquitous and well-understood technology for transmitting a bewildering array of data types. E-mail, World Wide Web pages (both standard HTTP and secure HTTPS), most file transfers, and most Instant Messaging applications rely on TCP for transporting data between clients over the Internet.

The reason for this widespread adoption of TCP is due to TCP's built-in controls for session establishment and tear down, where both hosts begin (and typically, end) connections with a proscribed sequence of packets, which in turn allows for session management including proper sequencing of data, congestion control, and other session quality-assuring tasks. By way of contrast, other common transport protocols such as the User Datagram Protocol [3] and the Internet Control Message Protocol [4] have no such native understanding of session setup and management.

2. The TCP Handshake

In order to facilitate the features of a connection-oriented protocol, all TCP connections begin with a "handshake," as described in RFC 793 [5]. Because there are three well-understood steps for this handshake, countless sources refer this as the "TCP Three Way Handshake." For example, a US-CERT advisory from 1996 includes a representative diagram of the classical understanding of TCP session establishment [6], noted here as Fig. 1.

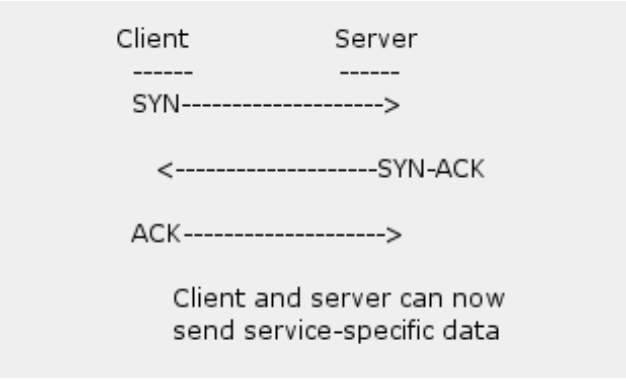

Figure 1. The three way handshake as described by US-CERT.

TCP sessions typically begin with one host (normally referred to as the client) sending a synchronization packet, or SYN, to another host (the server). The SYN contains an initial sequence number (ISN), a pseudo-random number which represents the beginning of the TCP session from the perspective of the client.

Next, the server acknowledges the client's SYN, and generates its own SYN. This

"SYN/ACK" packet contains both the server's ISN, as well as an acknowledgment number equal to the client's ISN plus 1.

Finally, the client acknowledges the server's SYN/ACK, and sends a packet with its own ISN incremented by one, as well as its acknowledgement number equal to the server's ISN plus 1.

Assuming all packets are sent and received without interruption, the typical TCP session setup is diagrammed in Fig. 2.

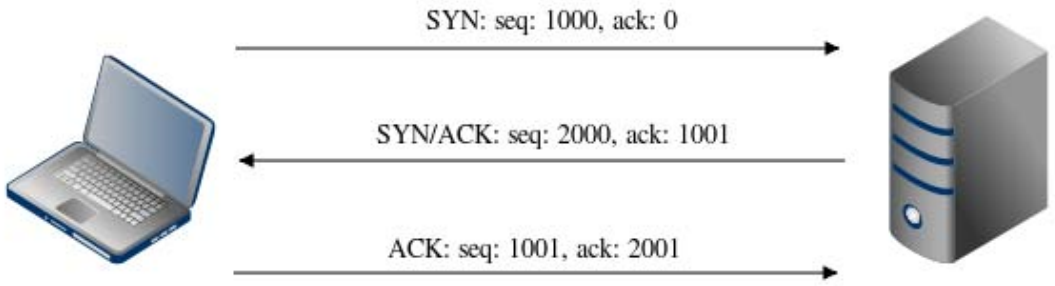

Figure 2. The TCP three way handshake.

3. Introducing the TCP Split-Handshake

The TCP three way handshake, described thus far, should be familiar to most experienced network engineers. However, section 3.3 of RFC 793 [5] which introduces this "three way (or three message) handshake" includes an intriguing figure of a four-step process, reproduced here as Fig. 3:

```
1) A --> B  SYN my sequence number is X
2) A <-- B  ACK your sequence number is X
3) A <-- B  SYN my sequence number is Y
4) A --> B  ACK your sequence number is Y
```

Figure 3. The four-step process as described by RFC 793

Because TCP implementations are permitted to combine steps two and three, most (and presumably, all) do, which is where the term "three way" originates. However, this is clearly a four step process. By splitting the second packet (the first packet from the server), into separate acknowledgement (ACK) and synchronization (SYN) packets, one might expect an RFC 793-compliant client to silently accept packet two, explicitly ACK packet 3, thus completing the handshake more-or-less normally. However, this is not the case. This was a surprising finding, and carries with it some fairly serious implications for network security

devices and applications which depend on the common model of TCP session establishment.

A proof-of-concept TCP listening agent was designed to test various client operating systems to determine how they might react to a split-handshake. This Ruby script, "fakestack.rb," is detailed in Appendix 1. Clients tested were fresh installations of Microsoft Windows XP, Service Pack 3; Apple Mac OS X v10.6 "Snow Leopard"; and Ubuntu 9.04 "Jaunty Jackelope" Linux.

