Off-Path TCP Exploits: Global Rate Limit Considered Dangerous

Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, Lisa M. Marvel[†]
University of California, Riverside, [†]*US Army Research Laboratory*
{ycao009,zhiyunq,zwang048,tdao006,krish}@cs.ucr.edu, lisa.m.marvel.civ@mail.mil

Abstract

In this paper, we report a subtle yet serious side channel vulnerability (CVE-2016-5696) introduced in a recent TCP specification. The specification is faithfully implemented in Linux kernel version 3.6 (from 2012) and beyond, and affects a wide range of devices and hosts. In a nutshell, the vulnerability allows a blind off-path attacker to infer if any two arbitrary hosts on the Internet are communicating using a TCP connection. Further, if the connection is present, such an off-path attacker can also infer the TCP sequence numbers in use, from both sides of the connection; this in turn allows the attacker to cause connection termination and perform data injection attacks. We illustrate how the attack can be leveraged to disrupt or degrade the privacy guarantees of an anonymity network such as Tor, and perform web connection hijacking. Through extensive experiments, we show that the attack is fast and reliable. On average, it takes about 40 to 60 seconds to finish and the success rate is 88% to 97%. Finally, we propose changes to both the TCP specification and implementation to eliminate the root cause of the problem.

1 Introduction

TCP and networking stacks have recently been shown to leak various types of information via side channels, to a blind off-path attacker [22, 14, 12, 21, 11, 29, 5]. However, it is generally believed that an adversary cannot easily know whether any two arbitrary hosts on the Internet are communicating using a TCP connection without being on the communication path. It is further believed that such an off-path attacker cannot tamper with or terminate a connection between such arbitrary hosts. In this work, we challenge this belief and demonstrate that it can be broken due to a subtle yet serious side channel vulnerability introduced in the latest TCP specification.

The two most relevant research efforts are the following: 1) In 2012, Qian *et al.*, framed the so called "TCP sequence number inference attack", which can be launched by an off-path attacker [22, 23]. However, the attack requires a piece of unprivileged malware to be running on the client to assist the off-path attacker; this greatly limits the scope of the attack. 2) In 2014, Knockel *et al.*, identified a side channel that allows an off-path attacker to count the packets sent between two arbitrary hosts [21]. The limitation is that the proposed attack requires on average, an hour of preparation time and works at the IP layer only (cannot count how many packets are sent over a specific TCP connection).

In this paper, we discover a much more powerful off-path attack that can quickly 1) test whether any two arbitrary hosts on the Internet are communicating using one or more TCP connections (and discover the port numbers associated with such connections); 2) perform TCP sequence number inference which allows the attacker to subsequently, forcibly terminate the connection or inject a malicious payload into the connection. We emphasize that the attack can be carried out by a purely off-path attacker without running malicious code on the communicating client or server. This can have serious implications on the security and privacy of the Internet at large.

The root cause of the vulnerability is the introduction of the *challenge ACK* responses [26] and the global rate limit imposed on certain TCP control packets. The feature is outlined in RFC 5961, which is implemented faithfully in Linux kernel version 3.6 from late 2012. At a very high level, the vulnerability allows an attacker to create contention on a shared resource, i.e., the global rate limit counter on the target system by sending spoofed packets. The attacker can then subsequently observe the effect on the counter changes, measurable through probing packets.

Through extensive experimentation, we demonstrate that the attack is extremely effective and reliable. Given any two arbitrary hosts, it takes only 10 seconds to successfully infer whether they are communicating. If there is a connection, subsequently, it takes also only tens of seconds to infer the TCP sequence numbers used on the connection. To demonstrate the impact, we perform case studies on a wide range of applications.

The contributions of the paper are the following:

- We discover and report a serious vulnerability unintentionally introduced in the latest TCP specification which is subsequently implemented in the latest Linux kernel.
- We design and implement a powerful attack exploiting the vulnerability to infer 1) whether two hosts are communicating using a TCP connection; 2) the TCP sequence number currently associated with both sides of the connection.
- We provide a thorough analysis and evaluation of the proposed attack. We present case studies to illustrate the attack impact.
- We identify the root cause of the subtle vulnerability and discuss how it can be prevented in the future. We propose changes to the kernel implementation to eliminate or mitigate the side channel.

2 Background

Security was not the primary concern in the design of TCP. There have been many security patches over the years at both the specification and implementation level. Interestingly, most new specifications are well thought out and typically improve security. Unfortunately, as we discover, one of the most recent specifications intended to improve security creates an even more serious vulnerability.

In this section, we first present the threat model that is being addressed in RFC 5961 and how the new specification is supposed to protect against blind in-window attacks. In the next section, we will show that how this specification introduces a new vulnerability.

Threat model: As illustrated in Figure 1, a realistic threat model for TCP is off-path attacks. There are three hosts involved: a victim client, a victim server and an off-path attacker. Any machine might act as the attacker in this model as long as its ISP allows the off-path attacker to send packets to the server with the spoofed IP address of the victim client. Alternatively, as shown in Figure 2, the off-path attacker is able to send packets to the client with the spoofed IP address of the victim server.

Blind in-window attacks: Under the above threat models, the most common attacks considered are "blind in-window attacks" where an off-path attacker sends spoofed TCP packets with guessed sequence numbers in an attempt to achieve DoS or data injection attacks. To succeed in such an attack, it is necessary to first know the target 4-tuple <src IP, dst IP, src port, dst port> of an ongoing TCP connection between a client and a server [1]. Once the correct 4-tuple is known, if the guessed sequence number of the spoofed packet happens to fall in the receive window, (called an in-window sequence number), one can in fact reset or inject *acceptable* malicious data into the connection. To be more precise, an in-window sequence number is one that satisfies the following condition, ($RCV.NXT \leq SEG.SEQ \leq RCV.NXT + RCV.WND$), where $SEG.SEQ$ is the guessed sequence number, $RCV.NXT$ and $RCV.WND$ are the sequence number of the next byte that the receiver expects to receive, and the receive window size, respectively. To carry out a blind attack, one typically needs to blast the entire sequence number space by sending a large sequence of spoofed packets. In this sequence, the sequence number of a packet is larger than that of its predecessor by a window size.

