Collaborative TCP Sequence Number Inference Attack — How to Crack Sequence Number Under A Second

Zhiyun Qian
Department of EECS
University of Michigan
2260 Hayward Street
Ann Arbor, MI, USA
zhiyunq@umich.edu

Z. Morley Mao
Department of EECS
University of Michigan
2260 Hayward Street
Ann Arbor, MI, USA
zmao@umich.edu

Yinglian Xie
Microsoft Research
Silicon Valley
1288 Pear Avenue
Mountain View, CA, USA
yxie@microsoft.com

ABSTRACT

In this study, we discover a new class of unknown side channels — "sequence-number-dependent" host packet counters — that exist in Linux/Android and BSD/Mac OS to enable TCP sequence number inference attacks. It allows a piece of unprivileged on-device malware to collaborate with an off-path attacker to infer the TCP sequence numbers used between a client and a server, leading to TCP injection and hijacking attacks. We show that the inference takes, in common cases, under a second to complete and is quick enough for attackers to inject malicious Javascripts into live Facebook sessions and to perform malicious actions on behalf of a victim user. Since supporting unprivileged access to global packet counters is an intentional design choice, we believe our findings provide important lessons and offer insights on future system and network design.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Internet (e.g., TCP/IP)*

General Terms

Security, Experimentation

Keywords

TCP hijacking, TCP sequence number, Network packet counters

1. INTRODUCTION

Since TCP was not originally designed for security, for years it has been patched to address various security holes, among which the randomization of TCP's initial sequence number (ISN), introduced in RFC1948 [7] in 1996 was an important one. It was proposed to guard against off-path spoofing attacks attempting to inject packets with forged source addresses (for data injection or reset attacks) [7, 8]. ISN randomization prevents easy prediction of sequence numbers; thus arbitrarily injected packets are likely to be discarded at the receiver due to invalid sequence numbers.

The patch has largely rendered most sequence-number-guessing-based attacks very hard to succeed. However, in recent years, new attacks are reported. In 2007, a study reported in Phrack magazine [1] has revisited the problem and claimed that TCP sequence number can still be inferred based on how a host treats in-window and out-of-window incoming packets. However, the scope of this attack is rather limited, primarily targeting long-lived connections with a rather low success rate (as shown in §3.3). In 2012, researchers have discovered that the sequence number inference attack can be more generally applicable, impacting even short-lived HTTP connections [26]. However, this attack heavily relies on the presence of sequence-number-checking firewall middleboxes deployed in the network. Specifically, the idea is that if a packet has passed the sequence-number-checking firewall, then it implies that the sequence number of the packet is considered within a legitimate window.

Our work generalizes these attacks by eliminating the strong requirements imposed on them to enable a broader class of attacks. Specifically, we make the following key contributions:

- Building on the threat model presented in the recent work [26], we generalize the sequence number inference attack by demonstrating that it can be reliably carried out without the help of the firewall middleboxes. Our work provides further evidence that relying on TCP sequence number for security is not an option.
- Distinct from the "error counters" (*e.g.,* packets rejected due to old timestamps) used in the previous study [26], which serves only as an indication of whether a packet is allowed to pass through the sequence-number-checking firewall, we discover a new class of packet counters — "sequence-number-dependent" counters in Linux/Android (1 counter) and BSD/Mac OS (8 counters) — that can directly leak sequence numbers without requiring the presence of firewall middleboxes, thereby elevating the danger of TCP injection and hijacking attacks.
- We are able to complete the sequence number inference within 4–5 round trips, which is much faster than the one previously proposed [26], due to both the property of newly discovered "sequence-number-dependent" counters as well as

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

a more efficient probing scheme. For instance, we show that it takes as little as 50ms to complete the inference, two orders of magnitude faster than previous method. It can even eliminate the need of conducting additional TCP hijacking attacks required before, resulting in a much higher attack success rate (See §5.1).

As a proof-of-concept demonstration, we show that our attack allows a piece of unprivileged malware on Android smartphones to hijack a Facebook connection, replacing the login page, or injecting malicious Javascripts to post new status on behalf of the victim user, or performing other actions. All these attacks (except the TCP hijacking attack) work on the latest Linux kernel. TCP hijacking requires kernel versions earlier than 3.0.2, which are still the case for the majority of the Android phones. Besides Android/Linux, we also demonstrate that the attack is applicable to the latest BSD/Mac OS. We believe our work presents an important message that today's systems still expose too much shared state with poor isolation.

The rest of the paper is organized as follows: §2 thoroughly describes the related work. §3 explains how to infer TCP sequence number (including both previous study and our discovery). §4 covers how we can leverage the sequence number inference as a building block to conduct a number of TCP attacks. §5 shows several cases studies demonstrating the impact on specific applications. §6 discusses why the problem occurred and concludes.

2. RELATED WORK

TCP sequence number inference attack. By far, there are only a few reported TCP sequence number inference attacks. The first one goes back to 1999 where a TCP stack bug causes the kernel to silently drop the third packet during "three-way handshake" if the ACK number is smaller than the expected ACK number, and sends a reset otherwise [4]. This allows an attacker to send spoofed ACK packets and infer the correct ACK number. This minor bug was quickly fixed. Besides it, there are three other closely related studies. One of them is described in the Phrack magazine [1] that uses the IPID side channel on Windows to infer both the server-side and the client-side TCP sequence numbers. According to our empirical results, such attack is theoretically possible but very hard to carry out. It can succeed under rather limited conditions due to a large number of packets required as well as the noisy side-channel that is leveraged. Following the same direction, a more recent work [20] improves the reliability of the attack by requiring certain control on the client (e.g., javascript through browser), yet it still relies on the noisy IPID side channel available on Windows only.

A closely related recent work [26] discusses how sequence-number-checking firewall middleboxes can leak the TCP sequence number state stored on the firewall. The idea is that if a packet has passed the sequence-number-checking firewall, it implies that the sequence number of the packet is considered within a legitimate window. Otherwise, it implies that the packet has an out-of-window sequence number. As a result, if an attacker can observe whether a spoofed packet has passed the firewall, he will be able to know if a guessed sequence number is correct. To do so, an attacker can intentionally craft a spoofed packet with certain errors (*e.g.,* old timestamp) and then leverage the error packet counters on the host (*e.g.,* packets rejected due to old timestamps) to tell if a spoofed packet has passed the firewall and reached the end-host. In our work, we make a major improvement by eliminating the requirement of firewall middleboxes altogether with the help of a class of "sequence-number-dependent" packet counters that we discover. In addition to a more general attack model, we also show significant improvements on success rate and attack speed with much lower network resource requirements.

Other TCP-sequence-number-related attacks. (1) TCP sequence number prediction attack. Different from TCP sequence number inference attack, the prediction attack relies on the non-randomness of TCP Initial Sequence Numbers (ISN) [25, 2]. To defend the attack, RFC1948 [7] standardizes the ISN randomization behavior such that different connections should generate random sequence numbers independently. (2) Blind TCP RST attack. Due to the fact that a connection will be reset as long as the sequence number of the reset (RST) packet falls in the current receive window, in a long-lived connection (*e.g.,* a BGP session), an attacker can brute force all possible target connections and sequence number ranges [8, 32] to cause denial of service.