In their default configurations, all three clients reacted to this proof-of-concept TCP server in the same manner, which appears inconsistent with the behavior described by the TCP three way handshake (described in previous section) and the TCP simultaneous-open connection (detailed in the next section). The procedure by which these sessions are established can be referred to as the "split-handshake," and is described in detail below.

3.1 Description of the split-handshake

First, a client sends a SYN packet (or "SYNs") the proof-of-concept server, as normal, picking a pseudo-random ISN. As discussed in Section 2, this is the normal means by which clients begin sessions with servers. Next, the server responds with a packet with the acknowledgement number incremented, and a random sequence number, but without the SYN flag set; only the ACK flag is set on the initial response from the server. The client silently accepts this packet, as is expected[1]. In step three, the server sends a SYN packet with a newly generated sequence number and the correct acknowledgement number.

Instead of merely ACKing this packet and entering the ESTABLISHED state, step four sees the client respond with a SYN/ACK packet, reusing its original sequence number (equal to the ISN), and setting its acknowledgement number to the server's ISN+1. The server then completes the handshake in step five by ACKing the client's SYN/ACK. It is in this way the TCP handshake was expanded to a "five way" sequence of packets, illustrated in Fig. 4.

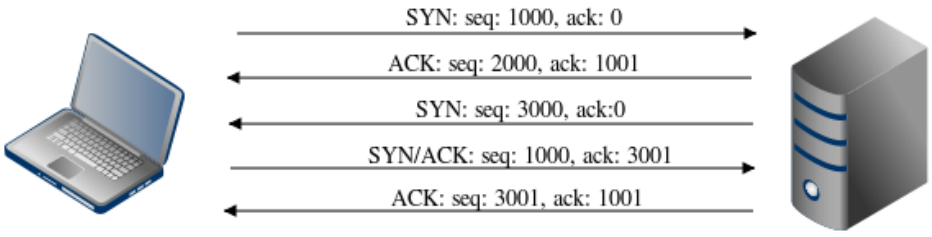

Figure 4. The Five Step TCP Split-Handshake

This handshake differs rather radically from the proscribed behavior in RFC 793, Section 3.4, and the effects of this are potentially profound in modern networks. The server never sends a SYN/ACK packet, but instead, that responsibility is assumed by the client in the tested configurations. From the point of view of any device which relies on the directionality of a literal SYN/ACK packet -- a packet with both the SYN and ACK bits set -- the roles of

[1] TCP hosts correctly should not ACK ACKs, as this would quickly consume all processing and networking resources.

client and server will appear reversed, with the client's ephemeral port behaving as a server receptor, and the server's well-known port behaving as a client receptor.

Further investigation reveals that step two (the server's initial ACK), appears to have no effect on establishing a new TCP session, and may optionally (or even randomly) dropped. Due to this behavior, this five-way handshake may itself be compressed to a four packet exchange. Fig. 5 illustrates this exchange, complete with example IP addresses and TCP ports. Note, the code listed in Appendix 1 may be used in either four way or five way mode, as the ACK packet may be commented out of the TCP flow.

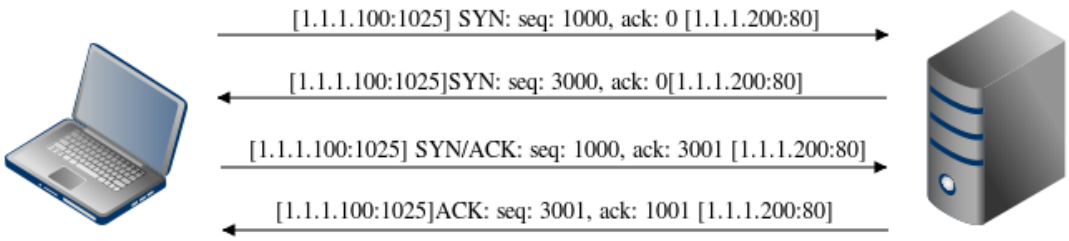

Figure 5. The Four Step Split Handshake

4. TCP Simultaneous-Open

The TCP three way handshake is not the only means by which TCP sessions may be established. RFC 793, Section 3.4, briefly touches on the notion of a pair of TCP hosts which simultaneously attempt to open a connection to each other via a SYN packet [5]. The "simultaneous-open" connection is more completely described by Stevens in TCP/IP Illustrated, Volume 1 [7]. Stevens imagines a scenario where both endpoints of the connection simultaneously (or nearly so) send each other SYN packets. This can be illustrated as in Fig. 6.

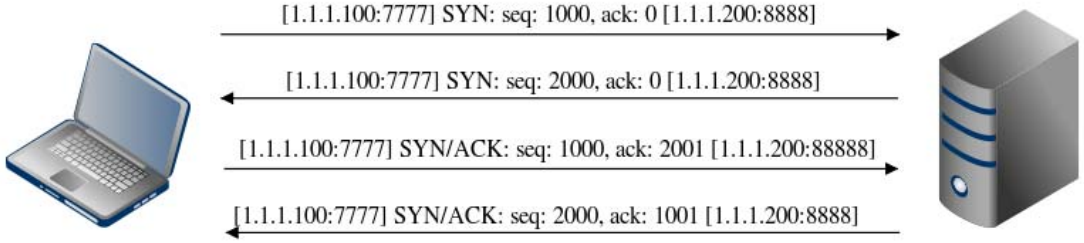

Figure 6. The simultaneous-open connection.