To defend against such attacks, RFC 5961 proposes several modifications on how TCP should process incoming packets, We highlight only the necessary details below.

2.1 Mitigating the Blind Reset Attack using the SYN Bit

An attacker might tear down an existing TCP connection by injecting SYN packets (TCP packets in which the SYN flag is set). This is because a valid SYN packet will cause the receiver to believe that the sender has restarted and thus, the connection should be reset.

In the former (pre-RFC 5961) Linux kernel versions, an incoming SYN packet is processed as follows:

- If the sequence number is outside the valid receive window, the receiver will send an ACK back to sender.
- If the sequence number is in-window, the receiver will reset this connection.

It is obvious that the attacker only needs a single SYN packet with an in-window sequence number to reset an ongoing TCP connection. Instead, RFC 5961 proposes modifications in processing the SYN packets as follows:

- If a receiver sees an incoming SYN packet, regardless of the sequence number, it sends back an ACK (referred to as a challenge ACK) to the sender to confirm the loss of the previous connection.
- If the packet is indeed initiated from the legitimate remote peer, it must have truly lost the previous connection and is now attempting to initiate a new one. Upon receiving the challenge ACK, the remote peer will send a RST packet with the *correct* sequence num-

[1] This can be achieved, among other methods, through brute-force attempts.

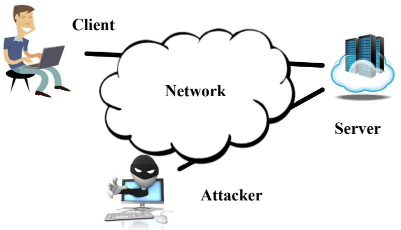


Figure 1: Threat model 1

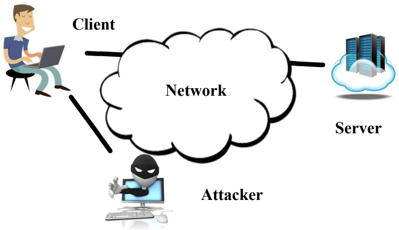


Figure 2: Alternative threat model

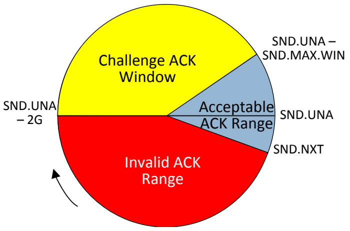


Figure 3: ACK window illustration

ber (derived from the ACK field of the challenge ACK packet) to prove that the previous connection is indeed terminated.

Hence, if the SYN packet is a spoofed one, it can no longer terminate a connection with an in-window sequence number.

2.2 Mitigating the Blind Reset Attack using the RST Bit

An attacker might also tear down the connection by injecting RST packets (TCP packets in which the RST flag is set) into an ongoing TCP connection.

In pre-RFC 5961 Linux kernels, just like in the SYN packet case, a RST packet can terminate a connection successfully as long as its sequence number is in-window. RFC 5961 suggests the following changes:

- If the sequence number is outside the valid receive window, the receiver simply drops the packet. No modifications are proposed for this case.
- If the sequence number *exactly* matches the next expected sequence number ($RCV.NXT$), the connection is reset.
- If the sequence number is in-window but does not exactly match $RCV.NXT$, the receiver must send a challenge ACK packet to the sender, and drop the unacceptable RST packet.

In the final case, if the sender is legitimate, it sends back a RST packet with the correct sequence number (derived from the ACK number in the challenge ACK) to reset the connection. On the other hand, if the RST is spoofed, the challenge ACK packet will not be observable by the off-path attacker. Therefore, the attacker needs to be extremely lucky to be able to succeed — only one out of 2^{32} sequence numbers will be accepted.

2.3 Mitigating the Blind Data Injection

An attacker might corrupt the contents of a transmission by injecting spoofed DATA packets. When a packet arrives, the receiver first checks the sequence number to make sure it is in-window; in addition, the ACK number will be checked. Pre-RFC 5961, the ACK number is considered valid as long as it falls in the wide range of $[SND.UNA - (2^{31} - 1), SND.NXT]$; this is effectively half of the ACK number space. Here, $SND.UNA$ is the sequence number of the first unacknowledged byte. $SND.NXT$ is the sequence number of the next byte about to be sent.

RFC 5961 suggests a much smaller valid ACK number range of $[SND.UNA - MAX.SND.WND, SND.NXT]$, where $MAX.SND.WND$ is the maximum window size the receiver has ever seen from its peer. This is illustrated in Figure 3. The reasoning is that the only valid ACK numbers are those that are (i) not too old (bytes that are recently sent) and (ii) not too new (receiver cannot ACK bytes that are yet to be sent). The remaining ACK values will be in the range of $[SND.UNA - (2^{31} - 1)$, $SND.UNA - MAX.SND.WND)$, denoted as the *challenge ACK window*. Even though ACK numbers inside this window are still considered invalid, the specification requires the receiver to generate outgoing challenge ACKs in response to packets with such ACK numbers. Overall, this more stringent ACK number check does not eliminate, but helps dramatically reduce the probability that invalid data is successfully injected. Specifically, if the $MAX.SND.WND$ is small (which is typically the case for most connections), then the acceptable ACK window will be much smaller than the half of the ACK number space (as illustrated in Figure 3).