Smartphone-based attacks. There have been a number of attacks against smartphones, many of which focus on leaking sensitive information [15, 16, 28]. In addition, there is a class of privilege escalation attacks on Android [17, 19, 14], but they are limited to gaining permissions that typically cannot affect the behavior of other applications. For instance, one application may gain the permission of reading the contact list or GPS location through other colluding or vulnerable apps, but it cannot tamper with the TCP connection of other applications given the OS's sandboxing mechanisms. Our study demonstrates that injection and hijacking of TCP connections can be achieved without requiring any special permission other than the permission to access the Internet.

Side-channel information leakage. A wide range of side channels have been investigated before: CPU, power, shared memory/files, and even electromagnetic waves, *etc.* Researchers have found that it is possible to construct various attacks, *e.g.,* to infer keystrokes through many side channels [30, 33, 29, 13, 18]. It is especially interesting to see how smartphones can allow malware to infer sensitive information through on-board sensors (which can also be considered as side-channels). For instance, Soundcomber [28] uses the audio sensor to record credit card numbers entered through keypad. In our work, we also rely on side-channels on the host, but the attacks infer information at the network-layer.

3. TCP SEQUENCE NUMBER INFERENCE ATTACK

The ultimate goal of the attack is to inject malicious TCP payload into apps running on a victim smartphone or client device. It is achieved by a piece of unprivileged on-device malware collaborating with an off-path attacker on the Internet. The main implication of this attack is that websites that do not use HTTPS will be vulnerable to various attacks such as phishing and Javascript injection because the HTTP response can be potentially replaced. Even if HTTPS is used, they are still vulnerable to connection reset attacks as we show that the sequence number can be quickly inferred in under a second.

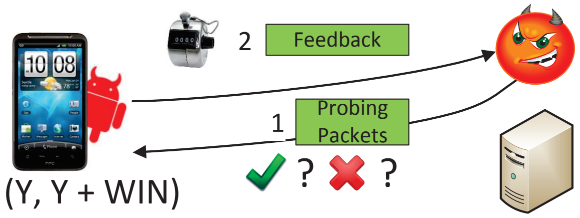


Figure 1: Threat model

3.1 Threat Model

The threat model is illustrated in Figure 1. There are four main entities: (1) The victim smartphone and a target application, constituting the attack target. (2) The legitimate server, which talks to the victim smartphone using an unencrypted application-layer protocol (*e.g.,* HTTP). The server can also become the attack target (see §5). (3) The on-device malware, which is unprivileged and cannot tamper with other apps directly. (4) The off-path attacker, who is capable of spoofing the IP address of the legitimate server and the victim smartphone. The off-path attacker and the malware collaborate to infer the correct TCP sequence number of the connection established between the target app and the legitimate server. Note that different from the threat model described in the recent study [26], this attack does not require the network firewall middlebox, making our attack model much more general.

At a high level, as shown in Figure 1, the off-path attacker needs two pieces of information: (1) the four tuples of a target connection, *i.e.,* source/destination IP addresses and source/destination port numbers and (2) the correct sequence number. The on-device malware can easily identify the current active connections (*e.g.,* through `netstat`), but it does not know the sequence number in use. In this attack model, the off-path attacker can send probe packets using the target four tuples with different guessed sequence numbers. The unprivileged malware then uses certain side-channels to provide feedback on whether the guessed sequence numbers are correct. Guided by the feedback, the off-path attacker can then adjust the sequence numbers to narrow down the correct sequence number.

3.2 Packet Counter Side Channels

In this study, we look at a particular type of side channel, packet counters, that can potentially provide indirect feedback on whether a guessed sequence number is correct. In Linux, the *procfs* [24] exposes aggregated statistics on the number of incoming/outgoing TCP packets, with certain properties (*e.g.,* wrong checksums). Alternatively, "`netstat -s`" exposes a similar set of information on all major OSes including Microsoft Windows, Linux, BSD, Mac OS and smartphone OSes like Android and iOS. Since such counters are aggregated over the entire system, they are generally considered safe and thus accessible to any user or program without requiring special permissions. The IPID side-channel [27] can be considered as a special form of packet counter that records the total number of outgoing packets since it is incremented for every outgoing packet. However, such side-channel is nowadays only available on Microsoft Windows and is typically very noisy.

Even though it is generally perceived safe, we show that an attacker can correlate the packet counter update with how the TCP stack treats a spoofed probing packet with a guessed sequence number. Different from the recent work [26] that uses certain "error counters" as an indication of whether a spoofed packet has passed the sequence-number-checking firewall middlebox, our hypothesis is that the TCP stack may increment certain counters when the guessed sequence number is wrong and remain the same when it is correct, or vice versa. Such counters can directly leak sequence numbers without the help of the firewall middlebox and are thus named "sequence-number-dependent counters" (details in §3.4 and §3.5). To investigate such a possibility, we first need to understand how TCP stack handles an incoming TCP packet and how various counters are incremented during the process.

3.3 TCP Incoming Packet Validation

In this section, we provide background on how a standard TCP stack validates an incoming packet that belongs to an established TCP connection. Specifically, we use the source code of the latest Linux kernel 3.2.6 (at the time of writing) as reference to extract the steps taken and checks performed on an incoming packet (the packet validation logic is stable since 2.6.28). Based on the source code, we summarize "sequence-number-dependent" side-channels on Linux and extend it to BSD/Mac OS.

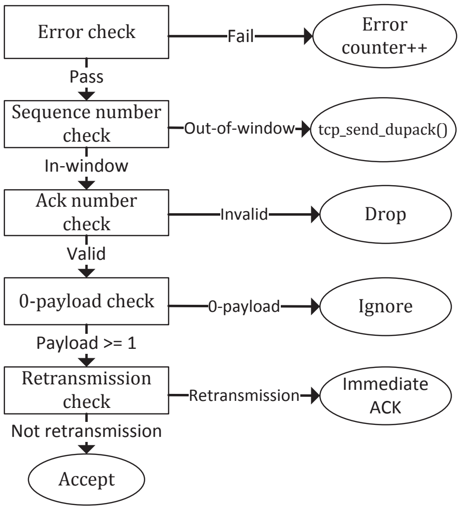


Figure 2: Incoming packet validation logic

As we can see in Figure 2, there exist five main checks performed by Linux TCP stack based on the corresponding source code as well as our controlled experiments. These checks are performed for any incoming TCP packet that is deemed to belong to an established connection based on the four tuples:

(1). Error check is for the purpose of dropping invalid packets early on. There are a number of specific error checks: 1) MD5 option check, 2) timestamp option check, 3) packet length and checksum check. Each has a corresponding error packet counter. If a specific error is caught, the corresponding host packet counter is incremented and the packet is not inspected further. Otherwise, it goes to the next step.