In practice, this is rarely, if ever, seen on the Internet today. This is likely due to the warning accompanying the Stevens example, "Many Berkeley-derived implementations do not support the simultaneous open correctly [...] The transition from the SYN_SENT state to

the SYN_RCVD state in Figure 18.12 is not always tested in many implementations" [7].

In addition, applications require a fairly contrived set of circumstances to initiate a simultaneous-open connection. Both hosts need to send SYN packets at virtually the same time, but more importantly, the port pair would have to be mirrored between hosts. In other words, the simultaneous-open service would have to be running on Host A's port 8888 and Host B's 7777 (using the Stevens example ports), and in addition, each host would need to be using the other's well-known port as its own ephemeral ports.

5. TCP Split Handshake Compared to Simultaneous-Open

The proof-of-concept implementation of the split-handshake differs primarily from the simultaneous-open connection because it does not require any simultaneity among SYN packets, nor does it require knowledge of the client's port prior to receiving the client's SYN.

Rather, in this split-handshake scenario, the server is passively listening for connections (representing the typical LISTENING state). It receives a SYN packet, and only then sends its SYN. The client moves from SYN_SENT to SYN_RECV, and sends a SYN/ACK. This behavior is no longer standard for the three way handshake, but is in accordance with the simultaneous-open behavior described by Stevens. However, instead of a SYN/ACK from the server, the client receives a mere ACK, and the session is established. This implies that the client has completed the usual three way handshake sequence, abandoning the "accidental" simultaneous-open connection sequence.

In effect, the split-handshake merges features of both the standard three way handshake and simultaneous-open. Moving from one style of session establishment to another, and back again, is certainly not envisioned by RFC 793. It is precisely this fluid behavior which can cause unexpected behavior in some modern session aware network equipment and applications.

6. Split Handshakes and Network Security Inspection Devices

6.1 Devices Under Test

In order to test the effectiveness of third party inspection in the face of the split-handshake, two dedicated security devices, the TippingPoint 2400 [8] and the Juniper SRX 5800 [9], and one software-based inspection device, Sourcefire Snort [10], were subjected to controlled testing. The TippingPoint device is a dedicated Intrusion Prevention System (IPS) appliance, which is designed to inspect communications between hosts for malicious network traffic. Once detected, the offending session is blocked from reaching the target. The Juniper device is a modular chassis-based system, equipped with an IPS component similar in function to the TippingPoint device, designed to detect and block network attacks, in addition to traditional stateful firewall and network address translation (NAT) functionality. The Snort system is a general purpose Ubuntu 9.04 Linux desktop

computer (described in Section 3), running the standard Snort intrusion detection system (IDS) service. The precise detection and blocking technologies enabling the TippingPoint and Juniper devices are proprietary, while the Snort engine is an open source C application available as an installable package from the "Universe" Ubuntu repository.

6.2 Test Design, Equipment, and Methodology

As the TippingPoint and Juniper systems are both dedicated, high-performance malicious traffic blocking systems, they are easily testable using the BreakingPoint Elite, a high-performance chassis-based network device testing system. Among other capabilities, the BreakingPoint Elite can deliver an array of attacks to test the effectiveness of network IPSes, and allows for several varieties of evasion techniques. The authors implemented the "SneakAck Handshake" evasion technique as a new feature of the BreakingPoint testing system [11], which allows test engineers to alter the BreakingPoint Security Component's TCP stack to allow TCP session establishment to behave as a split-handshake-enabled server, as described in Section 3. The BreakingPoint device routes traffic through the devices under test and acts as both the client and server; thus, as attacks are blocked and permitted, the results are automatically detected by the BreakingPoint device and a summary report is generated.

In contrast to the two IPS devices, the Snort device was not subjected to the BreakingPoint test equipment, for three reasons. Firstly, the Snort device is itself both the detection engine and the endpoint, which complicates BreakingPoint testing. Secondly, the Snort device's role is not traditionally one of a blocking device, but a detection device, so there is no purpose in relying on the BreakingPoint equipment to log attack success and failures. Finally, the Snort rule set is easily customizable, so a custom rule can be used to detect the payload of the fakestack.rb script, which enables a precise testing methodology using only a single "attack."

For the two IPSes, BreakingPoint representative test attacks were chosen which meet the following criteria:

a) Attacks are server-initiated. While many network attacks flow from a client to a server, there is a distinct class of "client-side" attacks [12], by which a malicious server attacks clients which connect to it. For example, a malicious web server might attack a browser application via an ActiveX control buffer overflow attack [13]. In order for an attacker to make effective use of the split-handshake, the attacking device should be the server, rather than the client, since the client cannot decide to use the three way handshake, a simultaneous-open connection, or the split-handshake. In the case of the Snort device, the "attack" is merely the HTTP payload of the fakestack.rb script as mentioned above.

b) Attacks are detected using the traditional TCP three way handshake. While the effectiveness of security devices in detecting client-side attacks regardless of evasion techniques is the subject of great interest to the authors, it is not the focus of this test. Thus, client-side attacks were chosen which are known to already be detectable by the devices under test. In the case of the Snort device, the custom Snort rule is already known to fire on

an identical payload delivered by a normal web server.