2.4 ACK Throttling

In general, as explained earlier, RFC 5961 enforces a much stricter check on incoming TCP packets; for example, it requires the RST packets to have an exact sequence number to actually reset the connection, whereas a "good enough" in-window value only triggers a challenge ACK. In order to reduce the number of challenge ACK packets that waste CPU and bandwidth resources, an ACK throttling mechanism is also proposed. Specifically, the system administrator can configure the maximum number of challenge ACKs that can be sent out in a given interval (say, 1 second). The RFC clearly states "An implementation SHOULD include an ACK throttling mechanism to be conservative." Therefore, the Linux kernel has faithfully implemented this feature by storing the challenge ACK counter in a global variable shared by *all* TCP connections. This approach, unfortunately, creates an undesirable side channel, as will be elaborated. We emphasize that the RFC states that ACK throttling applies to only *challenge ACKs* and not to regular ACKs. This means that the challenge ACK counter is unlikely to be affected by legitimate ACK traffic as the conditions that trigger challenge ACKs are all considered rare or due to attacks.

3 Vulnerability Overview

The Linux kernel first implemented all the features suggested in RFC 5961, in version 3.6 in September 2012. The changes were backported to certain prior distributions as well. The ACK throttling feature is specifically implemented as follows: a global system variable `sysctl_tcp_challenge_ack_limit` was introduced to control the maximum number of challenge ACKs generated per second. It is set to 100 by default. As this limit is shared across all connections (possibly including the connections established with the attacker), the shared state can be exploited as a side channel.

Assuming we follow the threat model in Figure 1, the basic idea is to repeat the following steps: 1) send spoofed packets to the connection under test (with a specific four-tuple), 2) create contention on the global challenge ACK rate limit, i.e., by creating a regular connection from the attacker to the server and intentionally triggering the maximum allowed challenge ACKs per second, and 3) count the actual number of challenge ACKs received on that connection. If this number is less than the system limit, some challenge ACKs must have been sent over the connection under test, as responses to the spoofed packets.

Depending on the types of spoofed packets sent in step 1, the off-path attacker can infer 1) if a connection specified by its four-tuple exists; 2) the next expected sequence number (*RCV.NXT*) on the server (or client); 3) the next expected ACK number (*SND.UNA*) on the server (or client). It is intriguing to realize that the three information leakages are enabled by the three (and only three) conditions that trigger challenge ACKs as described in §2.1, §2.2, and §2.3, respectively.

We elaborate below, the intuition on how the inference can be done in each case.

Connection (four-tuple) inference. Figure 4 shows the sequence of packets that an off-path attacker can send to differentiate between the cases of (i) the presence or (ii) the absence of an ongoing connection. In both cases, the attacker sends the same sequence of packets. Dashed lines represent packets with spoofed IP addresses. In the figure, the initial SYN-ACK packet is spoofed so that it appears to come from the client. The counter for the number of challenge ACKs that can be issued (100 initially) is tracked and depicted on the timeline of the server.

The hope is that the initial spoofed SYN-ACK packet will hit a correct four-tuple that corresponds to an active connection between the client and the server. In such a case (the left of Figure 4) the server will reply with a challenge ACK[2] (in accordance with the countermeasure proposed to defend against blind SYN packet injection as described in §2.1). At the same time, this will reduce the global challenge ACK count from 100 to 99. In the case where the spoofed SYN-ACK does not hit a correct four-tuple (on the right of the figure), the server will simply reply with a RST back to the corresponding client (as per TCP standards).

The attacker will then send 100 non-spoofed in-window RST packets to exhaust the challenge ACK count (this behavior is described in §2.2). In the *active connection* case, since the challenge ACK count is 99, the attacker can now observe only 99 challenge ACKs. In the *no connection* case, the attacker can observe 100. The difference in the number of challenge ACKs effectively leaks the information about whether a tested four-tuple corresponds to an active connection or not.

Sequence number inference. Assuming the attacker has already identified a four-tuple that corresponds to an active connection between the client and server, the off-path attacker now needs to guess a valid sequence number that is considered acceptable by the server. Figure 5 shows the sequence of packets that an attacker can send to distinguish between the cases of (i) *in-window* and (ii) *out-of-window sequence number*. In the first case where the spoofed RST packet has an in-window sequence number (but not the next expected sequence number), as per the countermeasure proposed to defend against blind RST packet injection as described in §2.2, a challenge ACK is triggered and this reduces the global challenge ACK count from 100 to 99. In the second case where the sequence number falls outside of the window, no challenge ACK will be generated (the global challenge ACK count remains at 100).

Similar to connection inference, the attacker will now send 100 non-spoofed in-window RST packets to exhaust the challenge ACK count. Once again, based on how many challenge ACKs are received, the attacker can tell if the guessed sequence number in the spoofed RST, is in-window or out-of-window.

ACK number inference. After an in-window sequence number of an active connection is identified, the attacker now will need to guess a valid ACK number that

[2] The effect is the same as sending a spoofed SYN. However, sending a SYN-ACK is generally more stealthy.

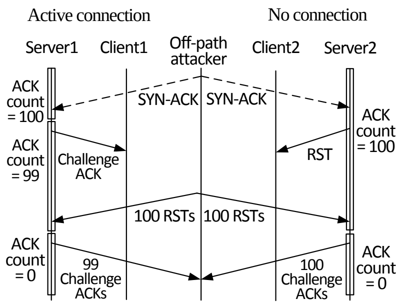


Figure 4: Connection (four-tuple) test

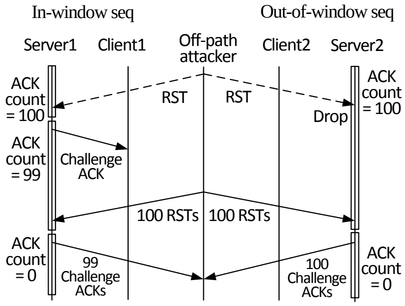


Figure 5: Sequence number test

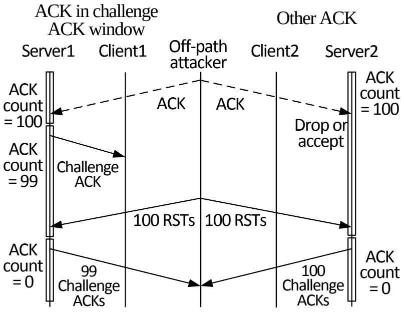


Figure 6: ACK number test

is considered acceptable by the server. Figure 6 shows the sequence of packets that an attacker can send to differentiate the cases of (i) *ACKs in challenge ACK window* and (ii) *other ACK numbers*. In the first case where the spoofed ACK packet has an ACK number in challenge ACK window (but with an in-window sequence number), the server will reply with a challenge ACK, in accordance with the countermeasure proposed to defend against blind data packet injection (as described in §2.3). Following the same procedure as before, an attacker can infer if the guessed ACK number falls in the challenge ACK window. As will be described in §5.2, this helps the attacker to eventually identify the *SND.NXT* on the server.