(2). Sequence number check is the most relevant check. It basically checks if a packet is in window by making sure that the ending sequence number of the incoming packet is larger than or equal to X, and the starting sequence number is smaller than or equal to X+*rcv_win*, where X is the next expected sequence number and *rcv_win* is the current receive window size. If the sequence number is out of window, it triggers an immediate duplicate acknowledgment packet to be sent back, indicating the correct sequence number that it is expecting. Otherwise, the next check is conducted.

(3). Acknowledge number check is an additional validity check on the packet. A valid ACK number should theoretically be within [Y, Y+*outstanding_bytes*] to be considered valid. Here Y is the first unacknowledged sequence number and *outstanding_bytes* is total number of outstanding bytes not yet acknowledged. Linux has a relaxed implementation which allows half of the ACK number space to be considered valid (we discuss its impact later). If the ACK number is considered invalid, then it is dropped without further processing. Else, the packet goes through the later non-validity-related checks.

(4). At this point the packet has the correct sequence number and the ACK number. The stack needs to check if it has any payload. If it does not have any payload, the packet is silently ignored unless there happens to be pending data that can be piggybacked. In particular, the host cannot send another 0-payload acknowledgment packet for the 0-payload incoming ACK packet, which will create endless TCP ACK storm [23].

(5). If the packet has non-zero payload, the final check is to detect retransmission by checking if the ending sequence number of the packet is smaller than or equal to the next expected sequence number. If so, it does not process the packet further and immediately sends an ACK packet to inform the other end of the expected sequence number. Since step 2 has already ensured that the ending sequence number cannot be smaller than the next expected sequence number, the only possible ending sequence number that can satisfy the retransmission check is the one equal to the next expected sequence number.

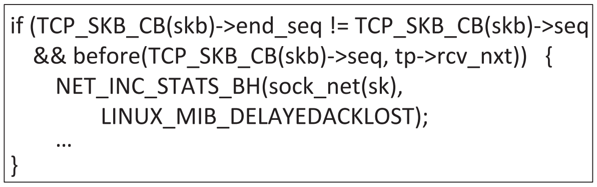
From the above description on how a TCP packet is handled, it is not hard to tell that depending on whether the sequence number is in or out of window, the TCP stack may behave differently, which can be observed by the on-device malware. Specifically, if it is an out-of-window packet with 0-payload, it most likely will not trigger any outgoing packet. However, if it is an in-window packet, it immediately triggers an outgoing duplicate ACK packet. As a result, it is possible to use the counter that records the total number of outgoing packets to tell if a guessed sequence number is in window.

A similar observation has been made by the previous study in the Phrack magazine [1]. The problem with their approach to infer sequence number is that such general packet counters can be very noisy — there may be background traffic which can increment the system-wide outgoing packet counters. It is especially problematic when the receive window size is small — a large number of packets need to be sent and the probing is very likely to have limited success. In fact, we have implemented such sequence number inference attack on a smartphone at home connected to the broadband ISP through WiFi with 10Mbps downlink bandwidth. Through 20 repeated experiments, we find that the inference always failed because of the noise of the background traffic

It is also worth noting that the error checks are performed at the very beginning, preceding the sequence number check, which means that the corresponding error counters used by the recent study [26] alone cannot provide any feedback on a guessed TCP sequence number.

3.4 Sequence-Number-Dependent Counter in Linux

The reason why the Phrack attack [1] is difficult to carry out is two-fold: (1) The required number of packets is too large; an attacker needs to send at least one packet per receive window in order to figure out the right sequence number range. (2) The counter that records the total number of outgoing packets is too noisy. Subsequently, we show that both problems can be addressed by using a newly discovered set of *sequence-number-dependent packet counters* that increment when the sequence number of an incoming packet matches certain conditions.



```
if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq
   && before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))  {
      NET_INC_STATS_BH(sock_net(sk),
            LINUX_MIB_DELAYEDACKLOST);
      …
}
```

Figure 3: tcp_send_dupack() source code snippet in Linux

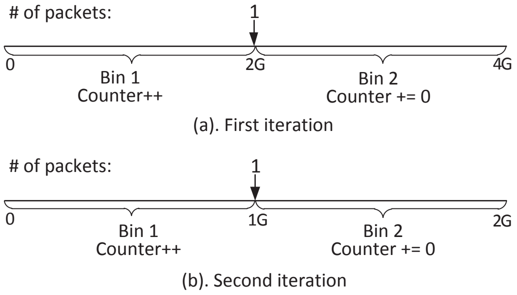
Server-side sequence number inference. We closely study the function *tcp_send_dupack()* which is called after the sequence number check (depicted in Figure 2). Within the function, we discover an interesting piece of code shown in Figure 3. The "if" condition says if the packet's starting sequence number is not equal to its ending sequence number (*i.e.,* the packet has nonzero payload), and its starting sequence number is "before" the expected sequence number, then a packet counter named *DelayedACKLost* is incremented (which is publicly accessible from /proc/net/netstat). This particular logic is to detect lost delayed ACK packets sent previously and switch from the delayed ACK mode into the quick ACK mode [12]. The presence of an old/retransmitted TCP packet is an indication that the delayed ACKs were lost.

The question is how "before()" is implemented. In Linux (and Mac OS), it basically subtracts an unsigned 32-bit integer from another unsigned 32-bit integer and converts the result into a signed 32-bit integer. This means that half of the sequence number space (*i.e.,* 2G) is considered before the expected sequence number. For instance, two unsigned integers 1G minus 2G would lead to an unsigned integer 3G. When converting to an signed value, we obtain -1G.

The net effect of the *tcp_send_dupack()* is that it allows an attacker to easily determine if a guessed sequence number is before or after the expected sequence number. Since the *DelayedACKLost* counter very rarely increments naturally (See §3.8), an attacker can use this counter as a clean and reliable side-channel.

Binary search. Using this special counter, it is straightforward to conduct a binary search on the expected sequence number. Note that the process is significantly different than the one proposed in the earlier work [26] in that the earlier work still requires sending one packet per "window", which results in a total of thousands or tens of thousands of packets. Here, as illustrated in Figure 4, the attacker only needs to send one packet each round and only a total of 32 packets, resulting in hardly any bandwidth requirement.

Specifically, as shown in the figure, in the first iteration, the attacker can try the middle of the sequence number space (*i.e.,* 2G). If the expected sequence number falls in the first half (*i.e.,* bin 1), the *DelayedACKLost* counter increments by 1. Otherwise, (*i.e.,* if it falls in bin 2), the counter remains the same. Suppose the attacker finds that the expected sequence number is in the first half after the first iteration, in the second iteration, he can try 1G to further narrow down the sequence number. After $\log_2 4G = 32$ rounds (also 32 packets), the exact sequence number can be pinpointed. The total inference time can be roughly calculated as $32 \times RTT$. In reality, the number of RTTs can be further reduced by stopping the inference at an earlier iteration. For instance, if it is stopped at the 31st iterations, the attacker would know that the sequence number is either X or X+1. Similarly, if the number of iterations is 22, the attacker knows that the sequence number is within [X, X+1024). In many cases, this is sufficient because the attacker can still inject a single packet with payload of 1460 bytes and pad the first 1024 bytes with whitespace (which effectively leaves 436 bytes of effective payload). For instance, if the application-layer protocol is HTTP, the whitespace is safely ignored even if they happen to be accepted as part of the HTTP response.