In addition, to better control the results of the test, attacks which meet the above criteria are also initiated without establishing a session at all. In other words, the attack is played out of the context of a session, using only ACK packets. Normally, such attacks will never succeed, for obvious reasons. Using this procedure as a control can indicate if the device under test is aware of sessions at all, or is merely detecting malicious traffic based purely on the data carried by individual packets regardless of their effectiveness in compromising clients.

6.3 Test Results

Among the three inspection technologies tested, three distinct result sets were observed.

6.3.1 Sourcefire Snort

The Snort device performed as expected in all tested circumstances. The test payload was detected both in the normal three way handshake scenario as well as the split-handshake scenario. The stream5 preprocessor component of the Snort device was used, as documented in Chapter 2 of the SNORT User's Guide [10]. The Stream5 preprocessor is responsible for session tracking and reassembly, indicating that the Snort device is capable of detecting client-side attacks equally well under both conditions. In addition, the Snort device did not detect the attack when replayed via a packet capture similar to that of Appendix 2, but with the initial three way handshake suppressed. This indicates that Snort, in a customary configuration, is both stateful in terms of TCP sessions, and does not flag un-sequenced TCP traffic as malicious events.

6.3.2 TippingPoint 2400

Of the TCP-based client-side attacks which are known to be detectable by the TippingPoint 2400 when initiated using the traditional handshake, these were reliably detected when delivered via a session initiated by the split-handshake technique as well. However, this does not appear to be due to any special consideration of session establishment, the detectable attacks were also blocked in the absence of any session establishment prior to the malicious payloads. This indicates that the TippingPoint device is unaware of a session context when it comes to inspecting TCP packets for malicious traffic, and does not consider the presence of an established session when performing its intended detection functions.

6.3.3 Juniper SRX 5800

Of the three devices under test, the Juniper SRX 5800 produced the most dramatic results when running a typical configuration. Like the TippingPoint appliance, the Juniper system was tested against a BreakingPoint test attacks under normal conditions, and attacks were selected which met the above-listed criteria and were successfully blocked by the Juniper IPS component.

Again, as a control, attacks are re-run without a proper session establishment phase of the TCP connection. Unlike the TippingPoint device, the Juniper system appears to be aware of

session state -- the non-established session attacks were (correctly) detected as invalid TCP traffic by the Juniper system, and dropped, indicating the Juniper system is not normally susceptible to Stick-style attacks of false positives.

Finally, the detectable attacks were retested using the split-handshake methodology, and of these attacks, none were blocked. In this scenario, it appears that under a default configuration, nearly all protection against malicious servers that clients normally enjoy in an SRX-controlled network is obviated by employing the split-handshake technique.

Initially, this indicates that the split-handshake is a very effective evasion technique, at least against the Juniper system in a default configuration. However, the Juniper system does have available an (apparently unique) IPS signature, the “TCP:AUDIT:S2C-SIMUL-SYN” attack object which is specifically intended to detect initial simultaneous-open sequence of packets [14]. This protocol anomaly object appears effective in shielding client-side targets from both simultaneous-open and split-handshake sessions, and after discussing the issue with Juniper engineers, the authors have learned this filter will be enabled by default for Juniper IPS products going forward.

7. Split Handshakes and Network Address Translation (NAT)

While the IPS/IDS implications of the split-handshake are interesting in and of themselves, other network devices are also responsible for tracking the state of new and newly-established sessions. Due to a perceived lack of adequate IPv4 address space, the Network Address Translation (NAT) strategy was developed. NAT devices allow endpoint TCP hosts with non-routable internal IP addresses to communicate across the Internet, and a pair of intervening NAT devices enables two machines with non-routable addresses to communicate to each other. Today, a large percentage of both home and office-based client end point hosts reside on networks with private address space with a NAT gateway.

The standards for the behavior of NAT strategies and devices is codified in RFC 5382 [15], which specifically requires that all conforming NAT gateways "handle the TCP simultaneous-open mode of connection initiation," in section 4.2. Because of this requirement, it can be presumed, then, that many NAT devices will allow for both simultaneous-open connections and split-handshake connections.

7.1. Cisco PIX-515E

To determine if split-handshake sessions are feasible over the Internet, informal testing was conducted using the Cisco PIX-515E, a firewall appliance with NAT capabilities, and a Cisco 2631 router configured to perform NAT. Like the TippingPoint and Juniper testing performed in Section 6, the Cisco PIX-515E was subjected to attacks using both normal TCP handshaking and split-handshaking. It was discovered that that the PIX was unable to recognize the split-handshake as a valid form of session establishment, and would not route packets through the NAT'ed address.

7.2. Cisco 2631

Conversely, the Cisco 2631 router did allow the split-handshake packets to traverse the NAT. The router was used as an edge router connected to the Internet, with the fakestack.rb script running on an NAT'ed internal address. A client across the Internet, approximately 14 hops away at the time of the test, was able to traverse several networks, successfully established a connection with the fakestack.rb server, and retrieved the payload as normal.