It is worth noting that once both the sequence number and ACK number acceptable by the server are inferred, an attacker can determine the sequence number and the ACK number acceptable by the client as well. This is because the *RCV.NXT* and *SND.NXT* on the server are basically equivalent to *SND.NXT* and *RCV.NXT* on the client [25, 18]. In practice, if the victim connection has ongoing traffic, the inferred sequence and ACK number may shift as the attack is in progress. We discuss such cases in §6.

An alternative approach for sequence number inference. In some cases a large number of RST packets observed in a short period time may be considered abnormal. Firewalls may even rate limit RST packets on a per-connection basis. In order to alleviate this, one can in fact replace RST packets with ACK packets, which are likely to stay under the radar. As shown in Figure 3, a challenge ACK will be sent when ACK number is in *challenge ACK window* while sequence number is in-window. Since the *challenge ACK window* space is at least $1/4$ of the entire 4G of the ACK number space, one can send 4 packets with ACK numbers 0, 1G, 2G, and 3G respectively and at least one packet will trigger a challenge ACK if the guessed sequence number is in-window. To understand why the *challenge ACK window* is at least this large, we first point out that the maximum receive window size is 1G with the TCP window scaling option (RFC 7323), which means that SND.MAX.WIN cannot be larger than 1G. Therefore, according to definition of the challenge ACK window described in §2.3, it is at least 1G as well. Given this, every spoofed RST packet sent earlier for sequence number inference is replaced by four ACK packets, which is less efficient but still effective. We have implemented and tested this alternative approach for sequence number inference. However, to simplify the description, we assume the use the original sequence number inference with RST packets in the subsequent sections.

4 Off-Path Connection Reset Attack

In the previous section, we illustrate how the global challenge ACK rate limit can theoretically leak information about an ongoing connection to an off-path attacker. In this section, we focus on how to construct a practical off-path connection reset attack that succeeds when a spoofed RST arrives with a matching sequence number of *RCV.NXT*. This requires an attacker to successfully carry out both *connection (four-tuple) inference* and *sequence number inference*. As will be discussed, to construct a realistic attack, several practical challenges need to be overcome. We assume the threat model to be the one in Figure 1 throughout the section, but the attack works with the alternative threat model (Figure 2) as well.

Goals and constraints. The main goal of the attack is to *quickly* and *reliably* conduct the sequence number inference and use it to reset an ongoing connection. The faster the attack succeeds, the more potent the DoS effect will be. However, the extent of the effect is subject to two practical constraining factors: (i) The bandwidth may be limited between the attacker and the victim (either server or client). (ii) Packet loss may occur between the attacker and victim, especially when they are far away. In this section, we focus only on designing fast probing schemes with given bandwidth constraints and leave the strategy to deal with packet loss to §6.

4.1 Time Synchronization

Challenge: As mentioned in §3, the challenge ACK rate limit is on a per second basis. In other words, the counter for the number of challenge ACK packets that can be issued, gets reset each second. Therefore, it is critical that in each cycle, all the spoofed and non-spoofed packets sent from the attacker arrive within the same 1-second interval, at the server.

One naive solution is that the attacker sends all those packets in a very short period (say, 10 ms), to ensure that the likelihood that they arrive within the same 1-second interval is high. Unfortunately, in practice, this solution does not work well since (i) many factors influence packet delays and thus, the gaps between packet arrival times at the receiver, might be much larger than the gaps in their transmission times, (ii) such bursts of traffic are likely going to experience congestion and packet loss. Thus, it is best for the attacker to synchronize with the clock on the server, so that the attacker can spread the traffic over the 1-second interval, without worrying that some packet arrivals may cross the boundary between two 1-second intervals.

The most common way to synchronize time between two machines is using the Network Time Protocol (NTP). But in practice, the attacker does not know if the server uses NTP, or to what NTP server it connects to; thus, it is not a reliable solution.

Solution: We propose a time synchronization strategy based on the very side channel introduced by the challenge ACK rate limit. The idea is to send more than 200 in-window RST packets spread out evenly in one second and check if we can see more than 100 challenge ACKs; if so, this indicates that we have crossed the boundary between two one second intervals (and have therefore not synced with the server yet). We then adjust the timing for next round of probing (shift it just enough) until we receive exactly the 100 challenge ACKs; in this case, we have succeeded in synchronizing with the server clock.

The reason we choose 200 packets is two-fold: 1) We are able to trigger at most 200 challenge ACKs no matter how many RST packets we send. These 200 challenge ACKs are triggered only when half of the RST packets arrive before the start of a new 1-second interval and half arrive after. 2) By evenly spreading the 200 packets over a 1-second window, i.e., sending one packet every 5ms, allows us to adjust the timing of the next round probing with the finest granularity. Specifically, we show that the time synchronization can be done in at most three rounds of probing in an ideal case (without packet losses).

Round 1: As described before, the attacker sends 200 in-window RST packets to the server evenly spread out over a 1-second window. The attacker then listens and counts the number of received challenge ACK packets. This value is stored as n_1. Here, the attacker listens for incoming packets for 2 seconds conservatively, before sending any additional packets to make sure a 1-second interval on the server has elapsed. Note that apart from the 200 RST packets, no other packet is sent to the server in this interval. If n_1 equals 100, it means that all 200 RST packets all arrive in the same 1-second interval on the server, thereby indicating that we have already synchronized with the server. Otherwise, it must be true that $n_1 > 100$, in which case the attacker proceeds to the next round.

