2012 IEEE Symposium on Security and Privacy

Off-Path TCP Sequence Number Inference Attack
How Firewall Middleboxes Reduce Security

Zhiyun Qian, Z. Morley Mao
{zhiyunq,zmao}@umich.edu, University of Michigan

***Abstract*—In this paper, we report a newly discovered "off-path TCP sequence number inference" attack enabled by firewall middleboxes. It allows an off-path (i.e., not man-in-the-middle) attacker to hijack a TCP connection and inject malicious content, effectively granting the attacker write-only permission on the connection. For instance, with the help of unprivileged malware, we demonstrate that a successful attack can hijack an HTTP session and return a phishing Facebook login page issued by a browser. With the same mechanisms, it is also possible to inject malicious Javascript to post tweets or follow other people on behalf of the victim. The TCP sequence number inference attack is mainly enabled by the sequence-number-checking firewall middleboxes. Through carefully-designed and well-timed probing, the TCP sequence number state kept on the firewall middlebox can be leaked to an off-path attacker. We found such firewall middleboxes to be very popular in cellular networks — at least 31.5% of the 149 measured networks deploy such firewalls. Finally, since the sequence-number-checking feature is enabled by design, it is unclear how to mitigate the problem easily.**

I. Introduction

TCP was initially designed without many security considerations and has been evolving for years with patches to address various security holes. One of the critical patches is the randomization of TCP initial sequence numbers (ISN) which can guard against off-path spoofing attacks attempting to inject packets with a forged source address (for data injection or reset attacks) [19]. ISN randomization prevents sequence numbers from being predicted, thus arbitrarily injected packets are likely to have invalid sequence numbers which are simply discarded at the receiver.

Firewall vendors soon realized that they can in fact perform sequence number checking at network-based firewalls and actively drop invalid packets even before they can reach end-hosts, a functionality advertised in products from major firewall vendors [15], [21], [3]. This feature is believed to enhance security due to the early discard of injected packets and the resulting reduced wasted network and host resources. Ironically, we discover that the very same feature in fact allows an attacker to determine the valid sequence number by probing and checking which sequence numbers are valid using side-channels as feedback. We name this attack "TCP sequence number inference attack".

Using the sequence number inference as a building block, we design and implement a number of attacks including TCP hijack. In general, all of our attacks require IP spoofing, which is still very common on the Internet according to a recent study [13]. Besides IP spoofing, different attacks may have different requirements. For instance, a long-lived connection inference attack requires only a remote attacker to perform remote scanning and injection of exploits on services that run over unencrypted long-lived connections (e.g., HTTP-based push services [5]). In contrast, TCP hijack requires an unprivileged and lightweight malware residing on the victim.

We implement all except one attacks that we proposed. They are experimented specifically on mobile devices operating under a nation-wide carrier that extensively deploys sequence-number-checking firewall middleboxes. We show that a successful TCP hijacking allows an attacker to take over a connection and inject malicious payload right after the connection is established. For instance, we demonstrated that the attack can return a phishing Facebook login page, as shown in a short YouTube video [9]. We can also inject malicious Javascript to perform actions on behalf of a victim user, e.g., to post tweets or follow other people.

We emphasize that even though our attack is implemented on mobile phones, it is not restricted to mobile devices or mobile networks. The reason for choosing this specific setting is that mobile networks make our experiments easier to carry out, as we have direct access to end devices behind the firewall. Also, the attack model of most TCP hijacking requires an unprivileged malware residing on the victim which fits the smartphone model well in that users often download untrusted third-party apps.

According to our measurement study, such firewalls are deployed in many carriers – at least 31.5% out of 149. This means the sequence number inference attack is widely applicable. It is likely to become more prevalent in the future as such functionality is considered to be advanced and desirable. Moreover, since we exploit the very behavior of sequence number checking — a firewall feature by design, it is unclear how to easily address the problem besides disabling the feature or employing application-layer encryption.

Our study makes the following contributions:

- We discover and report the TCP sequence number inference attack enabled by firewall middleboxes. We also devise techniques leveraging it as a building block to achieve TCP hijacking and a number of other attacks.
- We measure the popularity and characteristics of such middleboxes and found they are widely deployed in major cellular networks throughout the world.
- We survey a broad list of impacted applications ranging from Web-based attacks of directing users to a

DOI 10.1109/SP.2012.29

spoofed login page, application-based attacks of injecting malicious links to Windows Live Messenger chat messages, to attacks against servers in the form of DoS and spamming.

In the rest of the paper, we describe related attacks in §II and fundamentals of the TCP sequence number inference attack in §III. Next, we discuss the detailed attack requirements and design in §IV-B, and implementation results in §V. In §VI, we measure how many cellular networks have deployed the sequence-number-checking firewall middleboxes. In §VII, we describe what applications are impacted. Finally, we discuss what went wrong and conclude in §VIII.

II. Related Work

TCP-sequence-number-related attacks. In the past two decades, researchers have discovered a number of TCP attacks [33], [1], [12]. The most notable ones are TCP sequence number prediction [1] and TCP reset attack [33], [19]. Both attacks are related to IP spoofing and TCP sequence number, which are also the focus of our attack.

Sequence number prediction attack. Twenty years ago, certain OSes select the TCP Initial Sequence Numbers (ISN) based on a global counter which is incremented by a constant amount every second. It allows an attacker who has opened a connection to a server to obtain its current global counter and predict its next ISN with high confidence. With this prediction ability, an attacker can spoof the IP of a trusted client when talking to a target server, and complete the TCP 3-way handshake based on the guess of server's next ISN. The problem is fixed after the randomization of ISN is standardized and adopted.

Blind TCP RST attack. As described in RFC 5961 [27], the attack is possible because a reset (RST) packet is accepted as long as its sequence number falls within the current TCP receive window. In a long-lived connection (e.g., BGP sessions), an attacker knowing the target four-tuple can simply use brute force all sequence number ranges. Watson [33] has analyzed in detail the number of packets needed under various OS/setup taking into consider the source port can be random. A number of proposals, e.g., requiring the RST sequence number to exactly match the expected sequence number, are discussed in RFC 4953 [31]; however, they are not widely adopted likely due to backward-compatibility issue and the fact that source port randomization can already alleviate the problem.

Sequence number inference attack. The first known sequence number inference attack is described in 1999 [6] where the Linux 2.0.X kernel has a bug that silently drops the third packet in the three-way handshake when the ACK number is too small, and sends a reset when the ACK number is too big. Such behavior allows an attacker to infer the correct ACK number in an ACK packet to complete the TCP connection. However, it is an isolated bug that has been fixed since then. The other relevant attack described in Phrack magazine [24] infers the sequence number by relying on the fact that a packet with in-window sequence number can be silently dropped and a packet with out-of-window sequence number will trigger an outgoing ACK packet. The limitations of this work are that 1) it requires sending two orders of magnitude more packets considering the TCP receive window is usually very small (e.g., 16K); 2) it relies on a very noisy feedback channel (i.e., IPID) on the end-host. It is only targeting at long-lived connections where the host has low traffic rate.

Side-channel information leakage. Side channel leaks are known for decades. There are a wide range of side channels including CPU usage, power usage, shared memory/files, *etc.*. A variety of attacks are possible using side channels [35], [30], [14], [16]. For example, researchers have shown that it is possible infer keystrokes through shared registers [35] and packet size/timing analysis on encrypted traffic [30], [14]. On the newly emerged smartphones, various on-board sensors can also be used as side-channels. For instance, Soundcomber [29] uses the audio sensor to stealthily record credit card numbers entered through keypad. In our work, we also rely on network side-channels to infer TCP sequence number.

Middlebox security. Firewall middleboxes have been introduced for many years [21], [3]. Previous work has discovered various vulnerabilities on the firewalls themselves that range from not properly checking the sequence number of TCP RST packets resulting in DoS attack on active connections [4], to failure to correctly process specially-crafted packets forcing the middlebox to reload or hang. A more complete summary on firewall vulnerabilities can be found in a study done by Kamara *et al.* [22].

III. Fundamentals of TCP the Sequence Number Inference Attack

In this section, we introduce the sequence number inference attack by first describing the behavior of sequence number checking firewalls, then discussing how to use side channels to infer the sequence number state kept on such firewalls, and finally illustrating the attack by an example.

A. Sequence-Number-Checking Firewalls

Many stateful firewalls that track TCP state (e.g., SYN-SENT, ESTABLISHED) also track the sequence numbers of the bidirectional traffic. All major vendors including Cisco, Juniper, and Check Point have such products [15], [21], [3]. Typically, once a TCP connection is established, it only allows packets with sequence numbers within a window of the previously seen sequence numbers to go through. As an example illustrated in Figure 1(a), when the client and server exchange SYN and SYN-ACK packets, the firewall remembers the current sequence number to be X and Y for client and server respectively. Later packets originated from both sides will have to be in the window of X or Y, otherwise, they will be silently dropped. Such a feature is to prevent arbitrary packets from being injected into the connection. A window is needed because packets may arrive

out of order and should still be allowed by the firewall. Note that acknowledgment number is typically not checked by the firewall because packets may or may not even set the ACK flag. In fact, we verified that all major OSes accept incoming data packets that do not have ACK flag set. Based on our observation and experiments with real firewalls (See §VI), we found that sequence-number-checking firewalls may behave differently in the following ways.