More formal testing on the specific NAT implications of simultaneous-open was performed by Guha and Francis and documented much more completely in their work, "Characterization and Measurement of of TCP Traversal through NATs and Firewalls" [16]. Since support of simultaneous-open seems to be a prerequisite for support for the split-handshake, it can be presumed that any NAT device which is incapable of handling simultaneous-open is similarly incapable of handling the split-handshake technique. However, the converse of this cannot be assumed without specific NAT testing of split-handshake sessions.

8. Split Handshakes and Port Scanning

Port scanning is most often associated with malicious network behavior, though this stigma has faded considerably over time. Today, network engineers and attackers alike rely on port scanning techniques to determine the status of listening TCP ports on remote servers. For legitimate network engineers, port scanning is often a first step in diagnosing network application issues and testing host and network-based firewall configurations. Attackers, too, use port scanning to test firewall configurations, but usually with an aim to discover mis-configured or vulnerable services.

8.1 Port Scanning Test Methodology

To determine the effects of TCP handshake splitting on port scanning activities, two popular free port scanners were used, Insecure.Org's Nmap Security Scanner [17] and Foundstone's ScanLine [18]. Nmap is a highly portable, open source port scanner, and is arguably the industry standard for network engineers and attackers alike, as it offers a wide variety of options and several different styles of scanning network environments. ScanLine is more single-purpose and is a closed source Windows-based application, but is highly efficient on Microsoft Windows platforms for detecting the state of remote network ports.

For both tests, the target of the scanning is a remote Ubuntu-based host running the fakestack.rb script. In order to differentiate between a listening, filtered, and closed port, host firewall rules on the fakestack.rb host will allow connections to listening port TCP/22 (which will produce a SYN/ACK when a SYN is received), the closed port of TCP/8081 (which will produce a RST when a SYN is received). The firewall will not interfere with connections to TCP/8080 (these will instead be handled by fakestack.rb), and will silently drop connections to TCP/8082 (which will result in no response from the server at all). This port configuration is summarized here in Table 7.

Table 7. Target (server) host port status. The four ports, representing all possible initial states

TCP Port	Status
22	Open (normal stack)
8080	Open (fakestack.rb)
8081	Closed
8082	Closed, Filtered

8.2 Nmap

Nmap has several modes for detecting listening TCP ports, and for testing purposes, the two most common modes were chosen, the TCP SYN scan (using the command line option "-sS"), and the TCP connect scan (using the command line option "-sT"). If a user has administrative privileges on the machine from which he is scanning, the SYN scan is chosen by default, and if not, the connect scan is attempted. The main difference between the two is that the SYN scan generates packets internally, and inspects packets directly, while the connect scan relies on the scanning host's operating system to initiate and complete TCP sessions.

In the connection scan configuration, Nmap correctly determines the state of all four tested ports: TCP/22 and TCP/8080 are both reported as "open," TCP/8081 is reported as "closed," and TCP/8082 is reported as "filtered." Since the connection to TCP/8082 never moved out of the SYN_SENT state within the time limit, Nmap is able to make this determination by querying the OS's network state table. Similarly, since the network state table is ESTABLISHED for both ports TCP/22 and TCP/8080, nmap reports the port as open. Finally, since there is no state associated with TCP/8081, Nmap correctly interprets this as a closed port which must have seen a RST.

By contrast, using Nmap's SYN scanning mode, Nmap fails to report TCP/8080's state correctly. Instead of "open" (as the port appears to be for the normal, underlying TCP stack), Nmap reports this port as "filtered." This is due to Nmap's reliance on detecting, specifically, a SYN/ACK packet in response to its initial SYN.

In other words, port scanners which rely on a higher privileged process to create and receive packets will fail to detect a TCP port configured to respond with the split-handshake, while only port scanners with fewer privileges will successfully detect such listening ports.

8.3 ScanLine

In the ScanLine case, only Microsoft Windows platforms may be used to scan, so Microsoft Windows XP, Service Pack 3 (as described in Section 3), was chosen as the scanning host. ScanLine only supports one mode of TCP port scanning, which itself is tied to the underlying operating system's network state table.

Because of this reliance on the operating system to make remote port status determination, ScanLine consistently detects the fakestack.rb port of TCP/8080 as well as the normal TCP/22 port as open, while TCP/8081 and TCP/8082 are both reported as closed. ScanLine, unlike Nmap, makes no effort to distinguish between "closed" and "filtered" ports,

so these negative results are to be expected.

The results of the scanning of both Nmap and ScanLine are detailed in Table 8.

Table 8. Port Scanner Results. The four ports, as reported by Nmap SYN Scan, Nmap Connection Scan

TCP Port	Status	Nmap -sS	Nmap -sT	ScanLine
22	Open (normal stack)	Open	Open	Open
8080	Open (fakestack.rb)	Filtered	Open	Open
8081	Closed	Closed	Closed	Closed
8082	Closed, Filtered	Filtered	Filtered	Closed

9. Split Handshakes and Host Firewalls

Finally, each of test client hosts described in Section 3 were tested with their respective default host-based firewalls enabled, in order to determine the usefulness of the split-handshake with a normal, though possibly non-default, configuration for such clients.