(a). First iteration

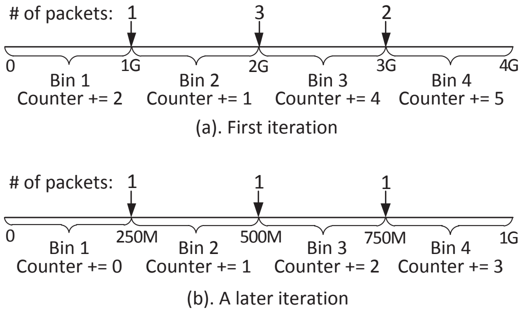
(b). Second iteration

Figure 4: Sequence number inference illustration using the *DelayedACKLost* packet counter (binary search)

N-way search. To further improve the inference speed, we devise a variation of the "N-way search" proposed in the recent work [26]. The idea is similar — 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. The difference is that the inference requires one or two orders of magnitude fewer packets compared to the previously proposed search.

Figure 5 illustrates the process of a 4-way search. In the first iteration, the search space is equally partitioned into 4 bins. The attacker sends one packet with sequence number 1G, three packets with sequence number 2G, and two packets with sequence number 3G. If the expected sequence number falls in the first bin, the *DelayedACKLost* counter increments by 2, as the two packets sent with sequence number 3G are considered before the expected sequence number. Similarly, the counter increments by a different number for different bins. In general, as long as the number of packets sent for each bin follow the distance between two consecutive marks on a circular/modular Golomb ruler [3], the *DelayedACKLost* counter increment will be unique when the expected sequence number falls in different bins.

In the later iterations, however, a much simpler strategy can be used. In Figure 5(b), an attacker can just send one packet per bin instead of following the circular Golomb ruler. The reason is that now that the search space is reduced to smaller than 2G, it is no longer circular (unlike the first iteration where the counter increment in the first bin can be impacted by the fourth bin). Now, if the sequence number falls in the first bin, then the counter remains the same; if it falls in the second bin, the counter will increment 1; and so on. We discuss the realistic settings and performance of different "N" in §3.7.



(a). First iteration

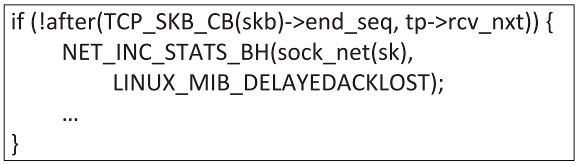
(b). A later iteration

Figure 5: Sequence number inference illustration using the *DelayedACKLost* packet counter (four-way search)

Client-side sequence number inference. Sometimes, it is necessary to infer the client-side sequence number, for the purpose of either injecting data to the victim server, or injecting data to the victim client with an appropriate ACK number. The latter is currently unnecessary as Linux/Android and BSD/Mac OS allows half of the ACK number space to be valid [26]. For the former, we can still use the same *DelayedACKLost* counter to infer the ACK number.

Specifically, as discussed in §3.3, the only ending sequence number that can satisfy the retransmission check is the one equal to the next expected sequence number. When that happens, the TCP stack increments the *DelayedACKLost* packet counter again. The source code of the retransmission check is shown in Figure 6.

Since the retransmission check is after the ACK number check, it allows an attacker to send a non-zero payload packet that has the ending sequence number equal to the next expected sequence number with a guessed ACK number. If it does not pass the ACK number check, the packet is dropped and the *DelayedACKLost* counter does not increment. Otherwise, the packet is considered a retransmitted packet and triggers the counter to increment. Based on such behavior, we can perform a binary search or N-way search on the ACK number similar to the sequence number search. In fact, the procedure is mostly identical.



```
if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
        NET_INC_STATS_BH(sock_net(sk),
                LINUX_MIB_DELAYEDACKLOST);
        ...
}
```

Figure 6: Retransmission check source code snippet from tcp_data_queue() in Linux

3.5 Sequence-Number-Dependent Counters in BSD/Mac OS

Inspired by the newly discovered counter in Linux, we further conduct a survey on the latest FreeBSD source code (version 10). Surprisingly, we find that at least four pairs of packet counters can leak TCP sequence number. The counters are confirmed to exist in Mac OS as well. This finding shows that the sequence-number-dependent counters are widely available and apparently considered safe to include in the OS. They are: 1) *rcvduppack* and *rcvdupbyte*; 2) *rcvpackafterwin* and *rcvbyteafterwin*; 3) *rcvoopack* and *rcvoobyte*; 4) *rcvdupack* and *rcvacktoomuch*. They can be either accessed through the standard "netstat -s" interface or sysctl API [11].

The first three pairs can be used to infer server-side sequence numbers. Specifically, based on the source code, the semantic of *rcvduppack* is identical to that of *DelayedACKLost*. *rcvdupbyte*, however, additionally provides information on the number of bytes (payload) carried in the incoming packets that are considered duplicate (with an old sequence number). This counter greatly benefits the sequence number inference. Following the same "N-way" procedure, the first iteration can be improved by changing the "k packets sent per bin" to "a single packet with k bytes payload". This improvement substantially reduces the number of packets/bytes sent in each iteration, especially when "N" is large (shown in §3.7).

The semantic of *rcvpackafterwin* and *rcvbyteafterwin* is similar to *rcvduppack* and *rcvdupbyte*, except that the former increments only when the sequence number is bigger than (instead of smaller than) certain sequence number X. In this case, X is the expected sequence number plus the receive window size. *rcvbyteafterwin* can be used similarly as *rcvdupbyte* to conduct the sequence number inference.

rcvoopack and *rcvoobyte* differ from the previous two pairs. They increment only when packets arrive out of order, or more precisely, when the sequence number is bigger than the expected sequence number yet smaller than the expected sequence number plus the receive window size. Even though an attacker needs to send a lot more packets to infer the TCP sequence number using this counter pair, at least they can be used to replace the original noisy side-channel in the Phrack attack [1] to improve success rate.

rcvdupack and *rcvacktoomuch* are used to determine the client-side sequence numbers. Specifically, the former increments when the ACK number of an incoming packet is smaller than or equal to the unacknowledged number (SND.UNA). The latter increments when the ACK number is greater than the sequence number of the next original transmit (SND.MAX). The comparison again follows the "unsigned integer to signed integer conversion" such that half of the ACK number space is considered to match the condition.

We currently did not combine the counters together to improve the inference speed. However, we do realize there are potential ways to speed things up. For instance, the *rcvdupbyte* and *rcvdupack* allows the client-side sequence number inference to be piggybacked with the server-side sequence number inference.

3.6 Sequence-Number-Dependent Counters in Microsoft Windows

Interestingly, Microsoft Windows OSes do not appear to expose such sequence-number-dependent counters and are thus not vulnerable to the attack. On Windows 7, for example, the TCP-related packet counters include the total number of incoming packets, outgoing packets, and the number of packets retransmitted from the output of "netstat -s". These packet counters do not leak sequence numbers directly.