Round 2: The attacker waits for 5ms (shifting the start time of the probes by 5ms) and repeats the same process as in the first step. The number of received challenge ACK this time is stored as n_2. If n_2 equals 100, the synchronization is done. Otherwise, the attacker proceeds to round 3.

Round 3: By comparing n_1 with n_2, the attacker can determine the final move to be synchronized. Specifically, we provide the following reasoning to support the decision. Assume that in step 1, x RST packets arrive in the first 1-second interval on the server, and y RST packets arrive in the second 1-second interval; note that $x + y = 200$. Similarly, in step 2, there are $(x-1)$ and $(y+1)$ RST packets arrive in the first and second 1-second intervals respectively, since in step 2 the attacker time shifts its probes by a period of 1 sub-interval. Thus, $n_1 = min(x,100) + min(y,100)$ and $n_2 = min(x-1,100) + min(y+1,100)$.

(i) If $n_2 \geq n_1$: Let us assume that $y \geq 100$ and $x \leq 100$; then $n_1 = min(x,100) + min(y,100) = x + 100$, and $n_2 = min(x-1,100) + min(y+1,100) = (x-1) + 100 < n_1$, which contradicts the assumption that $n_2 \geq n_1$; thus $y < 100$ and $x > 100$. With these conditions, $n_2 = 100 + (y + 1) = 100 + (200 - x + 1)$, or $(x-1) = 300 - n_2$. In step 2, $(x-1)$ RST packets arrive in the first 1-second interval on the server; thus, the attacker has to wait for $(x-1)$ sub-intervals, i.e., $(300 - n_2).\frac{1}{200}$ seconds to synchronize her time interval with the server.

(ii) If $n_2 < n_1$: With the same reasoning, the attacker knows that $x < 100$ and $y > 100$. In this case, $n_2 = (x-1) + 100$; thus, the attacker has to wait $(n_2 - 100)$ sub-intervals, or $\frac{n_2-100}{200}$ seconds to synchronize her time interval with the server.

If no packet loss occurs (which is likely due to the small number of packets sent every second), then the three rounds are enough to complete the synchronization process. To handle the rare event that packet loss may occur, we double check that the synchronization was successful by sending another round of 200 RST packets. If it is inconsistent with the previous round, we start over. As will be discussed later, such cases were almost never seen in our experiments.

Algorithm 1: Binary search for source port number

```
$left$ = left boundary of the port range
$right$ = right boundary of the port range
while $left < right$ do
    $mid = (left + right)/2$
    for $i = mid$ to $right$ do
        Send a spoofed SYN packet with $i$ as the client port number
    end for
    Send 100 RST packets on the legitimate connection
    Wait until the end of the 1-second interval, count the number of
    received challenge ACK packets
    if received ACK packets = 100 then
        $right = mid - 1$
    else
        $left = mid$
    end if
end while
return $left$; //the correct port value
```

```
if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq,
    ↪ TCP_SKB_CB(skb)->end_seq)) {
...
        goto discard;
}
if (th->rst) {
        if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt
            ↪ )
                tcp_reset(sk);
        else
                tcp_send_challenge_ack(sk, skb)
                    ↪ ;
        goto discard;
}
```

Figure 7: Logic of handling an incoming packet with RST flag in latest Linux kernels

4.2 Connection (Four-tuple) Inference

After time synchronization, the attacker can successfully launch subsequent attacks by knowing the boundaries between the 1-second intervals. The first step is "four-tuple inference", wherein the attacker determines if a connection is established between the client and the server. As mentioned in §2.1, the receiver will send back a challenge ACK (regardless of the sequence number of the packet) when a packet with a SYN flag set, arrives.

In §3, we discussed how this behavior can be exploited to determine whether or not a specific four-tuple is currently active. Basically, for each four-tuple in question, the attacker needs to send a spoofed SYN-ACK packet (a TCP packet in which the SYN and ACK flags are set) with <srcIP = clientIP, dstIP = serverIP, srcPort = X, dstPort = serverPort>. The above assumes both the client and server IP addresses are known. In addition, the server port is assumed to be publicly known according to its service type. Therefore, the only unknown is the source port the client uses. The maximum possible port range is $2^{16} = 65536$, and the default range on Linux is only from 32768 to 61000.

A naive approach is to test each port number at a time per second, as depicted in Figure 4, which, in the worst case, requires hours to complete. Therefore, a practical attack requires the attacker to test several port numbers in a second. Let us denote the maximum number of spoofed packets that can be sent in one second by n (constrained by network bandwidth). If n is large, one can search for the port number using a binary-search-like algorithm, the pseudo-code of which is shown in Algorithm 1. Specifically, assuming n is larger than 32767, in the first round the attacker can test the port range from 32768 to 65535 (the most likely half) in a 1-second interval. If the actual port number falls in the range, then the challenge ACK observed by the attacker at the end of the interval will be 99 (one goes to the victim). If the actual port number *does not* fall in this range, the observed number of challenge ACKs will be 100. In either case, the attacker can narrow down the search space by half and proceed to the next round of search.

An even better strategy is to divide the search space into multiple bins and probe them together in the same round. That way, one can eliminate $\frac{n-1}{n}$ of the search space. A similar multi-bin search strategy is used for sequence number inference in (§4.3).

In cases where n is smaller than 32767 (due to bandwidth constraints), the best the attacker can do is to simply try as many port numbers as possible in each round. The binary search or multi-bin search can be applied later when the search space becomes small enough.