9.1 Microsoft and Apple Host Firewalls

On Microsoft Windows XP, the Windows Firewall is enabled by default as of Windows XP Service Pack 2 (and 3). Enabling and disabling the firewall appears to have no effect on the acceptance of split-handshake sessions. Identical behavior was observed for Apple OS X, although as of the OS X "Snow Leopard" release, it is important to note the firewall is disabled by default. For both firewalls, there is no combination of standard options which can disallow servers using the split-handshake functionality while simultaneously allowing the normal three way handshake.

9.2 Linux Host Firewalls

The default firewall for Ubuntu Linux (known as netfilter/IPTables), like the Apple firewall, is disabled by default. Enabling the firewall with default settings using the command line tool "ufw" (or "uncomplicated firewall") allows for normal client-initiated sessions, but will cause incoming SYNs in response to client SYNs to be silently discarded. This is unsurprising, given the recent discussions on the netfilter developer mailing list [19]. To summarize that discussion, netfilter/IPTables did not intend to support simultaneous-open as of May of 2009, but is expected to offer complete support "soon."

As mentioned in Section 7, it may be presumed that any device which does not support simultaneous-open connections also cannot support split-handshakes, but with an important caveat that the converse cannot be said with certainty.

10. Conclusions and Recommendations

The research presented in this paper focuses on four main areas where the

split-handshake technique is shown to have a practical impact on the security posture of modern TCP/IP networks.

For network inspection devices such as the intrusion detection and intrusion prevention systems audited in section 6, if the device maintains a notion of “client” and “server” based solely on the directionality of the SYN packet or SYN/ACK packet, this determination may prove to be incorrect when the true server side of the connection employs the split-handshake. This fallacy, in turn, can limit the effectiveness of the IDS/IPS system. Server-based attacks, such as those performed by a malicious web site or FTP server, will appear to be headed the wrong direction as far as the inspection logic is concerned – if the decision to block a packet or terminate a session depends on this logic, the malicious payload may be allowed through. Thus, IDS/IPS vendors should take care to account for the split-handshake when inspecting attacks which target client hosts. This can be accomplished most easily by classifying the split-handshake itself as malicious and blocking sessions which exhibit this technique for TCP session setup. Alternatively, if simultaneous-open or split-handshake functionality is desired for interoperability reasons, IDS/IPS manufacturers should take care to ensure the models employed to determine directionality are adequately tested under these uncommon circumstances.

In the case of the network address translation problem documented in section 7, vendors of NAT devices should decide if the requirement to support simultaneous-open connections (as specified in RFC 5382) is a reasonable requirement to enforce. The decision to support only certain sections of RFCs in light of common security problems is certainly not without precedent. For example, it is not uncommon in moderately secure environments for hosts to silently drop incoming ICMP Echo Requests, even though RFC 1122 dictates that hosts must respond to ICMP Echo Requests [20]. At the very least, NAT device manufacturers should expose options to the user to support simultaneous-open (and by extension, split-handshake) sessions, so that network administrators may decide for themselves if the risk of allowing split-handshake sessions is worthwhile for their particular enterprises.

Port scanners, as described in section 8, should accurately detect the presence of network services which support split-handshake sessions, if only to assist in accurately assessing the requirement for simultaneous-open for NAT devices. Indeed, it would be ideal if port scanners would classify split-open services as a special case of “open,” so that further investigation can be targeted to these services. As demonstrated in section 8, the most common port scanning technique of sending SYN packets and collecting SYN/ACK responses is inadequate to determine the true profile of services offered by a TCP/IP host, and this failure can leave malicious servers undetected.

Finally, in the case of host-based firewalls, as illustrated in section 9, the design purpose of these applications is to shield hosts from undesired, often malicious, traffic. As with the case of NAT devices, these devices should expose a specific user option to enable or disable sessions which require split-open functionality, with a default deny rule.

11. Prior Work

While there exists some research material discussing the practical effects of the simultaneous-open method of TCP session initiation (most notably, the exhaustive work by Guha et. al. in the creation of NAT standards), the authors are unable to locate any mention of the TCP split-handshake technique and its application in evading detection by network security devices prior to this research. While this technique appears novel, though, it is not the first unanticipated feature of TCP which can lend itself to a practical malicious application.

The seminal work on such techniques is Ptacek and Newsham's paper, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," wherein the authors describe various methods to cause intrusion detection systems to misclassify traffic. Specifically, the Ptacek and Newsham describe an insertion attack, where overlapping IP fragments are designed specifically to avoid reassembly by an IDS, but can be reassembled by a victim host, and a method of evasion involving reordering and invalid TCP packets to achieve the same effect [21].

12. Further Research

Only a very small subset of devices which may behave in unexpected ways with the split-handshake technique have been tested by the authors. There are likely hundreds of hardware devices and software applications, produced by dozens of vendors, which can be considered network session aware. Through this small group of network devices and applications, we can expect the behavior of such network aware devices to vary wildly, and often unexpectedly. There is much testing work to be done by these individual vendors and the network engineering and network security communities at large, and a method of evasion involving reordering and invalid TCP packets to achieve the same effect [21].