3.7 Inference Performance and Overhead

We have implemented the sequence number inference on both Android (which incorporates the Linux kernel) and Mac OS. We are interested in the tradeoffs between different strategies in picking "N" in the "N-way search".

Generally, as "N" goes up, the total number of bytes sent should also increase. Since the first iteration in the "N-way" search requires sending more bytes, we pick a smaller "N" for the first iteration and a bigger "N" in the later iterations to ensure that the number of bytes sent in each round is similar. In the Linux implementation, we pick the following pairs of N, (2/2, 4/6, 8/30, 12/84); For Mac OS, we pick (2/2, 4/6, 34/50, 82/228). Here 4/6 means that we pick N=4 for the first iteration and N=6 for the later iterations.

As shown in Figure 7, we can see that the general tradeoff is that the fewer iterations an attacker wants, the more bytes he needs to send in total. For instance, when the number of iterations is 4, an attacker on Linux needs to send 13.7KB. With the presence of the *rcvdupbyte* counter in Mac OS, it requires to send only 8.4KB. This is a rather low network resource requirement because it takes only 70ms to push 8.4KB onto the wire with even just 1Mbps bandwidth. Going further down to 3 iterations requires sending 27.75KB for Mac OS. Depending on the available bandwidth and the RTT, we may or may not want to increase the number of bytes to save one round trip.

Next, we pick N=34/50 (4 round trips) for Mac OS attacks, and N=8/30 (5 round trips) for Linux attacks (with roughly the same resource requirement), and plot the inference time measured under various conditions. We control the RTT between the attacker and the victim in three different settings: 1) The victim is in an office environment (enterprise-like) connected to the network using WiFi, and the attacker is in the same building (the RTT is around 5-10ms). 2) The victim is in a home environment and the attacker is 50ms RTT away. 3) The victim is in a home environment and the attacker is 100ms RTT away. In Figure 8, we see that in the first setting the inference time for Android and Mac OS are 80ms and 50ms, which are low enough to directly launch injection attacks on HTTP connections with the guarantee that the inference finishes before the first legitimate response packet comes back (also discussed later in §4.2). In fact, inference time between 350ms and 700ms can be short enough in certain scenarios (see §5.1).

3.8 Noisiness of Sequence-Number-Dependent Counters

So far, we have claimed that these sequence-number-dependent counters are "clean" side-channels that rarely increment naturally even with background traffic. To quantitatively support this claim, we conduct a worse-case-scenario experiment as follows: We open a YouTube video at the background and browse web pages at the same time to see how often the counters get incremented. Since it is easier to do the multi-tasking on Mac OS, we choose it over the Android platform. The Android counters should increment even less frequently since smartphones are rarely used for video streaming and web browsing simultaneously.

We pick the *rcvdupbyte* counter (which is equivalent to *DelayedACKLost* on Linux) and run the experiments for about

8.5 minutes. The video is long enough that it has not been fully buffered by the end of the experiment. To quantify the counter noisiness, we break down the time into 30ms intervals to mimic the window of exposure during one round of probing, and then count how many intervals in which we observe any counter increment. As expected, there are only 10 intervals out of 16896 that have the increment. This indicates that the probability that the counter increments due to noise and interference with one round of probing is roughly 0.059%. Even if there are 22 rounds (worse case), the probability that the entire probing will be affected by the counter noisiness is only 1.2%.

4. DESIGN AND IMPLEMENTATION OF TCP ATTACKS

In the previous section, we described how to infer TCP sequence number efficiently and reliably using the newly discovered set of sequence-number-dependent packet counters. Since the sequence number inference only takes less than a second, it can be fast enough to launch many application-layer attacks. In this section, we discuss four possible TCP attacks that can be launched against a variety of applications. All of the attacks leverage the TCP sequence number inference as the essential building block, but the main difference is in the timing and reliability with slightly different requirements. We have implemented the attacks on both Android and Mac OS. We use Android as the example for description

Injection vs. Hijacking. Using the same terminology as a recent work [26], we define TCP hijacking to be the more powerful attack than TCP injection. Specifically, TCP hijacking allows an attacker to inject packets right after the TCP 3-way handshake. For instance, it enables an attacker to inject a complete HTTP response without any interference. In contrast, *TCP Injection* is more general and does not require this capability.

The four attacks are named as: (1). client-side TCP Injection, (2). passive TCP hijacking, (3). active TCP hijacking, (4). server-side TCP injection.

4.1 Attack Requirements

There are a number of base requirements that need to be satisfied for all of these TCP attacks. Note that our attacks have much fewer requirements than the one proposed in the recent study [26]. Specifically, we do not require a firewall middlebox in the network, which makes our attacks applicable in a much more general environment.

The set of requirements include: (1) malware on the client with Internet access, (2) malware that can run in the background and read packet counters, (3) malware that can read the list of active TCP connections and their four tuples, and (4) a predictable external port number if NAT is deployed. The first three requirements are straightforward. All of the Android applications can easily request Internet access, read packet counters (*i.e.,*/proc/net/netstat and /proc/net/snmp, or "netstat -s"), and read active TCP connections' four tuples (*e.g.,* through /proc/net/tcp and /proc/net/tcp6, or "netstat"). The requirements can be easily satisfied on most modern OSes as well. In addition, an off-path attacker needs the client's external port mapping to choose the correct four tuples when sending probing packets, so we need the fourth requirement. This requirement is also commonly satisfied, since many NAT mapping types allow the external port to be predictable to facilitate NAT traversal. For instance, our home routers directly map the internal ports to the external ports. According to recent measurement studies on the NAT mapping types [21, 31], the majority of the NATs studied do have predictable external ports. Further, even if the prediction is not 100% accurate, attacks may still succeed by guessing the mappings.

Additional requirements for passive TCP hijacking are C1 and S1:

(C1). Client-side ISN has only the lower 24-bit randomized. This requirement is necessary so that the malware can roughly predict the range of the ISN of a newly created TCP connection. In Linux kernels earlier than 3.0.2, the ISN generation algorithm is designed such that ISNs 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). This feature is designed to balance security, reliability, and performance. It is long perceived as a good optimization, with the historical details and explanations in this article [5]. 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. Even though it is a design decision and not considered a "vulnerability", since Linux 3.0.2, the kernel has changed the ISN generation algorithm such that two consecutive connections will have independent ISNs. The majority of Android systems that are on the market are still on Linux 2.6.XX, which means that they are all vulnerable to the passive TCP hijacking attack.

(S1). The legitimate server has a host-based stateful TCP firewall. Such a firewall is capable of dropping out-of-state TCP packets. Many websites such as Facebook and Twitter deploy such host firewalls to reduce malicious traffic. For instance, iptables can be easily configured to achieve this purpose [10]. Interestingly, as we will discuss later, this security feature on the server actually enables TCP hijacking attacks.

In order to perform active TCP hijacking attacks, the additional requirements include S1 and C2:

(C2). Client-side ISN monotonically incrementing for the same four tuples. This client-side requirement is in fact explicitly defined in RFC 793 [9] to prevent packets of old connections, with in-range sequence numbers, from being accepted by the current connection mistakenly. Even though the latest Linux kernel has eliminated the requirement C1, C2 is still preserved.