4.3 Sequence Number Inference

As discussed in §2.2, the receiver generates a challenge ACK in response to a RST packet that contains an in-window sequence number which does not match exactly the expected value. The related Linux kernel code is shown in Figure 7; the `tcp_sequence()` function returns true if the sequence number is in-window, and false if it is out-of-window. In the latter case, the packet will simply be dropped. When the sequence number is in-window and the packet has the RST flag set, its sequence number is analyzed further. As we can see, the connection is terminated only when the sequence number matches *RCV.NXT*; otherwise, a challenge ACK is sent.

The main difference between port number inference and sequence number inference is that the attacker does not need to check every possible sequence number to trigger a challenge ACK. Therefore, the attacker can divide the sequence number space into blocks whose sizes are equal to the receive window size, and probe with a guessed sequence number in each block to determine which sequence numbers fall in the receive window. Theoretically, an attacker can apply the same binary

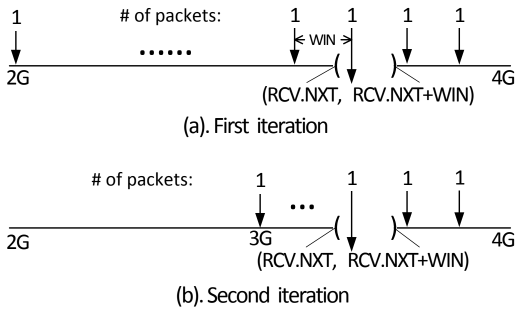
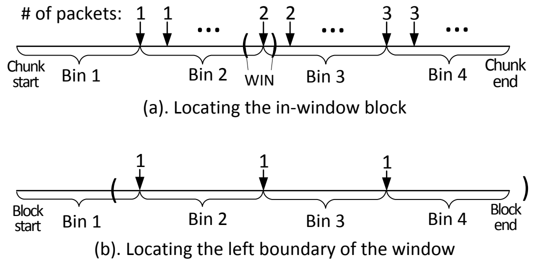


Figure 8: Binary search for sequence number illustration



# of packets: 1 1 ... 2 2 ... 3 3 ...
Chunk start
Bin 1
Bin 2
WIN
Bin 3
Bin 4
Chunk end
(a). Locating the in-window block
1
1
1
Block start
Bin 1
Bin 2
Bin 3
Bin 4
Block end
(b). Locating the left boundary of the window

Figure 9: Multi-bin search for sequence number illustration

search algorithm used in connection inference. This process is illustrated in Figure 8. In the first round of probing, the attacker can probe the right half of the sequence number space — (2G, 4G). If any of the spoofed RST packets triggers a challenge ACK, the attacker will observe less than 100 challenge ACKs at the end of the 1-second interval. If there are exactly 100 challenge ACKs observed, it indicates that the receive window is on the left side of the search space. In either case, in the second round, the attacker knows "which half the receive window belongs to." Let us say that the receive window is in the right half. The attacker would then divide the search space of (2G, 4G) into (2G, 3G) and (3G, 4G). Similar to the first round, only (3G, 4G) needs to be probed in order to determine the part that contains the receive window. This search will eventually stop after 32 rounds exactly (because the sequence number is 32-bit).

However, in practice, the sequence number search space is significantly larger than port number space. Let us consider a receive window size of 12600. This leaves the attacker 340870 possible blocks to search through. If the attacker were to transmit this many packets in one second, the bandwidth requirement would be around 150Mbps, which is extremely high. Likely, the attacker will have to perform a linear search by attempting to search as many blocks as allowed by bandwidth in one second.

Dealing with unknown window sizes: Ideally, the block size should be determined by the window size of the target connection, i.e., the server's receive window size. In reality, however, an *off-path* attacker cannot observe the window size. If the attacker chooses a smaller window size (compared to the actual window size), the attack will send more packets unnecessarily and take more time. On the other hand, if the guessed value is larger than the actual value, the attacker might miss the correct window of sequence numbers while traversing consecutive blocks. Thus, there is an inherent trade-off between the success rate and the cost incurred (in terms of time and bandwidth) of the attack. Even if the attacker can come up with a correct receive window size at one particular time, the size can change over time.

Our solution is to use a conservative estimate of the window size as the block size in the beginning and update it later given proper feedback. The conservative window size is determined by the initial window size advertised by the server in the SYN-ACK packet. By surveying Alexa top 100 websites, we find that the average initial receive window size is 26703. This window size is the lower bound as the window typically grows after the connection is established. To observe the initial window size, the attacker simply attempts to establish a valid (non-spoofed) TCP connection with the server. This strategy works because a server typically uses the same initial receive window size for all clients. Such a conservative estimate of window size may force the attacker to send more packets, but it at least will guarantee success. We will also discuss how to update the window size dynamically during the search process.

Next, we elaborate the design of sequence number inference:

- **Step 1 – Identify the approximate sequence number range.** Let us assume that the attacker, in *n* blocks, can send *n* spoofed packets per second (*n* is on the order of thousands in our experiments). We call such *n* consecutive blocks a chunk. The guessed sequence number is always chosen to be the first sequence number within a block. If at the end of the 1-second interval, the attacker observes 100 challenge ACKs, then the attacker proceeds to the next chunk, i.e., the next *n* consecutive blocks. If the attacker observes less than 100 challenge ACKs, it indicates that the receive window is within the chunk that was just probed. The attack can now proceed to step 2. Note that if the number of observed challenge ACKs is less than 99, it indicates that the initially estimated window size (block size) is too small.

For example, as illustrated in Figure 10(a), if there are two blocks whose beginning sequence numbers are inside the actual receive window, then the number of observed challenge ACKs will be 98; this indicates that the actual window size should be approximately twice the estimated window size (initial block size). We there-

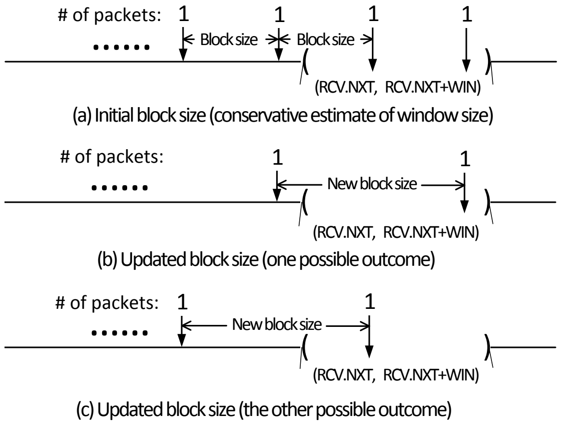


Figure 10: Window size estimate and adjustment

fore update the block size to be twice as much in the subsequent search steps. The two possible outcomes are shown in Figure 10(b) and Figure 10(c).