Window size: Ideally, the firewall should acquire accurate state information associated with the end-host and accepts packets if and only if they will be accepted by the end-host. For instance, this requires the firewall to dynamically track the advertised receive window of the end-host, which can be expensive in terms of overhead. In practice, we found that firewalls typically initializes the window size to a fixed value according to the window scaling factor (a TCP option) carried in the SYN and SYN-ACK packet. It is typically calculated as $64K \times 2^N$, where N is the window scaling factor. The maximum possible receive window size is 1G and some firewalls simply use the fixed 1G window directly.

Left-only or right-only window: Some firewalls may only have a left window or right window such as (Y-WIN, Y) or (Y,Y+WIN). As discussed later, we found the nation-wide carrier that we studied indeed has left-only window firewalls because it buffers out-of-order (right-window) packets. Similar behavior was previously reported [32].

Window moving behavior: We found two general cases when the existing window will move: 1) In-order TCP packet arrives. It implies that the window can only move forward. We thus name it *window advancing*. 2) Any packet with an in-window sequence number. For instance, if Z is in (Y-WIN, Y+WIN), it can shift the window to (Z-WIN,Z+WIN). It implies that the window can either move forward or backward. We name this behavior *window shifting*. For the rest of the paper, we assume the *window advancing* behavior, which is more popular according to our measurement study, unless explicitly stated otherwise.

Such firewall products claim that the sequence-number-checking feature can improve security by defending against connection hijacking [3], which ironically turns out to be the opposite. We demonstrate that as long as the target four-tuple (source/destination IP and port) is known, an attacker can probe using the spoofed target four-tuple to infer the valid sequence number, due to the very behavior that the firewall treats packets with in-window and out-of-window sequence numbers differently. Figure 1(b) illustrates such an attack model. The firewall's differentiation behavior, coupled with the ability that an attacker can get feedback regarding which packets are allowed, effectively breaks the non-interference security property [16]. We discuss how to obtain the target four-tuple and feedback below.

B. Obtaining Four Tuples – Threat Model

We outline three main threat models where the target four-tuple can be known:

(1). On-site TCP injection/hijacking. An unprivileged malware runs on the client with access to network and the list of active connections through standard OS interface (e.g., "netstat" command). It cannot tamper with other applications or OS services. A successful TCP sequence number inference attack in this case can compromise the security of other applications or even OS services.

Note that the attacker can also carry out other local privilege-escalation attacks under this threat model, but the most known privilege-escalation attacks on Android are still at the application layer without breaking the OS sandbox [18]. In contrast, our attack allows the malware to break the sandbox and compromise the security of other apps. Regardless, our attack provides additional capabilities to the attackers.

(2). Off-site TCP injection. An attacker simply guesses the four tuples. For instance, popular services typically have well-known port numbers and a few load-balancing IP addresses. To attack such services, the attacker only needs to enumerate client IP and port number. This usually works only when the target connection is long-lived, e.g., instant messenger or push notification services.

(3). Establish TCP connection using spoofed IPs. An attacker in this case initiates the connection himself, in which case the four tuples are obviously known. Coupled with IP spoofing, an attacker can use this attack to establish TCP connections with a target server using spoofed IPs (e.g., for spamming or denial-of-service).

C. Obtaining Feedback – Side Channels

As mentioned, to launch the sequence number inference attack, an attacker needs feedback regarding which packets went through the firewall. We discover two main side-channels that can serve the purpose:

1. OS packet counters: On Linux, the *procfs* [23] exposes aggregated information on the number of incoming/outgoing TCP packets, with or without errors (e.g., wrong checksum). Alternatively, "netstat -s" exposes a similar set of information on all major OSes including Windows, Linux, BSD, and smartphone OSes like Android and iOS. If the packet went through the firewall middlebox, then the incoming packet counter will increment accordingly. Although such counters can be noisy as they are aggregated over the entire system, we show that some of the TCP error counters rarely increment under normal conditions and can be leveraged as a clean side channel.

2. IPIDs from responses of intermediate middleboxes: IPID is a 16-bit field in the IP header. In practice, many OSes, including middlebox OSes, have such monotonically incrementing IPIDs (a known side channel for inferring how many packets a target system has sent [26]). In addition, many networks allow intermediate middleboxes (e.g., routers) to reply with "time-to-live (TTL) expired" ICMP messages (See §VI-B for measurement results) to inform the source of a discarded packet due to the TTL field reaching zero. Thus, an attacker can craft packets with

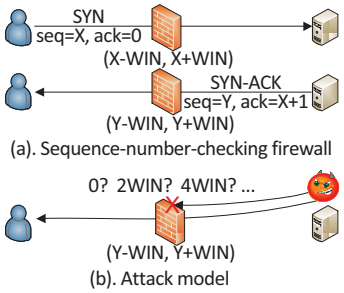

Figure 1: Sequence number checking stateful firewall and attack model

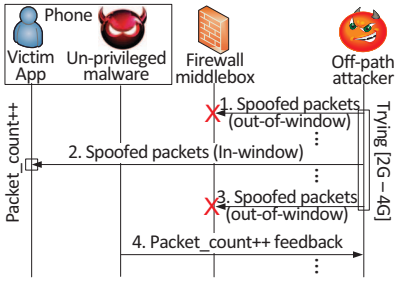

Figure 2: An attacker tries to infer sequence number

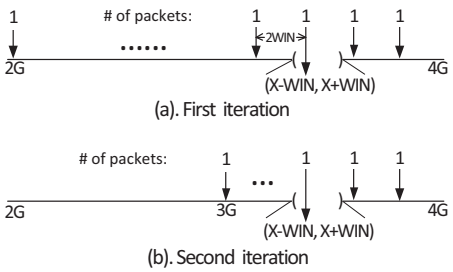

Figure 3: Sequence number space search illustration

TTL values large enough to reach the firewall middlebox, but small enough that they will terminate at an intermediate middlebox instead of the end-host, triggering the TTL-expired messages. By reading the IPID values generated by the intermediate hop before and after sending the spoofed probing packets, an attacker can infer if probing packets went through the firewall.

Both side-channels can serve the same purpose. An attacker can decide which to use depending on their availability and how noisy the side-channels are.

D. Sequence Number Inference

Now that we know how to obtain the target four-tuple and feedback regarding which packets are allowed, we need an efficient way to infer the sequence number. A naive approach is to test out each individual window sequentially. In particular, one can check if 0 is in-window, if 2WIN is in-window, *etc.* as shown in Figure 1(b). However, that requires $\frac{4G}{2WIN}$ round trips to determine which window the sequence number falls in, which can take too long to finish.

In Figure 2, we illustrate a much faster approach – a binary-search-like inference that tries half of the sequence number space at a time (e.g., 0 to 2G) and iteratively narrow down the sequence number. Here we assume the first threat model where an unprivileged malware runs on the client that colludes with an attack server. We also assume that the attacker has prior knowledge of the firewall behavior (e.g., window size), which can be easily obtained offline. Figure 2 illustrates the procedure where the attack server first tries the upper half of the sequence number space [2G,4G). As shown in the figure, packets at time 1 and 3 are dropped and only a single in-window packet at time 2 is allowed. Upon receiving the packet, the phone will increase the packet counter. At time 4, after the attack server finishes probing [2G,4G), it can query the malware for the delta of packet counter before and after the probing. Based on the incremented packet counter, the attack server knows that [2G,4G) is the correct range. Otherwise, it is likely that the other half [0,2G) is the correct one.

In Figure 3, we illustrate this example again via the sequence number space view. In the first iteration trying out [2G,4G), a series of packets are sent with sequence numbers on equally spaced interval of 2WIN (with $\frac{2G}{2WIN}$ number of packets sent). Given every 2WIN range is covered by a packet, one and only one packet will be allowed to go through if the current sequence number kept on the firewall indeed falls in [2G,4G). In the second iteration, it continues to try [3G,4G) to further narrow down the sequence number. Even though the number of packets sent at each iteration can be large (especially at the beginning iterations), it is not hard to see that: 1). the search algorithm takes $\log_2 4G = 32$ iterations to complete, which is the same complexity as a standard binary search algorithm; 2). the larger the WIN is, the fewer probing packets are required. We discuss further optimizations to improve the number of iterations and inference time in §V.

Note that this example assumes *window-advancing* firewalls. In the case of *window-shifting* firewall, similar procedure still applies yet it only allows an attacker to determine a range of possible sequence numbers instead of narrowing down to the exact value. It is because the first in-window packet already erases the original state of the sequence number by shifting the center of the window away. Nevertheless, it still can allow an attacker to narrow down the sequence number to a much smaller range, which in many cases can be inferred using brute force by the attacker. We omit the details here and focus on window-advancing firewalls given the latter is most commonly observed.