4.2 Client-Side TCP Injection

In this attack, an attacker attempts to inject malicious data into a connection established by other apps on the phone. The essential part of the attack is the TCP sequence number inference which has already been described in detail. The challenge is that the injected data may compete with the data sent from the legitimate server. For instance, considering the connection under attack is an HTTP session where a valid HTTP response typically follows immediately after the request is sent, by the time the sequence number inference is done, at least part of the HTTP response is already sent by the server. The injected HTTP packets likely can only corrupt the response and cause denial of service instead of serious damage.

Even though the timing requirement sounds difficult to satisfy, we did implement this attack against websites such

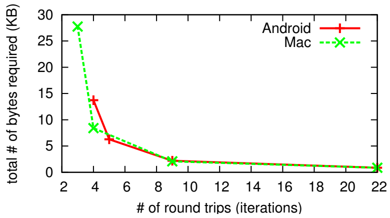


Figure 7: Tradeoff between inference speed and overhead

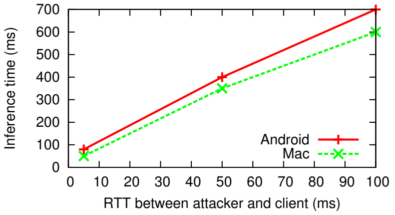


Figure 8: Relationship between RTT and inference time

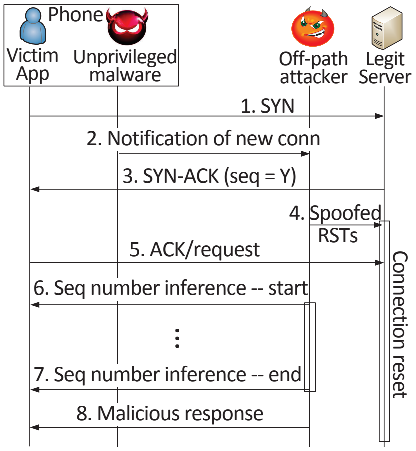


Figure 9: Passive TCP hijacking sequence

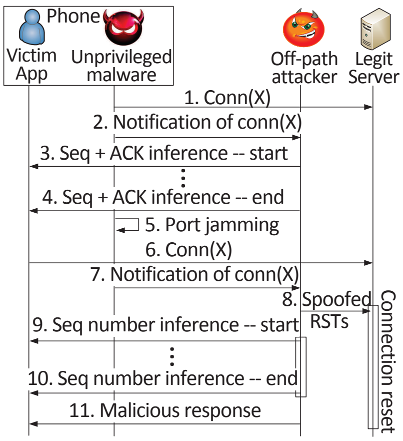

Phone
Victim App
Unprivileged malware
Off-path attacker
Legit Server
1. Conn(X)
2. Notification of conn(X)
3. Seq + ACK inference -- start
4. Seq + ACK inference -- end
5. Port jamming
6. Conn(X)
7. Notification of conn(X)
8. Spoofed RSTs
9. Seq number inference -- start
10. Seq number inference -- end
11. Malicious response
Connection reset

Figure 10: Active TCP hijacking sequence

as Facebook where we are able to inject malicious Javascripts to post new status on behalf of a victim user. The detail is described in §5.1.

The idea is to leverage two common scenarios: (1) The server may take a long time to process a request and assemble the response. This is especially common as many services (websites) take longer than 100ms or more to process a request. The fact that the sequence number inference time in certain scenarios (when RTT from the server to the client is small) can be made below 100ms makes the injection attack as powerful as hijacking. (2) A single TCP connection is reused for more than one pair of HTTP request and response. The idea is to use the inferred sequence number for injecting malicious data not on the first HTTP request but the later ones. In both cases, an attacker has enough time to conduct sequence number inference.

4.3 Passive TCP Hijacking

Passive TCP hijacking allows an attacker to hijack TCP connections that are passively detected. This means that the attacker can hijack TCP connections issued by the browser or any other app, regardless of how and when they are made. It is the most powerful TCP attack in this study. As demonstrated in §5, with this attack, it is possible to replace the Facebook login page with a phishing one.

The high-level idea is the same as proposed in the recent work [26], which 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 has to be triggered right after the legitimate server sends SYN-ACK. Requirement C1 allows the malware and the attacker to predict the rough range of victim's ISN and send reset packets with sequence numbers in that range. This is helpful because the attacker is required to send 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 since it helps prevent the legitimate server from generating RST upon receiving out-of-state data or ACK packets from the victim.

The attack sequence diagram is shown in Figure 9. Time steps 1 to 3 are the same as the previous attack where the unprivileged malware detects and reports the newly established TCP connection. In addition, the malware also needs to establish a connection to the off-path attacker to report the current ISN value (high 8 bits). With this information, at time 4, the off-path attacker can flood the legitimate server with a number of spoofed RSTs enumerating the lower 24 bits (sequence numbers can increment by a step size as large as the server's receive window size). Note that the RST packets have to arrive before the ACK/request packets at time 5; otherwise, the server may send back the response packets before the attacker. Of course, the server may need some time to process the request as well, which can vary from case to case, allowing the attacker additional time to complete the reset procedure. After the legitimate server's connection is reset, all future packets from the victim app will be 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, the attacker conducts the sequence number inference described earlier and injects malicious content afterwards at time 8 with the inferred sequence number. A more detailed analysis on the bandwidth and time requirement is discussed in a similar setting in a prior work [26].

4.4 Server-side TCP Injection

In this attack, an attacker tries to inject malicious payload into a victim connection, destined for the server (as opposed to the client). For instance, as shown in the case study in §5, we are able to target at Windows live messenger protocols to inject malicious commands to cause persistent changes to the victim user account, *e.g.,* adding new friends or removing existing friends.

This attack is straightforward by combining the sequence number inference and ACK number inference as described in §3. We omit the detailed attack sequence as it does not include other important steps. This attack has no additional requirements besides the base requirements. In general, applications with unencrypted and stateful protocols are good attack targets.

4.5 Active TCP Hijacking

In this attack, an attacker attempts to hijack connections. However, because the latest Linux kernel since 3.0.2 has the entire 32-bit randomized for ISNs of different four tuples, requirement C1 is no longer satisfied. In this case, we show that it is still possible to launch a weaker version of TCP hijacking by "actively" performing offline analysis as long as requirement C2 is satisfied. As shown in §5, we have successfully used the port-jamming-assisted active TCP hijacking to replace a Facebook login page with a phishing one.

Requirement C2 specifies that the ISN for the same four-tuple always increments with time. This implies that as long as an attacker can infer the client's ISN for a particular four-tuple once, he can store the value for a future connection that reuses the same four-tuple, and reset the server using the stored ISN (plus the increment by time) so that the attacker can hijack the connection.