Acknowledgements

The testing equipment to verify the research was supported by BreakingPoint Systems. Special thanks are offered to BreakingPoint's security and application protocol departments, Juniper Networks, and the "Austin Hackers Anonymous!" group, for their invaluable insight and verification of the split-handshake effect, involving reordering and invalid TCP packets to achieve the same effect [21].

References

[1] John, Wolfgang & Tafvelin, Sven, "Analysis of Internet Backbone Traffic and Header Anomalies Observed". IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, Pp 111-116. October 2007.

[2] Cooperative Association for Internet Data Analysis (CAIDA), “Analyzing UDP Usage in Internet Traffic”. Available at: http://www.caida.org/research/traffic-analysis/tcpudpratio/

[3] Postel, Jon, “User datagram protocol”. ARPANET Working Group Requests for Comments, no. 768. August 1980.

[4] Postel, Jon, “Internet control message protocol”. ARPANET Working Group Requests for Comments, no. 792. September 1981.

[5] Postel, Jon, “Transmission control protocol”. ARPANET Working Group Requests for Comments, no. 793. September 1981.

[6] United States Computer Emergency Response Team (US-CERT), “CERT advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks”. Available at: http://www.cert.org/advisories/CA-1996-21.html

[7] Stevens, Richard (1994). 18.8 Simultaneous open. In *TCP/IP Illustrated, Volume 1: The Protocols* (pp. 250-252). Boston; Addison-Wesley.

[8] TippingPoint, “TippingPoint Intrusion Prevention Systems”. Available at: http://www.tippingpoint.com/products_ips.html

[9] Juniper Networks, “SRX-Series Hardware: Documentation”. Available at: http://www.juniper.net/techpubs/hardware/srx-series.html

[10] Snort Project, “Snort User's Manual”. Available at: http://www.snort.org/assets/125/snort_manual-2_8_5_1.pdf

[11] Beardsley, Tod, “TCP Portals: The Handshake's a Lie!”. BreakingPoint Labs Blog. Available at: http://www.breakingpointsystems.com/community/blog/tcp-portals-the-three-way-handshake-is-a-lie

[12] Riden, Jamie, “Client-Side Attacks”. The Honeynet Project. Available at: http://www.honeynet.org/node/157

[13] Vreugdenhil, P. & TippingPoint, “Microsoft Office OWC10.Spreadsheet ActiveX msDataSourceObject() Heap Corruption Vulnerability”. ZDI Advisories. Available at: http://www.zerodayinitiative.com/advisories/ZDI-09-054/

[14] Juniper Networks, “TCP: S2C Ambiguity Simultaneous SYN”. Attack Objects Index. Available at: https://services.netscreen.com/restricted/sigupdates/nsm-updates/HTML/TCP:AUDIT:S2C-SIMUL-SYN.html

[15] Guha, S. ed., “NAT Behavioral Requirements for TCP”. Network Working Group Requests for Comments, no. 5382. October 2008.

[16] Guha, S., & Francis, P, “Characterization and Measurement of TCP Traversal Through NATs and Firewalls”. IMC '05: Proceedings of the 5th ACM SIGCOMM conference on

Internet measurement, Pp 199-211. September 2005.

[17] Lyon, Gordon, ed., “Nmap Reference Guide”. Available at: http://nmap.org/book/man.html

[18] Foundstone, Inc., “ScanLine v1.01: Command Line Port Scanner”. Available at: http://www.foundstone.com/us/resources/proddesc/scanline.htm

[19] Kadlecsik, Jozef, “TCP simultaneous open using iptables NAT?”. Available at: http://marc.info/?l=netfilter&m=124386207628596&w=2

[20] Braden, R. ed., “Requirements for Internet Hosts – Communications Layers”. Network Working Group Requests for Comments, no. 1122. October 1989.

[21] Ptacek, T. & Newsham, T., “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection”. Available at: http://insecure.org/stf/secnet_ids/secnet_ids.html

Appendix

Appendix 1. Fakestack.rb

Fakestack.rb is a Ruby script designed to simulate a server which has implemented the TCP split-handshake connection procedure. It has been tested successfully using Ruby 1.8.6 and 1.8.7, and requires the modules PcapRub and PacketFu, both available at <http://code.google.com/p/packetfu/>.