E. Timing of Inference and Injection — TCP Hijacking

For the TCP sequence number inference and subsequent data injection to be successful, a critical challenge is timing. If a user is in the middle of a session, injected TCP packets may not be "meaningful" at all. Specifically, since the sequence number inference takes time to finish, the server could already send part or all of the response (e.g., HTTP response). The injected packets then will likely just corrupt the original response, which may or may not achieve the attacker's goal.

To address the challenge, we design and implement a number of *TCP hijacking* attacks (described in §IV-B) where injection can happen at deterministic timing, e.g., right after the TCP three-way handshake. This can, for instance, allow an attacker to inject a complete HTTP response without any interference from the original response. In contrast, *TCP Injection* is a general term that does not

Table I: Summary of identified TCP sequence number inference attacks and their requirements

Req. ID	Requirement explanation	On-site TCP hijacking						Off-site injection		Spoofed conns
		Reset-the-server		Preemptive-SYN		Hit-and-run		URL phishing	Conn infer	
		Packet count	IPID	Packet count	IPID	Packet count	IPID			
C1	Malware on client with Internet access	X	X	X	X	X	X			
C2	Malware can read packet counters	X		X		X				
C3	Malware can read active TCP four tuples	X	X	X	X	X	X			
C4	Client has coarsely predictable ISNs	X	X							
N1	A client can spoof another client's IP			X	X			X		X
N2	A shared responsive intermediate hop		X		X		X	X	X	X
N3	Client network has NAT boxes deployed								X	
N4	Predictable external port if NAT deployed	X	X	X	X	X	X			X
N5	Additional firewall middlebox deployed					X	X			
S1	Legitimate server has stateful firewall	X	X							
S2	Attack server closer to client			X	X					

assume any specific timing of the injection.

IV. TCP Attack Analysis and Design

Applying the basic TCP sequence number inference as a building block, we detail the design of a number of TCP attacks, each associated with a list of corresponding requirements. We show that they are widely applicable and feasible under many client/server/network combinations.

A. Attack Requirements

We first introduce two base requirements for all attacks: 1) the ability to spoof legitimate server's IP on the Internet, and 2) a sequence-number-checking firewall deployed in the client's network or anywhere in the network observing traffic flows in both directions. The former is a known problem and still widely prevalent on today's Internet [13], and the latter is required for the sequence number inference.

Besides the base requirements, we provide a complete list of requirements in Table I, only a subset of which are required for any specific attack. We use "C", "N", and "S" to represent client-side, network, and server-side requirements.

Client-side requirements mainly have to do with malware's capability. For instance, C1 specifies that the malware needs Internet access. C2 requires access to the first side-channel (i.e., packet counter) to obtain feedback. C3 specifies that the malware can run in the background and continuously monitor the creation of any new TCP connection. C1–C3 are common capabilities that an unprivileged program has in modern OSes. To be more stealthy, the malware could hide its monitoring activity until the target app (e.g., browser app) is launched. C4 is a byproduct of the design decision made in many UNIX-like OSes (e.g., Linux 3.0.1 and earlier) where the ISN for different connections are not completely independent. Instead, the high 8 bits for all ISNs is a global number that increments slowly (every five minutes) and only the low 24 bits are produced as random numbers. The design is to balance across security, reliability, and performance, and it is long perceived as a good optimization (more details discussed in [7], [2]). The result of this design is that the ISN of two back-to-back connections will be at most $2^{24} = 16,777,216$ apart.

Network requirements relate to policies in the network. For instance, N1 specifies that client-side IP spoofing is allowed. As discussed, this is fairly common on the Internet [13] and also observable in cellular networks according to a recent study [32]. N2 corresponds to the second side channel to obtain feedback as described in §III-C. The requirement further states that such an intermediate hop must be on the path for both the attacker connection and the victim connection for the feedback to be useful (§VI-B shows more than half of the networks that have sequence-number-checking firewalls satisfy this requirement). N3 simply describes that a standard NAT is deployed in the client's network. N4 says that the NAT-mapped external port has to be predictable which is a typical requirement for P2P applications [28]. The requirement is necessary for on-site attacks that need externally-mapped four tuples (as described in the next section). A recent measurement study on NAT mapping type in cellular networks [32] shows that the majority of the networks satisfy the requirement. N5 states that there is an additional sequence-number-checking firewall deployed in the network, which is actually what we observe in the nation-wide cellular network. Except for N1, other requirements are mostly network design decisions and cannot be classified as "vulnerabilities."

Server-side requirement S1 states that the legitimate server has to deploy host-based stateful firewall that drops out-of-state TCP packets. Many websites such as Facebook and Twitter deploy such firewalls to reduce malicious traffic. For instance, iptables can be easily configured to achieve this [8]. Note that interestingly this security feature on the server turns out to help enable one of the TCP hijacking attack. S2 requires the attack server's network latency to the victim needs to be smaller compared to the legitimate server.

B. Attack Design

In this section, we describe in detail each attack and the corresponding requirements. Specifically, we design three classes of attacks for each threat model as described earlier in §III-B: 1) On-site TCP hijacking/injection. 2) Off-site TCP hijacking/injection. 3) Spoofed connection

establishment. Each class has several attacks with the same goal but different requirements.

1) On-site TCP hijacking: As noted, TCP hijacking allows packets injected right after the connection is established. It is more powerful than the general case of injection but with more requirements. Thus, we focus on the hijacking attack design which also covers the general case of injection. In total, we devise three TCP hijacking attacks and all of which are implemented and tested against the nation-wide cellular network, since all requirements are satisfied in the network (As shown in §V).

The first TCP hijacking is **Reset-the-server**. The high-level idea is to reset the connection on the legitimate server as soon as possible to allow the attacker to claim to be the legitimate server talking to the victim. The key is that such reset packets have to be triggered right after the legitimate server sends SYN-ACK. To achieve this, we leverage requirement C4 which allows an attacker to predict the rough range of victim's ISN and send reset packets with sequence numbers in that range. This is helpful because then the attacker can send much fewer spoofed RST packets (thus with lower bandwidth requirement) compared to enumerating the entire 4G space. Further, after the legitimate server is reset, requirement S1 is necessary as it helps prevent the legitimate server from generating RST upon receiving out-of-state data or ACK packets from the victim. Here we focus on the design of the attack. Implementation and feasibility analysis are covered in §V.

Figure 4 illustrates the attack sequence. Here the attacker is off-path and not man-in-the-middle. It is positioned between the victim and legitimate server for ease of illustration only. Starting at time 1, the victim app first initiates a TCP SYN. At time 2, the malware discovers the new connection attempt by continuously monitoring the output of "netstat", and it immediately notifies the attack server about the new connection including the four tuples. The malware also starts a new connection to the attack server so that the server knows the current ISN. At time 3, the legitimate server receives the SYN, and replies with a SYN-ACK. At time 4, the attack server floods the legitimate server with a number of spoofed RST packets based on the previously gathered ISN. As discussed earlier in §III-E, the RST packets have to arrive before the ACK/request packets at time 5; otherwise, the legitimate server will respond before the attacker can send any malicious content.

From there on, the legitimate server's connection is reset. All future packets from the victim are considered out-of-state and silently dropped due to requirement S1. For instance, the ACK packet received at time 5 is silently discarded. From time 6 to 7, we omit the sequence number inference procedure described earlier in §III-D. At time 8, the attack server can inject data using the inferred sequence number.

Table I summarizes the requirements for the attack. Depending on the side-channel used for feedback, the set of requirements for this attack methodology is (C1,C2,C3,C4,N4,S1) using the packet count feedback, and (C1,C3,C4,N2,N4,S1) using the intermediate hop IPID feedback. Note that N4 is needed because all RST packets need to have the correct external source port number.

The second TCP hijacking is **Preemptive-SYN**. The high-level idea is similar to Reset-the-server in that it also tries to prevent the legitimate server's packets from reaching the client. The difference is that it does so by turning the firewall middlebox's sequence number checking feature against the legitimate server. Remember that the middlebox initializes the current sequence number from SYN and SYN-ACK packet, if an attacker can preemptively send spoofed SYN packets before the legitimate SYN-ACK packet (e.g., when requirement S2 is satisfied), the firewall will initialize the sequence number according to the spoofed SYN instead of the legitimate SYN-ACK. Spoofed SYN packet is allowed due to TCP simultaneous open [25]. The attacker cannot directly spoof a SYN-ACK packet without the knowledge of a valid acknowledge number. Another difference is that such an attack needs requirement N1 to allow the sequence number inference from the client's network. Specifically, a separate attack phone inside the network is required to spoof the victim's IP and infer the sequence number of the victim's SYN. As described later in §V, the firewall is deployed at the *Gateway GPRS Supporting Node* (GGSN) level [34] such that a single attack phone can spoof hundreds of thousands of IPs of other devices. As a result, the attack phone and the victim phone can be in different cities or states as long as they go through the same GGSN. The details are described below.