The detailed attack sequence is demonstrated in Figure 10, at time 1, the unprivileged malware establishes a connection on its own to a target server of interest (*e.g.,* Facebook server), and notifies the off-path attacker immediately (at time 2) so that it can infer the client ISN of the used four tuples (through time 3 to 4). Now, assuming that the attacker knows that a victim app is about to initiate a connection to the same server, an attacker can immediately perform port jamming to exhaust all the local port numbers (at time 5) so that the victim app's connection can only use the local port number that was in the inferred four tuples (we will describe how port jamming can be done later). Now that the victim connection reuses the same four tuples, the malware can immediately notify the off-path attacker (at time 6) which uses the previously inferred client-side ISN to reset the server (at time 7). Subsequently, the attack sequence is identical to the end of passive TCP hijacking.

In the above attack sequence, one critical part is the knowledge of when the victim app initiates the connection to the target website. One simple strategy is to actively trigger the victim app to make the connection through the unprivileged malware. On Android, for instance, any app could directly invoke the browser going to a given URL, before which the attacker can perform the port jamming.

One alternative strategy is to perform offline analysis on as many four tuples as possible so that it can essentially obtain the knowledge of ISN for all possible four tuples going to a particular website (without requiring port jamming). This way, after the offline analysis is performed, the attacker basically can launch passive TCP hijacking on any of the four tuples that have been previously analyzed. Since each client-side ISN inference should take a little over a second, an attacker can infer, for instance, 1000 four tuples in 15 minutes. Even though a connection to Facebook may have 1% probability falling in the range, the user may repeatedly visit the website and the probability that all of the connections failing to match any existing four tuples is likely very low. We have verified that the ISN for the same four-tuple does increment consistently over time for over an hour. We suspect that the cryptographic key for computing ISN does not change until reboot in Linux 3.0.2 and above.

To jam local ports, the unprivileged malware can simply start a local server, then open many connections to the local server intentionally occupying most of the local port except the ones that are previously seen for inference. 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. Interestingly, we find such limit can be bypassed if the established connections are immediately closed (which no longer counts towards the limit). The local port numbers are not immediately released since the closed connections enter the TCP_TIME_WAIT state for a duration of 1 to 2 minutes.

5. ATTACK IMPACT ANALYSIS FROM CASE STUDIES

Experiment setup. As discussed earlier, even though our attacks are implemented on both Android and Mac OS, we choose to focus on Android in our implementation and experiments. We use two different phones: Motorola Atrix and Samsung Captivate. We verified that all attacks work on both Android phones, although the experimental results are repeated based on Atrix. The WiFi networks include a home network and a university network. The off-path attacker is hosted on one or more Planetlab nodes in California.

We describe four case studies corresponding to the four TCP attacks proposed in the previous section. We also present experimental results such as how likely we can succeed in hijacking the Facebook login page based on repeated experiments.

For all attacks, we implemented the malware as a benign app that has the functionality of downloading wallpapers from the Internet (thus justifying the Internet permission). Since the malware needs to scan netstat (or /proc/net/tcp and /proc/net/tcp6 equivalently) for new connection detection, which can drain the phone's battery very quickly, we make the malware stealthy such that it only scans for new connections when it detects that the victim app of interest is at the foreground. This can be achieved by querying each app's *IMPORTANCE_FOREGROUND* flag which is typically set by the Android OS whenever it is brought to the foreground. Further, the malware queries the packet counter only when the off-path attacker instructs it to do so. The malware is only used in our controlled experiment environments without affecting real users.

Note that most apps except the browser on the smartphones do not have an indication about whether the connection is using SSL, which means that the users may be completely unaware of the potential security breach for unencrypted connections (*e.g.,* HTTP connections used in the Facebook app).

5.1 Facebook Javascript Injection

We launch the attack based on **client-side TCP injection** as described in §4.2. Recall that the injection can happen only after the sequence number inference finishes. If the inference cannot be done earlier than the response comes back, the attacker will miss the window of opportunity.

By examining the packet trace generated by visiting the Facebook website where a user is already logged in, we identify two possible ways to launch the Javascript injection attack. The first attack is surprisingly straightforward. Basically, when the user visits `m.facebook.com`, the browser issues an HTTP request that fetches all recent news. We observe that it consistently takes the server more than 1.5 seconds to process the request before sending back the first response packet. According to our results in §3.7, the inference time usually finishes within 0.7s even when RTT=100ms. It allows enough time for an attacker to inject the malicious response in time (or inject a phishing login page as well). As shown in Table 1, the success rate is 87.5% based on 40 repeated experiments in our home environment where

	RTT_a=70ms[1]	RTT_a=100ms
Succ Rate	97.5% (39/40)	87.5% (35/40)

[1] RTT_a is the RTT between the attacker and the client

Table 1: Success rate of Facebook Javascript injection (case study 1)

RTT=100ms. It goes up to 97.5% when the experiment is conducted in the university network where RTT=70ms. The failed cases are mostly due to packet loss.

The second attack is based on the observation that multiple requests are issued over the same TCP connection to the Facebook site. Even if the attacker is not able to infer the sequence number in time to inject response for the first request (*e.g.,* Facebook may improve the server processing time in the future), he can still perform inference for the second request. Specifically, if the user visits the root page on Facebook, the browser on one of the Android phones (Samsung Captivate) will send two HTTP requests: the first request is asking for the recent news; the second request seems to be related to prefetching (*e.g.,* retrieving the friend list information in case a user clicks on any friend for detailed information).

Since there is a delay of about 1s between the end of the first request and the start of the second request, an attacker can monitor if the sequence number remains the same for a certain period of time to detect the end of the first response. Furthermore, the second request takes about 100ms to process on the server. A simple strategy that an attacker can employ is to just wait for around 1.1s before injecting the malicious response for the second request. A more sophisticated attacker could also monitor the start of the second request by tracking the current ACK number. Specifically, when the second request is sent, the valid ACK number range moves forward by the number of bytes in the request payload.

In our proof-of-concept implementation, we always inject the Javascript after waiting for a fixed amount of time after the connection is detected, which can already succeed for a few times. However, a more sophisticated attacker can definitely do better.

5.2 Phishing Facebook Login Page

We launch this attack based on **passive TCP hijack** which passively monitors if a new connection to Facebook is made. In this case study, we look at how to replace the Facebook login page by resetting the Facebook immediately after it has responded with SYN-ACK.

We assume that the user is not already logged in to Facebook. Otherwise, as described in the previous attack, the server processing delay for the first HTTP request is so long that is is too easy to succeed. When the user is not logged in, the server processing delay will become negligible and the effective time window for reset to succeed is basically a single round trip time. This scenario is also generic enough that the attack can be applied for many other websites such as `twitter.com`.

In Table 2, we show how likely the attack can succeed under different conditions. For instance, when there's a single Planetlab node, the success rate is a little below 50%. However, when we use two nodes for latency values of 70ms and 100ms respectively, the success rate increases significantly to 62.5% and 82.5% , indicating that we have more bandwidth to reset the server. In addition, the result also verifies that the larger the RTT, the more likely the attack can succeed.

Note that the 100ms RTT to Facebook may sound very large given the popularity of CDN services. However, the CDNs are mostly used for hosting static contents such as images and Javascripts. For webpages that are highly customized and dynamic (*e.g.,* personalized Facebook news feed), they are very likely to be stored on the main server in a single location (*e.g.,* Facebook main servers are hosted in California). We find that this is a common design for many sites with dynamic contents (*e.g.,* twitter).