- **Step 2 – Narrow down the sequence number space to a single block.** From step 1, we know that the receive window is within a chunk. We now further narrow down the search space to an exact block within the chunk. Note that we have now updated the block size so that there will be one and only one block that can trigger challenge ACKs. To locate the exact block, the same binary search strategy outlined in Figure 8 can be used except that the search space now is dramatically reduced after step 1.

The located block has a beginning value which, is an in-window sequence number; therefore, one of the following is true: (i) its beginning value is the correct sequence value; or (ii) the correct sequence value is in its left neighboring block. In the first case, since the sequence number matches the $RCV.NXT$, the spoofed RST packet can already terminate the connection. In the second case, the attacker performs an additional search in the left neighboring block (see Step 3).

- **Step 2 (optimized version) – Identify the correct sequence block using multi-bin search.** With the previous assumption that the attacker can send n spoofed packets per second, with a binary search, the first round requires only $\frac{n}{2}$ packets (as we divide a chunk into two halves initially). The second round requires only $\frac{n}{4}$ packets and so on. As we see, the number of packets sent in each round reduces quickly. This is not an efficient use of the network bandwidth. We show that it is possible to speed up the search process by sending more packets per round (still at most n per round).

The idea is, instead of dividing the search space into two halves in each round, we can divide the space into multiple bins and probe them simultaneously. This is illustrated in Figure 9(a) where 4 bins are present in a chunk. Each bin here holds an equal number of blocks. To determine which bin the receive window falls in, the attacker sends a different number of spoofed RST packets in each bin. In the example, he sends 1 RST packet per block in the 2nd bin, 2 RST packets per block in the 3rd bin, and 3 RST packets per block in the 4th bin. Since the receive window can fall into one and only one of the bins, the attacker can determine which bin it is in, by observing how many challenge ACKs are received at the end of the 1-second interval. If there are 100 challenge ACKs received, it indicates that the receive window is in the 1st bin (since no RST packets were sent in the 1st bin). Receipt of 99 challenge ACKs indicates that the receive window is in the 2nd bin, etc.

Note that the more bins we have, the faster we can narrow down the sequence number space. However, the number of bins chosen for each round is constrained by n. The larger the n, the more the bins that can be created. The number of bins is also capped at 14, given that the number of spoofed packets may already exhaust the 100 challenge ACK counter in one round $(0+1+2+...+13=91)$.

- **Step 3 – Find the correct sequence number using binary search.** Now we are sure that $RCV.NXT$ is within a specific block, we need to locate its exact value. To achieve the goal, another modified binary search strategy is used here. The observation is that the correct sequence number ($RCV.NXT$) is the highest value in the block, such that any spoofed RST packet with a sequence number less than it will not trigger a challenge ACK packet. It is worth noting that we may not realize which value is the correct sequence number until the connection is terminated, as all the probing packets are RST packets.

- **Step 3 (optimized version) – Find the correct sequence number using multi-bin search.** Similar to the previous multi-bin search, the attacker can divide the single block into many small bins and probe them simultaneously. All bins before the left boundary of the receive window ($RCV.NXT$) will not trigger any challenge ACKs; the ones after will. Thus, in this step attacker only sends one spoofed packet per bin and accumulates all the challenge ACKs received from right to left (See Figure 9). If the attacker sees (100-X) challenge ACKs at the end of the 1-second interval, it indicates that X probed bins are after $RCV.NXT$. In Figure 9, let us say we divide the block into 4 bins. After probing them, the number of observed challenge ACK will be 97 because 2nd, 3rd, and 4th bins turn out to be after $RCV.NXT$. Note that if the observed challenge ACK is 100, it indicates that the correct sequence number is somewhere inside the 4th bin (but not its beginning value).

Similar to the previous multi-bin search, the number of bins chosen for each round is constrained by n. In addition, the number of bins is always capped at 100, as the spoofed packets may exhaust the limit of 100 challenge ACK count.

The RST off-path TCP attack is successfully launched after the above three steps. The exact number of probing rounds depends on the available bandwidth, and will determine the time it takes to finish the attack. We will evaluate this in §7.

5 Off-Path Connection Hijacking Attack

In this section, we discuss how an off-path attacker can hijack an ongoing connection and inject spoofed data. The methodology used to inject data into the client or to the server are similar; thus, without loss of generality, we exemplify the attack targeting the server. First, we describe the challenges that the attacker will need to overcome; subsequently, the entire attack process is described in detail.

5.1 Challenges and Overview

The attacker will experience obstacles that are similar to those associated with launching an off-path reset attack. In addition, the following additional challenges need to be addressed.

Preventing unwanted connection reset. As described in §4.3, the RST packets with in-window sequence numbers are leveraged towards identifying the next expected sequence number on the connection. However, with that process, sending a RST packet with the exact, expected sequence number ($RCV.NXT$) to the server will terminate the TCP connection; this is not the goal of the hijack attack. The challenge is thus, to infer $RCV.NXT$ without causing connection termination.

Identifying both the sequence number and ACK number. In order to trick the server into believing that the injected data is valid, and sent from the server, the attacker needs to know both the correct sequence number ($RCV.NXT$) and the acceptable ACK range on the server side of the connection. The latter is typically a fairly small range as discussed in §2.3.