As shown in Figure 5, initially the victim app sends a TCP SYN packet (with sequence number X) at time 1, followed by the malware reporting the new connection. Due to requirement S2, at time 3, the attack server receives the notification and immediately sends a preemptive SYN (with sequence number Z) which reaches the firewall middlebox before the legitimate server's SYN-ACK. Also, note that the preemptive SYN packet does not actually reach the phone (easily achieved with small TTLs set deliberately by the attacker), necessary to prevent the phone from replying with SYN-ACK which triggers connection reset from the legitimate server and prevents the connection from being established. At time 4, the legitimate server's SYN-ACK packet is dropped at the firewall because its sequence number Y is now considered out-of-window of (Z-WIN,Z+WIN), assuming that Y and Z are unlikely close together. During time 5 and 6, the attack phone tries to infer the sequence number of the victim's original SYN with the intermediate hop feedback. At time 7, after finishing inferring the sequence number, the attack phone reports it to the attack server which then sends a spoofed SYN-ACK with the correct acknowledgment number. Since the victim never actually sees any response after it sends SYN, thinking the delay is likely due to resource issues, it happily

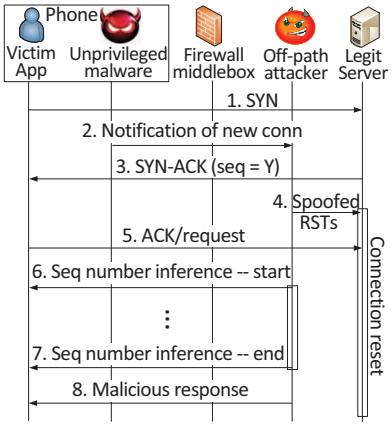

Figure 4: Reset-the-server hijacking

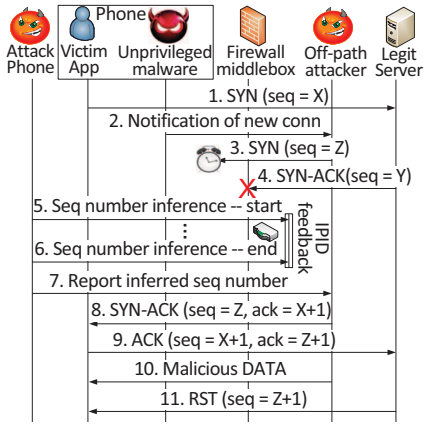

Figure 5: Preemptive-SYN hijacking

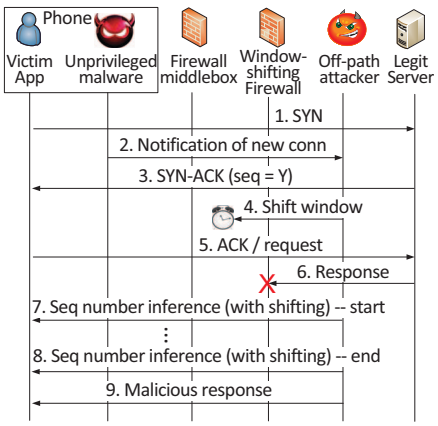

Figure 6: Hit-and-run hijacking

accepts the SYN-ACK and replies with ACK to complete the connection.

There is however still one remaining challenge — the ACK packet at time 9 will trigger a reset once it arrives at the legitimate server, which will terminate the connection immediately. To get around this problem, the attack server has to inject data packets immediately following the spoofed SYN-ACK at time 10 so that it arrives before the RST packet at time 11. As long as the data packet is accepted before the connection is RST, the damage is already done. For instance, we verified that in a HTTP session, a small data packet containing an iframe pointing to a malicious URL still makes the browser follow the URL and load the content even through the connection is reset immediately after.

The requirements are (C1,C2,C3,N1,N4,S2) using packet count feedback, or (C1,C3,N1,N2,N4,S2) using intermediate hop IPID feedback. Here N4 is required because the preemptive SYN packet needs to have the correct external source port number as the destination port.

The last TCP hijacking is **Hit-and-run**. This attack is possible only when the network deploys two different firewall middleboxes, which is what we observed in the nation-wide carrier elaborated in §V. In general, assuming that the sequence number inference is carried out in network external to the mobile device, the two different firewalls have to satisfy the following: a window-shifting firewall is deployed external to a window-advancing firewall. The network may intentionally set up the external firewall for general packet-filtering (which is simpler and potentially cheaper) and the internal one is for more advanced intrusion detection (which requires packet reassembly and incurs more overhead). The problem with this setup is that the window-shifting firewall allows an attacker to intentionally shift the window away from its original position which effectively disallows packets sent from the legitimate server. At the same time, the attacker still can shift the window back when it is necessary to traverse the internal window-advancing firewall to conduct the sequence number inference. This particular two-firewall setup effectively eliminates the requirement C4 and S1 in the Reset-the-server attack. We emphasize that the combined effect of the two firewalls is still a window-advancing firewall and previous two TCP hijacking attacks still work.

Figure 6 illustrates the attack process in detail. In this example, we use the setup of the nation-wide network where the internal window-advancing firewall has a left-only window of 1G. However, in the general case, the attack is possible as long as it is a window-advancing firewall. Time 1–3 match that in the Reset-the-server attack. At time 4, however, instead of resetting the connection on the server, the attacker tries to intentionally shift the window away from its original position. Specifically, regardless of the original window's position, an attacker can send an array of spoofed packets with sequence number 4G, 4G-(WIN-1), 4G-2(WIN-1), ..., all the way to 0. It is not hard to see that the center of the window will be deterministically shifted to 0 (we show the feasibility in §V). This way, at time 6, the legitimate server's response is highly likely to be dropped by the window-shifting firewall (assuming its sequence number has a low probability of being close to 0). Note that packets sent at time 4 do not need to go further beyond the window-shifting firewall, as easily achieved using a small TTL. These TTL-expired ICMP packets are sent to the legitimate server, which may unintentionally terminate the connection on the server side in extremely unlucky situations. Specifically, the ICMP packet embeds the original TCP header which includes the sequence number. The connection will be terminated only if the sequence number happen to exactly match the one used in the SYN-ACK packet. If that happens, then all client's packets in the future will trigger the legitimate server to respond with RST packets and stop the attack. However, having an exact match of the server's SYN-ACK sequence number is highly unlikely.

At time 7, the sequence number inference is started. However, since the window was shifted to 0 in the sequence number space. Now it is necessary to shift it again in order

to allow the attacker's sequence number inference packets to pass through the window-shifting firewall. To do so, we can piggyback the sequence number inference packets along with the packets for shifting the window. For instance, an attacker can infer if the sequence number is in [0,2G) by trying 0, WIN-1, 2(WIN-1), ...up to 2G, which not only can shift the window from 0 to 2G, but also tested the [0,2G) range. Since the internal firewall has a 1G window, only 0 and 1G needs to be sent with a large TTL to go through it. All other packets can have a small TTL so that they only pass through the external firewalle. If either 0 or 1G passes through the internal firewall, then the sequence number falls in [0,2G). Otherwise, it falls in [2G,4G). One additional challenge is that the legitimate server may retransmit its "lost" response packet during the inference. As a result, the attacker has to shift the window back to a "safe" spot to prevent the retransmitted packets from passing through. For instance, one simple way is to shift the window to 0 every time after an iteration (which is what we did in our implementation). Such "position-reset" happens so fast that it is very unlikely the retransmitted packets can catch the "shifting" window.

The requirements are (C1,C2,C3,N4,N5) using packet count feedback, or (C1,C3,N2,N4,N5) using intermediate hop IPID feedback.

2) Off-site TCP injection/hijacking: Off-site attacks do not require the unprivileged malware but they are generally harder to carry out given the challenge to obtain target four-tuple.

However, **URL phishing** is a special case where an attacker can also acquire target four tuples by luring a user to visit a malicious webpage that subsequently redirects the user to a legitimate target website. A successful attack can replace the content of the target website, or if the user is previously logged in, the attacker can inject malicious Javascript to steal authentication cookies or perform actions on behalf of the user.

Here is how it works: assuming the user visited the malicious webpage, the attacker can obtain the client IP. It is also easy to obtain the legitimate website's IP given the common use of only a few load-balancing IPs. The remaining missing information is the source port number used in the next connection to the legitimate website. If the attacker can predict that, he can hijack the connection using the preemptive-SYN technique introduced earlier, i.e.,, start sending preemptive SYN packet right after the client is about to be redirected to the legitimate website (i.e., make a connection to the legitimate server). However, many browsers seem to always assign a random local port number for different web pages which makes the port prediction very difficult. To overcome the challenge, we design a simple strategy to intentionally occupy as many local ports as possible so that the next port used is selected from a much smaller pool.

Specifically, the malicious website can instruct the client to open many connections to the malicious site (or any other server) to consume a large number of local ports. In addition, the occupied port numbers tend to be contiguous according to our experiment likely due to the origination from the same Javascript. One challenge is that the OS may limit the total number of ports that an application can occupy, thus preventing the attacker from opening too many concurrent connections. Nevertheless, we found such limit can be bypassed if the established connections are immediately closed (which no longer counts towards the limit). The local port numbers are still not released since the closed connections enter the TCP TIME_WAIT state for a duration of 1–2 minutes. If an attacker can manage to open enough connections, he can easily use brute force the remaining ports by sending many preemptive SYN packets simultaneously. The rest of the attack works exactly the same as in the preemptive-SYN hijacking. Here the on-device malware is not required since the attacker already knows the target four-tuple.