5.3 Command Injection on Windows Live Messenger

Leveraging **server-side TCP injection** described in §4.4, the case study of command injection attack on Windows Live Messenger is an interesting example of server-side attack carried out on a connection where the user is already logged in. The main connection of Windows Live Messenger runs on port 1863 and uses Microsoft Notification Protocol (MSNP) which is a complex instant messenger protocol developed by Microsoft [6]. Many Windows Live Messenger clients on Android as well as the ones on the desktops (including official ones) use plaintext in their connections, thus allowing the attack. Once upon the detection of a vulnerable Windows Live Messenger app running or a connection established to known port numbers and IP addresses that are associated with the app, an attacker can launch this attack.

We have verified that the commands that are possible to inject into the server include, but not limited to, (1) adding a new friend or removing an existing friend (specified by the account email address), (2) changing the status messages, and (3) sending messages to friends. Given that the messenger client is idle most of the time and the fact that the client-side sequence number inference only takes 2–3 seconds, the attack can be launched fairly easily. The commands can cause serious damage. For instance, the add-friend command allows an attacker to add its malicious account as a friend which can subsequently send spam or phishing messages. In addition, after being added as a friend, the attacker can read the friend list (email accounts) of the victim user, delete them, or spam them. Finally, new status posting can be part of the phishing attack against the friends as well.

5.4 Restricted Facebook Login Page Hijack

This attack is launched based on **active TCP hijack** as described in §4.5. The goal of this attack is still to hijack TCP connections. However, due to the lack of ability to reset the server-side connection in the new version of the Linux kernel, it requires offline analysis on the client-side ISN of the target four tuples.

In our implementation, we develop a simple Android test malware that performs the offline analysis right after it is started. The four tuples we target include a pre-selected local port and the Facebook server IP that's resolved for m.facebook.com. After the analysis, the attack takes a lit-

	One Planetlab node		Two Planetlab nodes	
	RTT_b=70ms[1]	RTT_b=100ms	RTT_b=70ms	RTT_b=100ms
Succ Rate	42.5% (17/40)	47.5% (19/40)	62.5% (25/40)	82.5% (33/40)

[1] RTT_b is the RTT between the attacker and the Facebook server

Table 2: Success rate of Facebook login page injection (case study 2)

tle over one second, and it performs port jamming immediately (which takes about 5 seconds). After this, our malware app immediately sends an Intent that asks to open m.facebook.com through the browser. An attacker may come up with reasons such as asking a user to use his Facebook account to register for the app. When the browser starts the connection to Facebook, the malware works with the off-path attacker to hijack the connection (as described in §4.5). We have verified that the Facebook login page can indeed be hijacked following these steps.

The main difficulty in this attack is not about successfully inferring the sequence number. Instead, it requires the user to be convinced that the app indeed has a relationship with the target website (*i.e.,* Facebook) so that the user will enter his password into the browser.

6. DISCUSSION AND CONCLUSION

From these real attacks, there are a few lessons that we learn: (1) Even though OS statistics are aggregated and seemingly harmless, they can leak critical internal network/system state through unexpected interactions from the on-device malware and the off-path attacker. Similar observations have been made recently in a few other studies as well, *e.g.,* using procfs as side channels [22]. Our study reveals specifically that the packet counters can leak TCP sequence numbers. (2). Our systems today still have too much shared state: the active TCP connection list shared among all apps (through netstat or procfs); the IP address of the malware's connection and other apps'; the global packet counters. Future system and network design should carefully evaluate what information an adversary can obtain through these shared state.

On the defense side, there are a few measures that may improve security: (1) always using SSL/TLS, (2) removing unnecessary global state (such as the active TCP connection list and packet counters) or only allow privileged programs to access such state, (3) providing better isolation among resources, *e.g.,* providing a separate set of packet counters for each app. With IPv6 widely deployed, we may even provide different source IP addresses for connections in different processes on a device so that malware will not be able to learn the IP address of the connection established by another process. In the extreme case, each app may run in its own virtual machine.

To conclude, we have demonstrated an important type of TCP sequence number inference attack enabled by host packet counter side-channels under a variety of client OS and network settings. We also offer insights on why they occur and how they can be mitigated.

7. REFERENCES

[1] Blind TCP/IP Hijacking is Still Alive. `http://www.phrack.org/issues.php?issue=64&id=15`.

[2] CERT Advisory CA-1995-01 IP Spoofing Attacks and Hijacked Terminal Connections. `http://www.cert.org/advisories/CA-1995-01.html`.

[3] Golomb Ruler. `http://en.wikipedia.org/wiki/Golomb_ruler`.

[4] Linux Blind TCP Spoofing Vulnerability. `http://www.securityfocus.com/bid/580/info`.

[5] Linux: TCP Random Initial Sequence Numbers. `http://kerneltrap.org/node/4654`.

[6] MSN Messenger Protocol. `http://www.hypothetic.org/docs/msn/`.

[7] RFC 1948 - Defending Against Sequence Number Attacks. `http://tools.ietf.org/html/rfc1948`.

[8] RFC 5961 - Improving TCP's Robustness to Blind In-Window Attacks. `http://tools.ietf.org/html/rfc5961`.

[9] RFC 793 - Transmission Control Protocol. `http://tools.ietf.org/html/rfc793`.

[10] Stateful Firewall and Masquerading on Linux. `http://www.puschitz.com/FirewallAndRouters.shtml`.

[11] sysctl Mac OS X Manual. `https://developer.apple.com/library/mac/#documentation/Darwin/Reference/Manpages/man3/sysctl.3.html#//apple_ref/doc/man/3/sysctl`.

[12] TCP Delayed Ack in Linux. `http://wiki.hsc.com/wiki/Main/InsideLinuxTCPDelayedAck`.

[13] 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.

[14] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proc. of USENIX Security Symposium*, 2011.

[15] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*, 2011.

[16] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[17] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. of USENIX Security Symposium*, 2011.

[18] 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.

[19] 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.

[20] Y. Gilad and A. Herzberg. Off-Path Attacking the Web. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[21] S. Guha and P. Francis. Characterization and Measurement of TCP Traversal through NATs and Firewalls. In *Proc. ACM SIGCOMM IMC*, 2005.

[22] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Proc. of IEEE Security and Privacy*, 2012.

[23] L. Joncheray. A Simple Active Attack against TCP. In *Proc. of USENIX Security Symposium*, 1995.

[24] G. LEECH, P. RAYSON, and A. WILSON. Procfs Analysis. `http://www.nsa.gov/research/_files/selinux/papers/slinux/node57.shtml`.

[25] R. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. Technical report, 1985.

[26] Z. Qian and Z. M. Mao. Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security. In *Proc. of IEEE Security and Privacy*, 2012.

[27] 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.

[28] 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.

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

[30] M. Vuagnoux and S. Pasini. Compromising electromagnetic emanations of wired and wireless keyboards. In *Proc. of USENIX Security Symposium*, 2009.

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

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

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