In order to function correctly, the host running fakestack.rb must inhibit the operating system from responding to calls to fakestack.rb's listening port. This is easily accomplished on an Ubuntu Linux server using the "ufw" (uncomplicated firewall) iptables/netfilter control application.

```
#!/usr/bin/env ruby

# Go get PacketFu and PcapRub at http://code.google.com/p/packetfu

require 'packetfu'

# USAGE: sudo fakestack.rb eth0 8080

$iface = ARGV[0] || "eth0"

$port = (ARGV[1] || 8080).to_i

puts "Watching for SYNs on #{$iface} to #{$port}..."

cap = PacketFu::Capture.new(:iface => $iface, :start => true, :filter => "tcp and dst port #{$port}")

$config = PacketFu::Utils.whoami?(:iface => $iface)

$http_payload =
```

```
"PGh0bWw+PGJvZHk+CjxwPjxpPlRoaXMgd2FzIGEgdHJpdW1waCw8YnI+Ckkn\n"+
"bSBtYWtpbmcgYSBub3RlIGhlcmU6IEhVR0UgU1VDQ0VTUy48YnI+CihJdCdz\n"+
"IGhhcmQgdG8gb3ZlcnN0YXRlIG15IHNhdGlzZmFjdGlvbi4pCjwvaT48L3A+\n"+
"CjwvYm9keT48L2h0bWw+Cg==\n"
$http_response =<<EOF
HTTP/1.0 200 Ok\r
Server: micro_httpd\r
Date: #{Time.now.gmtime}\r
Content-Type: text/html; utf-8\r
Content-Length: #{$http_payload.unpack("m").first.size}\r
Last-Modified: #{Time.now.gmtime}\r
Connection: close\r
\r
#{$http_payload.unpack("m").first}
EOF
puts "Listening on port #{$port}..."
loop do
$established = false
$disconnected = false
$http_sent = false
puts "Setting up the fake stack..."
while($disconnected == false) do
  cap.stream.each do |pkt|
    packet = PacketFu::Packet.parse pkt
    # Establish a session.
    if packet.tcp_flags.syn == 1 && !$established
      if packet.tcp_flags.ack == 0
        puts "Got a SYN with seq = #{seq = packet.tcp_seq} from
#{packet.ip_saddr}"
```

```
      puts "Generating packets..."
      ack_packet = PacketFu::TCPPacket.new(:config => $config)
      ack_packet.ip_daddr= packet.ip_saddr
      ack_packet.tcp_src = $port
      ack_packet.tcp_dst = packet.tcp_src
      ack_packet.tcp_ack = seq + 1
      puts "Duping and splitting SYN and ACK..."
      syn_packet = ack_packet.dup
      ack_packet.tcp_flags.syn = 0
      ack_packet.tcp_flags.ack = 1
      ack_packet.recalc
      # Comment out the next two lines to avoid sending the ack.
      puts "Sending ACK..."
      ack_packet.to_w($iface)
      syn_packet.tcp_flags.urg = 0
      syn_packet.tcp_flags.ack = 0
      syn_packet.tcp_flags.psh = 0
      syn_packet.tcp_flags.rst = 0
      syn_packet.tcp_flags.syn = 1
      syn_packet.tcp_flags.fin = 0
      syn_packet.tcp_ack = 0
      syn_packet.tcp_seq = rand(0xffffffff)+1 # New sequence number
      syn_packet.recalc
      puts "Sending SYN..."
      syn_packet.to_w($iface)
    else
      puts "Got a SYNACK with seq = #{seq = packet.tcp_seq} from
#{packet.ip_saddr}"
      ack_packet = PacketFu::TCPPacket.new(:config => $config)
```

```
      ack_packet.ip_daddr = packet.ip_saddr
      ack_packet.tcp_src = $port
      ack_packet.tcp_dst = packet.tcp_src
      ack_packet.tcp_ack = seq + 1
      ack_packet.tcp_seq = packet.tcp_ack
      ack_packet.tcp_flags.ack = 1
      ack_packet.recalc
      puts "Acking the SYNACK. The handshake's a LIE!"
      ack_packet.to_w($iface)
      $established = true
    end
  elsif $established && !$http_sent
    if packet.tcp_flags.ack == 1
      if packet.tcp_flags.psh == 1
        puts "Got a PSH/ACK with seq = #{seq = packet.tcp_seq}, probably the
GET..."
        puts packet.payload.inspect
        ack_packet = PacketFu::TCPPacket.new(:config => $config)
        ack_packet.ip_daddr = packet.ip_saddr
        ack_packet.tcp_src = $port
        ack_packet.tcp_dst = packet.tcp_src
        ack_packet.tcp_ack = seq + packet.payload.size
        ack_packet.tcp_seq = packet.tcp_ack
        # Normal people: One packet GET
        # IE8: Two packet GET, break right before the Keep-Alive.
        # IE6: Three packets, it screws me up at the moment. Just use Firefox.
        if packet.payload.size > 0 # Thanks for the 2 packet GET, IE.
          ack_packet.tcp_flags.ack = 1
          ack_packet.recalc
```

```
          puts "Acking the GET..."
          ack_packet.to_w($iface)
          http_packet = ack_packet.dup
          http_packet.payload = $http_response[0,1400] # Enforce a max size.
          http_packet.recalc
          puts "Delivering the payload..."
          http_packet.to_w($iface)
          $http_sent = true
        end
      end
    end
  elsif $http_sent
    if packet.tcp_flags.rst == 0
      puts "Payload ack'ed with seq = #{seq = packet.tcp_seq}. RSTing, since
FINs are for chumps."
      rst_packet = PacketFu::TCPPacket.new(:config => $config)
      rst_packet.ip_daddr = packet.ip_saddr
      rst_packet.tcp_src = $port
      rst_packet.tcp_dst = packet.tcp_src
      rst_packet.tcp_ack = seq
      rst_packet.tcp_seq = packet.tcp_ack
      rst_packet.tcp_flags.rst = 1
      rst_packet.tcp_flags.ack = 1
      $disconnected = true
      rst_packet.recalc
      puts "Sent a RST"
      rst_packet.to_w($iface)
    else # Got a RST, thanks!
      $disconnected = true
```

```
        end
      end
    end
  end
end
end
```