Long-lived connection inference. Besides URL phishing, another type of off-site injection is to target long-lived connections. Instead of guessing the target four tuples, we discover that it is possible to "query" a network and check if a particular four-tuple is active through a single ICMP packet. If the attack targets at popular services, the server IP and port are typically known, thus the search space is reduced to only different client IP/port combinations. Since many popular services using unencrypted long-lived HTTP connections to implement PUSH services [5], the attack would basically allow remote scanning and injection of HTTP-based exploits.

This attack is possible because NAT boxes maintain state about active or in-session TCP connections, identified by four tuples. Out-of-session packets are denied access. Such behavior can leak information about existing/active sessions (similar to the reason why sequence number can be leaked). For instance, one approach is to use the intermediate hop IPID side-channel again to infer if packets with spoofed target four-tuple can go through. Note that such spoofed packets should not reach far enough to the firewall middlebox, so it does not matter what sequence number the spoofed TCP packets have. In total, the attacker has to send at least three packets (two to get the IPID before and after the spoofed probing and one is the probing packet) to query a single four-tuple, and the results may not be always reliable due to possible IPID noise.

A more efficient and reliable approach we discover is through sending a single ICMP error message (e.g., network or port unreachable) to query a four-tuple. Specifically, since many NAT boxes check the embedded TCP four tuples inside ICMP packets and allow them through only when the four tuples match existing sessions, an attacker can easily craft ICMP packets embedding target four tuples and check if they can go through. More importantly, the source IP address of the ICMP packets themselves do not

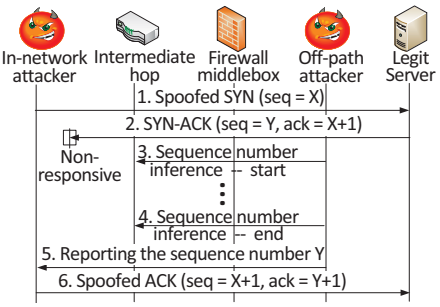

Figure 7: Establish TCP connection using spoofed IPs

have to be spoofed. This is because ICMP packets are often sent by IPs other than the two communicating parties such as a gateway. This allows the attacker to receive direct response in the form of a TTL-expired message from the intermediate hop.

3) Establish spoofed connections: The goal of this attack is to establish TCP connections to a legitimate server from an attacker using spoofed IPs. It closely resembles the traditional TCP sequence number prediction attack where an attacker can guess the sequence number of the legitimate server's SYN-ACK and establish connections using spoofed IPs. We are essentially launching the same attack, but here we are "inferring" instead of randomly "guessing" the sequence number. As elaborated in §VII-C, this attack can be a useful building block of DDoS attack or spamming where each connection has a distinct source IP, thus overcoming IP-based blocking.

The attack sequence diagram is fairly simple as shown in Figure 7. At time 1, an in-network attacker sends a spoofed SYN with an unresponsive source IP (more discussion below). At time 2, the server replies with a SYN-ACK back to the spoofed IP. However, as the spoofed IP is unresponsive, the packet does not trigger any response packet. The attack server then performs the sequence number inference during time 3 and 4. Upon completion, it reports the inferred sequence number to the in-network attacker at time 5, which in turn sends the spoofed ACK packet using the inferred sequence number to complete the TCP handshake with the victim server at time 6.

Here unresponsive IPs are either IPs that may not be currently used by any device, or they drop out-of-state TCP packets on their own (e.g., by host-based firewalls). We found that there are many such unresponsive IPs in the nation-wide cellular network that we tested. The requirements of this attack are (N1,N2,N4).

V. Attack Implementation and Experimental Results

We have implemented the complete end-to-end attacks for all three threat models. Below is our experiment setup.

Client platform. We use Android smartphones because it fits the first threat model well and it can easily connect to the nation-wide cellular network with sequence-number-checking firewalls. Other smartphones such as iPhone could also be used since it also satisfies all the client-side requirements. We implement the malware that spawns a service to run in the background and monitors new target connections to attack. To prevent from being scanning the active target connections too aggressively, the malware stops running whenever the screen is off. In fact, it can start the scanning activity only when detecting the target app is at the foreground. We tested the attacks ourselves on three different controlled Android phones (no other people is attacked by the malware) with OS versions of Android 2.2 and 2.3.4 and from three vendors (HTC, Samsung, and Motorola). The default window scaling option is 2 and 4 for Android 2.2 is 2.3.4 respectively.

Network. The experiments are conducted on an anonymized nation-wide carrier that widely deploys firewall middleboxes at the GGSN-level. The carrier satisfied all the network-side requirements (N1 to N4), which allows us to realistically test all attacks except for URL phishing. However, different GGSNs [34] may have slightly different network policy. For instance, some GGSNs prevent internal hops from replying with TTL-expired messages, thus violating requirement N2. IP spoofing is however allowed in every GGSN which enables an attacker to spoof a large range of IPs (many /16), potentially affecting many users.

Firewall. We found firewalls are deployed in all of the carrier's GGSNs. There are two main types: the first has a fixed window size (i.e., $WIN = 1G$) with window-advancing behavior, the second computes the window size based on the window scaling factor (as mentioned in §III-A) with window-shifting behavior. The first one also has a left-only window since it buffers out-of-order packets. In certain GGSNs, only the first firewall is deployed. In others, both are deployed with the second one external to the first one (which enables the hit-and-run hijacking).

Proxy setting. We found that if the proxy is enabled through the *Access Point Name* (APN) configuration [10], then the firewall middlebox is no longer visible, which we suspect is due to the specific network topology setup and is a special case. In general, a proxy is similar to NAT that essentially rewrites the external IP and port. Only the browsers seem to be affected by the proxy setting and thus attacks on mobile apps are not affected. We do not have complete data on how many phones by default have the proxy enabled, but we do know that the Motorola Android phone by default does not use the proxy.

In summary, the diversity of the network and firewall setup implies that carriers may not be fully aware of the potential impact of various network configurations on security.

A. Side-channel

So far we have introduced the two side-channels — packet counter and intermediate hop IPID, now we discuss them in more details. For the packet counter, we found that Android has all the standard and advanced Linux packet counters accessible through publicly-readable procfs. The following is a list of relevant counters identified.

/proc/net/snmp: InSegs. This is a basic counter that simply records the number of incoming TCP packets received by the OS, regardless if the packet contains error (e.g., wrong checksum). It is the most straightforward counter but may be noisy as there can be background traffic received by the client during the sequence number inference.

It turns out that it is possible to find other much less noisy counters. The idea is to leverage the mismatch in the logic of identifying error packets between the firewall middlebox and the client. For instance, we can craft packets that look erroneous to the client but perfectly legitimate to the firewall. The result is that the firewall still checks the sequence number, but when the packet reaches the client, it will be dropped and the corresponding error packet counter will be incremented. Note that these error packet counters are much less noisy because they are rarely incremented caused by naturally occurring packet corruption. Specifically, we found the following promising counters on Android.

/proc/net/netstat: InErrs. This counter should be incremented every time when, among other things, a packet with wrong checksum is received. If the firewall lets packets with wrong checksum through, then an attacker can craft such packets and make use of this counter. However, we verified that the firewall in the nation-wide cellular network already drops packets with incorrect checksum.

/proc/net/netstat: PAWSEstab. This counter is incremented when a TCP packet with an old timestamp is received. PAWS, or Protect Against Wrapped Sequences, is a mechanism that relies on timestamp to prevent old packets with wrapped-around sequence numbers from being mistakenly received, a TCP extension standardized in RFC 1323 [20]. All Android phones that we tested have this counter enabled and the firewall does not check the timestamp at all (likely due to overhead concerns). As a result, our implementation uses this counter for all on-site attacks.

For the intermediate hop IPID side-channel, we found that the noise level is quite tolerable. Specifically, the IPID of the intermediate hop increments only when the hop (e.g., router) itself is originating packets (e.g., TTL-expired messages or packets generated for routing protocols). In contrast, packets passing through the hop do not affect its IPID. That means that the IPID should not increment very often. Moreover, since the probing packets are back-to-back, the window for observing such noise is very small. In practice, we found that sending 1–4 packets per window range is usually enough to overcome the IPID noise.

B. Sequence Number Inference

Theoretically, the time to complete a binary-search-like probing is $32 \times RTT$. Assuming a cellular RTT of 200ms, the total time should be about 6.4 seconds. However, as observed in our experiments, it also takes time to send a large number of packets to cover the large sequence number space. In addition, we also add padding time during the probing to prevent packets arrive out-of-order. In practice, the binary-search-like probing can take up to 10 seconds to complete with an RTT of 200ms, which can be too long since a user may be able to notice the delay. To speed up the probing, we implement a number of optimizations.

The first optimization is that instead of inferring the exact sequence number, we can stop the inference once we know the sequence number is within a range (e.g., of 256 possible numbers). Later, it will not be difficult to simply brute force all 256 sequence numbers simultaneously. In a binary search, this can reduce $\log_2 256 = 8$ RTTs, which is significant.

The second optimization is based on the observation that the sequence number inference is heavily round-trip-bound instead of bandwidth-bound. As a result, we devise an algorithm that reduces the number of network round trips significantly. The idea is that instead of eliminating half of the sequence number space each iteration, we can eliminate $\frac{N-1}{N}$ of the search space by simultaneously probing N-1 of N equally-partitioned bins. We could send different number of packets in different bins. As an example where $N = 4$, we could send 1 packet each window in the first bin, 2 packets each window in the second bin, and 4 packets each window in the third bin. This way, an attacker could tell which bin the sequence number falls in by looking at the increment of the packet counter. We name the probing technique "N-way search". It is not hard to see the resulting number of iterations can be computed as $\log_N 4G$. For instance, if $N = 4$, $\log_N 4G = 16$, which is only half of that the original binary search needs.

At a glance, it seems that the bigger N, the better. However, we also note that by increasing N, the total number of probing packets also increases (since it requires more packets for each bin) and so is the inference time. In practice, we use a small N=2 (i.e., binary search) at the beginning few iterations, and use larger N (e.g., $N = 4$) towards the end, which turns out to work very well. When using the packet counter feedback, we found that it takes only about 4–5 seconds to complete the inference when RTT is at around 200ms.

C. On-site TCP hijacking

We next describe more details on the most critical part of each hijacking attack. We also analyze the bandwidth requirement when necessary (e.g., to reset the server) and present the experimental results in Table II measured using the Android 2.3.4 OS where we hijack m.facebook.com with a Planetlab server acting as the attack server.

1) Reset-the-server: In this attack, the most critical part is to successfully reset the server. As described before, we leverage requirement C4 which tells the attacker that the victim connection's ISN is at most 16,777,216 away (either smaller or larger) from the ISN of the attacker-initiated connection. Since RST packets with any sequence number that falls in the receive window can terminate the connection [33], the max number of required RST can be calculated as $\frac{16777216 \times 2}{server\_rwnd}$ where $server\_rwnd$ represents the server's TCP receive window size. Further,

given that the RST happens right after the server sending out SYN-ACK, $server\_rwnd$ is in fact the initial TCP receive window size denoted as $server\_init\_rwnd$. Typically, $server\_init\_rwnd$ is about three to four full TCP packets long as per TCP slow start. For instance, m.facebook.com uses 4380, twitter.com uses 5840, and the corresponding number of required RST packets is 7661 and 5746 respectively. However, different websites can have very different values. We found chase.com uses 32805 which is almost a magnitude larger. In general, the larger the $server\_init\_rwnd$, the fewer packets required.

Moreover, to successfully reset the server in time, all RST packets have to be delivered between time 3 and 5 as shown in Figure 4. If they arrive after time 5, the server may already respond to the client's request. Thus, the valid time window for reset is basically a round trip time between the client and the server. The bandwidth requirement is then computed as $\frac{16777216\times 2}{\frac{server\_init\_rwnd}{RTT}} \times 40bytes \times 8bits$. In our experiment in cellular networks where $RTT = 200ms$, it will be $327Kbps - 12Mbps$ (as shown in Table II), depending on the $server\_init\_rwnd$ values mentioned above. When RTT is smaller (as on the Internet), the bandwidth requirement will increase proportionally. This is another reason why cellular devices are particularly vulnerable and easy to attack. Although the bandwidth requirement may seem high, it is important to note that bandwidth resource is becoming more abundant and cheaper. For instance, the uplink bandwidth of a standard home Comcast network can be up to 4.2Mbps (tested in our home). The bandwidth requirement can even be distributed across a number of bots. Moreover, the bandwidth requirement is not a hard requirement and the attack can be attempted multiple times. For instance, it will be good enough to use TCP hijacking to steal a user's password just once. In our experiment, we use a Planetlab server acting as the attack server to reset m.facebook.com. We are not certain about the exact bandwidth, but the reset success rate is quite good according to our experiment.

As shown in Table II, the success rate of reset-the-server hijacking is 65% after 20 experiments with 7 failures in total. 5 of them are caused by the RST race condition failure. Other 2 are due to sequence number inference failures (e.g., packet loss). As we can see, the success rate is high enough to cause real damage. It takes 4 to 5 seconds to complete the inference when measured with packet count feedback. It takes only 2 seconds using intermediate hop feedback as the probing does not go through the cellular link. The downside is that the latter may not always be available. Nevertheless, since we observe that it takes more than 10 seconds to tear down a connection after several rounds of retransmission, the inference time is definitely short enough.

2) Preemptive-SYN: During implementation, we found one interesting detail about the intermediate hop feedback where its TTL-expired message can inadvertently terminate the client-side connection. It happens only when a TTL-expired message embedding a TCP header with a sequence number matching the original SYN's sequence number (similar to the hit-and-run hijacking case). Our optimization on the sequence number inference should already alleviate the problem since we stop inference much earlier so that it is unlikely a spoofed packet has the same sequence number as in the original SYN.

Table II: TCP hijacking bandwidth requirements and results

	Reset-the-server	Preemptive-SYN	Hit-and-run
BW required	0.3 – 12Mbps	None	6.6 – 26Mbps
BW factor	server_init_rwnd, RTT	None	WIN, RTT
Success rate	65%	65%	85%
Inference time	4–5s	6–7s	8–9s

Note that there is no bandwidth requirement for this attack as long as requirement S2 is satisfied. Interestingly, according to Table II, the success rate is still measured to be 65% after 20 experiments. However, out of the 7 failed cases, 6 are due to the sequence number inference (likely caused by the noise in IPID side-channel). 1 of them seems to be due to a load balancing change that causes the connection to the attack server to go through a different intermediate hop. However, we observe that this happens very rarely. In terms of the inference time, it takes about 6 to 7 seconds, slightly longer than the Reset-the-server attack, due to the need to send more packets per window to overcome the noise in the IPID side-channel.

3) Hit-and-run: The critical part of this attack is to shift the window in time at the very beginning to prevent legitimate server's packets from going through the firewall. The number of packets required is computed as $\frac{4G}{WIN-1}$ since one packet is sent per $WIN - 1$. Depending on the window scaling factor, WIN is 256K and 1M respectively for the two Android OSes. The bandwidth requirement is basically $\frac{\frac{4G}{WIN-1}}{RTT} \times 40bytes \times 8bits$ or 26Mbps and 6.6Mbps if we plug in the two WIN values (as shown in Table II). One thing to note is that the window scaling factor is incremented every time a new Android version is pushed out, presumably to take advantage of the increasing cellular network bandwidth. This indicates that future attacks will have even lower bandwidth requirement.

As shown in Table II, the success rate is 85% with only 3 failed cases caused by the inference failure. No failure is observed for the initial window shifting likely due to the lower bandwidth requirement with the window scaling factor of 4. Note that we need to shift the window back and forth in each iteration, which means more packets are sent and packet loss is thus more likely. For the same reason, the inference time is a little longer.

D. Off-site TCP injection

We were not able to implement the **URL phishing** attack on the nation-wide network, which is the only attack we did not implement. The reason is that when NAT is deployed, the attack requires knowing the client's private IP in order to conduct the sequence number inference from the client's

network (same as preemptive-SYN). However, without on-site malware, it is difficult to obtain the device IP (i.e., private IP) through mobile browsers. The only way to get device IP seems to be through Java applet which is not supported on mobile browsers. We have confirmed neither Javascript nor Flash can do so. Note that this attack is feasible for cellular carriers using public IP addresses for their mobile devices (there are in fact many such carriers according to a recent study [32]).

We did implement the **long-lived connection inference** using a single ICMP packet and run a small-scale experiment on the nation-wide carrier to measure the number of cellular IPs actively using Android's push notification service. We pick a particular push server IP 74.125.65.188 and port 5228 (push service port), and choose an entire /16 cellular IPs to probe. For each IP, we enumerate every port within the default local port range for Android: 32768 – 61000. To avoid probing too aggressively, our experiments conservatively rate limit the probing to 6 seconds per IP. Interestingly, using the single-ICMP-packet probing, we found that about 7.8% of the IPs have a connection with the server. That means it is fairly easy to find popular services to attack. Even through the connections are encrypted, it is still possible to carry out connection reset attacks. In fact, this approach is much more efficient than the traditional reset attack where combinations of client port number and sequence number need to be enumerated.

E. Establish spoofed connections

We implement the attack mostly as described in §IV-B3. The only difference is that instead of spoofing a single IP, we spoof as many IPs (for different connections) to a controlled target server as possible. Specifically, we try to spoof all IPs inside a /16 IP range in the nation-wide carrier.

For each IP that we want to spoof, we need to first test if the IP is responsive. To do so, we first send a SYN packet with a spoofed IP from the attack phone inside the cellular network to our attack server which responds with a legitimate SYN-ACK back. If the spoofed IP is responsive, a RST will be generated. Otherwise, we consider the IP to be unresponsive. For any unresponsive IP, we send a second spoofed SYN, this time, destined to the victim server (i.e., a controlled lab server). The rest of the work is to simply conduct the sequence number inference from the attack server using the intermediate hop feedback so that we can spoof a correct ACK packet to complete the connection.

Ideally an attacker can simultaneously spoof many IPs. However, we found that there is only a single shared responsive intermediate hop where all the TTL-expired messages essentially share a single IPID counter. If we parallelize the process, different experiments probing to the same intermediate hop can interfere with each other. Consequently, we can only pipeline the process as much as possible to make sure there is always one sequence number inference procedure probing to the intermediate hop.

Through our experiments, we found that there are 80% of IPs are unresponsive, which means that there are plenty of IPs an attacker can make use of to establish spoofed connections. We found that we can make about 0.6 successful connection per second on average with more than 90% success rate (the failed cases are mostly due to sequence number inference error).

VI. Vulnerable Networks

To understand the susceptibility of the existing networks to the described attacks, in this section, we report the measurement results of firewall implementations and availability of responsive intermediate hop, through a deployed mobile application (referred to as *MobileApp*) on the Android market (the malware described earlier was not on the market). The MobileApp measures the network performance and policy and reports the results to users so that they have incentives to run our app. The data are collected between Apr 25th, 2011 and Oct 17th, 2011 over 149 carriers uniquely identified by their Mobile Country Code (MCC) and Mobile Network Code (MNC).

A. Firewall implementation types

Methodology. We focus on the three firewall implementation properties described in §III-A. The three properties are selected based on experiences with the firewalls encountered in real carrier networks as well as a number of trial-and-errors on the earlier deployment of our MobileApp.

To infer the **window size**, we try the following WIN values in order: 2G, 128M, 16M, 1M, 512K, 256K, 64K. Note that testing exhaustively all possible window size values is too time-consuming as a long timeout (i.e., 4 seconds) is needed for each probing packet to account for long cellular RTTs. Specifically, for each WIN value, our MobileApp server test sequence numbers *X-WIN+2* and *X+WIN-2* to check if they can trigger any response. X is the next expected server-side sequence number. The adjustment by 2 is to accommodate a slightly smaller window implementation from the common values. The reverse ordering by window size is to finish the test more quickly if there is no sequence number checking (i.e., WIN=2G).

To test the **left-only/right-only window** behavior, we always try the left window and then the right one (to be consistent). If the left window probing packet is allowed but not the right one, we conclude it is left-only window. Similarly, we can discover right-only window firewalls. Additionally, we have to eliminate the window-shifting case where the left-window packet can shift the window to the left so that the right window packet may be falsely considered as "out-of-window". Such cases can be detected by the test described next.

To test if the firewall has **window-shifting** behavior, the basic procedure is as follows: once a left-window packet with sequence number *X-WIN+2* is allowed by the firewall, we try to shift it further left by (WIN-1) twice. If both attempts of shifting succeed, we try the sequence number

Table III: Sequence-number-checking firewall types

Window Size	Left/Right	Window Moving	# of Carriers
64K	left-only	window-advancing	6
fixed>128M	left&right	window-advancing	5
window scaling	left&right	window-advancing	7
window scaling	left&right	window-shifting	17
window scaling	left-only	window-advancing	10
-2G	left-only	unknown	2

X-WIN+2 again. If the window is indeed shiftable, its center is already shifted left by 2(WIN-1), making *X-WIN+2* out-of-window (and the packet will be dropped). There are two corner cases that need to be considered to ensure the validity of the results. The first one is that since we do not cover all possible window sizes, the inferred window size WIN may be an under-estimate of the actual window size. We address this explicitly by shifting the window far enough beyond an over-estimate of the actual window size. The second one has to do with resetting the window position to its original value after left window test is done before the right window test.

Results. Overall, out of the 149 carriers, we found 47 carriers that deploy sequence-number-checking firewalls with at least two completed supporting experiments. 10 other carriers were found to be suspicious but with only one experiment, thus are excluded due to possible errors caused by packet loss. If we consider only the 47 carriers, 31.5% of the carriers are subject to the sequence number inference attack. The nation-wide network we tested is excluded from this analysis because it is somewhat a special case with two different firewalls deployed. We did not look for a similar two-firewall setup in the measurement and thus cannot conclude the number of other carriers with such two-firewall setup. In essence, our experiments test the combined effects of all sequence-number-checking firewalls.

A detailed breakdown of the measured firewall implementations is shown in Table III.

Window size. We can observe three main window sizes: 1). 64K — some legacy firewalls only support this value (window scaling is not supported), 2). window scaling — where the size is calculated based on the window scaling factor, 3). fixed > 128M — could be 1G as found in the nation-wide cellular network. There is one last window size listed as "-2G" which means that the left window is wide open, but no packets are allowed for the right window.

Left-only or right-only window. Interestingly, we discover that many networks have left-only window firewalls. For the nation-wide carrier, it is because the internal firewall buffers out-of-order packets as discussed before. However, we found this may not be the case for other carriers. Upon a closer inspection, we realize that some firewalls actually have an even smaller-than-64K right window set based on the initial receive window size (sometimes below 8K) carried in the client-side SYN (instead of based on the window-scaling factor). This behavior matches the ideal firewall that dynamically adjust the window size based on the currently advertised receive window. On the other hand, the left window is still kept to be fixed in case old packets are lost and retransmitted. Since we did not test window sizes smaller than 64K, it is possible that some of the left-only window carriers can in fact be left&right. Regardless, such minor variations do not impact the attack as the window size can be obtained offline.

Window moving criteria. We found 17 carriers to have shiftable windows and all with left&right windows, making it difficult to infer the exact sequence number but still susceptible to attacks. The other majority of 30 carriers, however, allow the exact sequence number to be inferred.

B. Intermediate hop feedback

Methodology. We devise the following probing technique to infer if any intermediate hop is responsive: from the previous experiments we can gather an in-window and an out-of-window sequence number. We conduct two TCP traceroutes with those two sequence numbers respectively. If there is any hop that responds to the first traceroute (with in-window sequence number) but not to the second one, we flag such hop. Additionally, we send two traceroutes (ICMP error messages) embedding a correct four-tuple and a wrong one (with a modified port number). If any hop responds to the correct one but not the incorrect one, we consider the single ICMP packet probing as possible.

Results. Out of all the 47 carriers that have sequence-number-checking firewalls, 24 carriers have responsive intermediate hops that reply with TTL-expired ICMP packets. 8 carriers have NAT that allow single ICMP packet probing to infer active four tuples.

VII. Vulnerable Applications

The TCP sequence number inference attack opens up a whole new set of attack venues. It breaks the common assumption that communication is relatively safe on encrypted/protected WiFi or cellular networks that encrypt the wireless traffic. In fact, since our attack does not rely on sniffing traffic, it works regardless of the access technology as long as no application-layer protection is enabled. In this section, we illustrate the broad impact of the attack by a mere glimpse at a number of impacted applications.

A. Web-based attack

Facebook/Twitter: We found that the login pages for both desktop and mobile browser are not using SSL. They are subject to phishing attack where the login page can be replaced. Further, when users are logged in, webpages by default are not SSL-enabled (unless turned on in the account settings). It allows Javascript injection which simply sends a HTTP post request to perform actions on behalf of the users such as posting a message or following other users. Both Facebook and Twitter servers have host-based stateful firewalls that satisfy requirement S1, which enables Reset-the-server hijacking. In both cases, gaining access to users' social networking account is a huge privacy breach.

Banking: Similar to a previous study [17], we survey 68 banking websites from a keyword "bank" search from

Google, 4 of which are found to have non-SSL login page. There is one other website which uses SSL in most pages but not one specific account query page which also contains a login form. Also, one website has a login helper program download link in HTTP that allows the binary to be replaced. In all cases, successful attacks can cause direct financial loss. We also verified that all bank servers deploy host-based stateful firewalls which satisfy requirement S1.

B. Application-based attack

Facebook app: The latest version of the Facebook app as of this writing was updated on October 5, 2011. We found that it is impossible to replace the login page as it is part of the built-in UI (i.e., not fetched over the network). However, we do find two sensitive connections not using SSL. Even though we did not test our attacks specifically on them, it is quite obvious that they are subject to our attacks.

- The main page (e.g., news feed) is fetched through HTTP (html/text) which is subject to tampering.
- A critical Javascript is fetched through HTTP. An attacker can inject malicious Javascripts to perform actions on behalf of the user just as the web-based attack.

Windows Live Messenger app: The protocol [11] is in plaintext without encryption in most client implementations, which allows an attacker to inject arbitrary messages while a user is logged in. The protocol does not require any nonce carried in the server's notification of incoming messages. We verified that an attacker can indeed succeed in posting malicious links (e.g., to spread virus or spam).

Stocks app: The number one stocks app on the Android market uses Google finance through HTTP to display stock prices. It allows an attacker to inject misleading prices which can cause potential financial loss. Moreover, we verified that instead of blindly injecting HTTP responses to a request (to guess for a particular stock), an attacker can inject "HTTP 301 – Moved Permanently" message to redirect the request to its own server which can read which stocks the app is requesting and send the corresponding fake prices. Unlike a browser with an address bar, such redirection happens transparently.

Advertisement: We tested that advertisements provided through AdMob are fetched over HTTP. An attacker can thus replace the original advertisement with his own to gain revenue. Note that this attack is not intrusive and can be carried out repeatedly to achieve long-term benefits, as long as the malware is kept on the device.

C. Server-side attack

The "Establish spoofed connections" attack described in §IV-B3 allows an attacker to establish connections with a target server using many spoofed IPs. It can be applied in the following scenarios:

Mail server spamming. Using spoofed IPs generally can increase the probability that a spam email is accepted by the mail server since IP-based spam blacklists are unlikely to catch all bad IPs at once. Without IP spoofing, an IP repeatedly sending spam is likely blacklisted very quickly. We tested that we can successfully deliver emails by simply sending a spoofed data packet (with SMTP commands) to our departmental mail server and acknowledging server's response (via a number of spoofed ACK packets).

DoS of servers. Web server and other public-facing servers are subject to DoS attacks due to a large number of spoofed connections. Note that it is different from SYN flooding in that the connections are actually established, so SYN-cookie-based defense is not effective. We experimented the attack against our own sshd server running on Ubuntu 11.04 (server kernel build) and found that the 0.6 conn/s rate is in fact enough to cause new legitimate ssh connections rejected sporadically when the number of active connections reach a certain limit. We suspect it is due to a security kernel counter-measure triggered to block new connections, which also causes the collateral damage.

VIII. Discussion and Conclusion

After constructing a diverse set of attacks, we explore what actually went wrong and how we can fundamentally correct them. We discuss the following four aspects.

Firewall design. It is interesting and surprising to realize that the more checks the firewall performs, the more information it can leak. For instance, if it checks the four-tuple and allows only packets belonging to an existing session to go through, then an attacker can infer which four tuples are active. If it checks sequence number, then the sequence number inference attack becomes possible. Similarly, if a firewall checks acknowledgment number according to RFC 793 [25] where half of the acknowledge number space is considered valid (as is in the latest Linux TCP stack implementation), then it may allow an attacker to additionally infer the appropriate acknowledgment number, which can help preemptive-SYN attack eliminate the requirement of IP spoofing in the client's network. Our study suggests that firewall middlebox designs should be carefully evaluated on potential leakage of sensitive network state.

Side-channels. We have summarized two side-channels that serve as feedbacks of the sequence number inference, without which the attack would not be possible. They are intermediate hop IPID and host packet counter. We study whether they are fundamentally difficult to eliminate. For IPID, the answer is negative, as many host OSes such as Linux already use randomized IPIDs. However, for packet counter, it seems that such aggregated information is always available on most OSes and considered harmless. Our study suggests that such information can be abused. One way to mitigate the problem is to add a permission requirement to read such packet counters. However, many users may simply grant the permission. The other important aspect is that the firewall does not check the TCP timestamp option (likely due to overhead concerns) which allows an attacker to leverage the less noisy *PAWSEstab* counter. It suggests a dilemma of the firewall design – it has to tradeoff between performance and the completeness of checks.

Other side-effects. We discover several other notable side-effects of the current host TCP implementation or setup. For instance, the coarse-grained ISN predictability is a byproduct of the Linux TCP implementation. Also, the fact that a server can be kept silent after being reset is caused by the side-effect of the server's host-based firewalls. Interestingly, such implementations and setups are well intended, yet they in fact facilitate the attacks. In the end, we do not think they are the culprit of the problem because even if these two side-effects are eliminated, it prevents only the Reset-the-server hijacking.

HTTPS-only world. In general, SSL should be able to defeat most attacks. Hopefully one of the results of our study is to help push the HTTPS-only world. We do note that even if SSL is employed by the websites, there is a special case where an attack may still succeed. Specifically, when a user types in a URL such as www.chase.com, the default browser behavior is to initiate a normal HTTP request first unless the user specifically types in https://www.chase.com. It is generally the server that subsequently redirects the browser to the https site via a "301 – Moved Permanently" HTTP response. Instead of redirecting the browser, an attacker can simply respond directly with a phishing page to the initial HTTP request. In this case, the only difference is that the browser will not show the https icon. However, average users may not notice.

In conclusion, we are the first to report the TCP sequence number inference attack using state kept on middleboxes and attacks built on it. We demonstrate that many networks and applications are affected today. We also provide insights on why they occur and how they can be mitigated.

REFERENCES

[1] CERT Advisory CA-1995-01 IP Spoofing Attacks and Hijacked Terminal Connections. http://www.cert.org/advisories/CA-1995-01.html, Retrieved on 03/04/2012.

[2] CERT Advisory CA-2001-09 Statistical Weaknesses in TCP/IP Initial Sequence Numbers. http://www.cert.org/advisories/CA-2001-09.html, Retrieved on 03/04/2012.

[3] Check Point – What's New for FireWall-1/TCP Sequence Checking. http://www.checkpoint.com/ngupgrade/whatsnew/products/features/tcpseqcheck.html, Retrieved on 03/04/2012.

[4] Cisco Security Advisory: Cisco Secure PIX Firewall TCP Reset Vulnerability. http://www.cisco.com/en/US/products/products_security_advisory09186a00800b1397.shtml, Retrieved on 03/04/2012.

[5] Comet (programming). http://en.wikipedia.org/wiki/Comet_(programming), Retrieved on 03/04/2012.

[6] Linux Blind TCP Spoofing Vulnerability. http://www.securityfocus.com/bid/580/info, Retrieved on 03/04/2012.

[7] Linux: TCP Random Initial Sequence Numbers. http://kerneltrap.org/node/4654, Retrieved on 03/04/2012.

[8] Stateful Firewall and Masquerading on Linux. http://www.puschitz.com/FirewallAndRouters.shtml, Retrieved on 03/04/2012.

[9] TCP hijacking video demo. http://youtu.be/T65lQtgUJ2Y, Retrieved on 03/04/2012.

[10] Access Point Name. http://en.wikipedia.org/wiki/Access_Point_Name, Retrived on 03/04/2012.

[11] MSN Messenger Protocol. http://www.hypothetic.org/docs/msn/, Retrived on 03/04/2012.

[12] S. M. Bellovin. A Look Back at "Security Problems in the TCP/IP Protocol Suite". In *ACSAC*, 2004.

[13] R. Beverly, A. Berger, Y. Hyun, and k claffy. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In *Proc. ACM SIGCOMM IMC*, 2009.

[14] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: a reality today, a challenge tomorrow. In *Proc. of IEEE Security and Privacy*, 2010.

[15] Cisco. Cisco ASA 5500 Series Configuration Guide using the CLI, 8.2. http://www.cisco.com/en/US/docs/security/asa/asa82/configuration/guide/conns_tcpnorm.html, Retrieved on 03/04/2012.

[16] R. Ensafi, J. C. Park, D. Kapur, and J. R. Crandall. Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks using Model Checking. In *Proc. of USENIX Security Symposium*, 2010.

[17] L. Falk, A. Prakash, and K. Borders. Analyzing websites for user-visible security design flaws. In *Proc. of Usable privacy and security*, 2008.

[18] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: attacks and defenses. In *Proc. of USENIX Security Symposium*, 2011.

[19] F. Gont and S. Bellovin. Defending Against Sequence Number Attacks. RFC 6528, 2012.

[20] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, 1992.

[21] Juniper. Stateful Inspection Firewalls. http://www.abchost.sk/download/204-4/juniper-stateful-inspection-firewall.pdf, Retrieved on 03/04/2012.

[22] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen. Analysis of Vulnerabilities in Internet Firewalls. In *"Computers & Security"*, 2003.

[23] G. LEECH, P. RAYSON, and A. WILSON. Procfs Analysis. http://www.nsa.gov/research/_files/selinux/papers/slinux/node57.shtml, Retrieved on 03/04/2012.

[24] lkm. Blind TCP/IP hijacking is still alive. In *Phrack Magazine, issue 64*, 2007.

[25] J. Postel. TRANSMISSION CONTROL PROTOCOL. RFC 793, 1981.

[26] Z. Qian, Z. M. Mao, Y. Xie, and F. Yu. Investigation of Triangular Spamming: A Stealthy and Efficient Spamming Technique. In *Proc. of IEEE Security and Privacy*, 2010.

[27] A. Ramaiah, R. Stewart, and M. Dalal. Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961, 2010.

[28] E. S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT Behavioral Requirements for TCP. RFC 5382, 2008.

[29] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS*, 2011.

[30] D. X. Song, D. Wagner, and X. Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *Proc. of USENIX Security Symposium*, 2001.

[31] J. Touch. Defending TCP Against Spoofing Attacks. RFC 4953, 2007.

[32] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An Untold Stody of Middleboxes in Cellular Networks. In *SIGCOMM*, 2011.

[33] P. A. Watson. Slipping in the Window: TCP Reset Attacks. In *CanSecWest*, 2004.

[34] Q. Xu, J. Huang, Z. Wang, F. Qian, A. Gerber, and Z. M. Mao. Cellular Data Network Infrastructure Characterization and Implication on Mobile Content Placement. In *Proc. ACM SIGMETRICS*, 2011.

[35] K. Zhang and X. Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *Proc. of USENIX Security Symposium*, 2009.