At a high level, our design of the attack consists of the following steps: First, the attacker finds an in-window sequence number on the server using the techniques described in §4.3. Based on this, the attacker will be able to guess the range of acceptable ACK values that trigger challenge ACKs. The range of these acceptable values (ACK window) can be used to identify the highest acceptable ACK number, i.e., $SND.NXT$, on the server. We will show next that obtaining this ACK number then allows the attacker to infer the exact expected sequence number on the server without resetting the connection.

5.2 Inferring Acceptable ACK Numbers

Assuming an in-window sequence number is already inferred, we now discuss how an attacker can infer the next ACK number, $SND.UNA$, which is expected by the server. As illustrated in Figure 3, an incoming data packet is accepted if the ACK number is in the range of $[SND.UNA - MAX.SND.WND, SND.NXT]$. If not, the receiver will respond with a challenge ACK packet, if the ACK number is in the range of $[SND.UNA - (2^{31} - 1), SND.UNA - MAX.SND.WND)$; this range is called the *challenge ACK window*. It is obvious that $SND.UNA$ can be computed if one can successfully infer the left boundary of the challenge ACK window, $SND.UNA - (2^{31} - 1)$. This in turn can be found using the following approach.

Step 1: Identify the challenge ACK window position. According to RFC 1323, by using the window scaling option, the maximum receive window size can be extended from 2^{16} to a maximum of $2^{30} = 1G$. Thus, the $MAX.SND.WND$ cannot be larger than 1G. Accordingly, the *challenge ACK window* size is between 1G and 2G, which is one quarter of the entire ACK space size. Because of this, we divide the entire ACK space into 4 bins and probe each bin to check which bin(s) the *challenge ACK window* falls in. In our implementation, we probe the first value of each bin, i.e. 0, 1G, 2G, 3G. We know for certain that either one or two bins can trigger challenge ACK packets. Therefore, we need to send different number of packets for each bin to differentiate the resulting cases. A simple strategy is to send one packet at ACK number 0, two packets at 1G, four packets at 2G, and 8 packets at 3G. For instance, if the number of observed challenge ACKs is 94 (6 missing), then we can infer that both ACK number 1G and 2G have triggered challenge ACKs. If the number of observed challenge ACKs is 96 (4 missing), then only ACK number 2G has triggered challenge ACKs. We can then easily determine the "left-most" bin whose beginning value falls in challenge ACK window.

Step 2: Find the left boundary of the challenge ACK window Now the problem is, given the bin located in the previous step, we need to identify an ACK number in the left neighboring bin, such that it is the "left-most" value (in the circular sense) that can still trigger challenge ACKs. This is a problem that can be solved in a similar way to the last step of sequence number inference using multi-bin search (§4.3).

Finally, when the left boundary of the challenge ACK window ($SND.UNA - (2^{31} - 1)$) is found, an acceptable ACK value ($SND.UNA$) is trivially computed.

5.3 Identify the Exact Sequence Number

To locate $RCV.NXT$ without resetting a connection, we leverage the knowledge learned about the various ACK number ranges. The idea is that, instead of sending spoofed RST packets (which may terminate a connection), the attacker can send spoofed *data* packets with ACK numbers that fall in the challenge ACK window and thus, intentionally trigger challenge ACKs (if the sequence number is in-window). Combined with the fact no challenge ACK will be triggered if the guessed sequence number is before *RCV.NXT* (considered old packet and dropped), *RCV.NXT* can be located as the "left-most" value that can trigger challenge ACKs. The search process is in fact similar to the last step in sequence number inference except that we now use spoofed data packets.

Now that the attacker knows both the *RCV.NXT* and *SND.UNA* on the server, it is trivial to inject legitimate-looking data packets that will be accepted by the server. Further, it is also trivial to inject legitimate-looking data packets to the client because the *RCV.NXT* on the server is effectively the *SND.UNA* on the client, and the *SND.UNA* is the *RCV.NXT* on the client (assuming no traffic is in flight). In §7.2, we will present a case study on how a web service can be hijacked by a completely blind off-path attacker.

6 Other Practical Considerations

We have fully implemented the attacks described in §4 and §5. In §7, we will evaluate the effectiveness and efficiency of the attacks extensively. In this section, we outline a few practical considerations that need to be handled.

Detecting and handling packet loss. So far, we have assumed that spoofed connections will not incur packet loss and the challenge ACK side channel has no noise. However, in reality, even if the number of packets sent per second is chosen conservatively (well below bandwidth constraints), there is still no guarantee that packet loss will not occur, and a host may legitimately generate challenge ACKs that are not triggered by the attack. They exhibit the same effect to the attacker — the number of observed challenge ACKs will be smaller than expected. In this paper, we call them both packet loss for convenience. We address packet loss based on the two following principles: 1) when in doubt, repeat the probes; 2) add redundancy in the probing scheme to proactively detect packet loss.

In the initial step of the sequence number search, if packet loss occurs, the number of observed challenge ACKs may reduce to 99; the attacker thus, may incorrectly conclude that a chunk that contains the receive window is located. This will affect all subsequent search steps. Therefore, every time when a "plausible" chunk is detected, we repeat the probe on the same chunk. The search will proceed to step 2 only when both rounds return exactly 99 challenge ACKs (no more, no less).

In step 2 and step 3 of the sequence number search, we add redundancy to actively detect packet loss so that we repeat only the round of probing that experienced packet loss. The idea is similar to using parity bits. In each round, instead of allowing the number of observed challenge ACKs to be any value equal to or below 100, we can construct the probing packets such that only odd number of challenge ACKs will be considered a valid outcome. If an even number of challenge ACKs is received, packet loss must have happened. This strategy can be visualized by referring to Figure 9(a). Instead of sending 1, 2, or 3 packets per block for each bin, we will send 1, 3, and 5 packets per block for each bin. This means that if the receive window falls in 2nd bin, the number of challenge ACKs will be 99; if the receive window is in 3rd bin, the number of challenge ACKs will be 97, etc.

Both schemes are implemented and shown to be very effective in cases where the network conditions between the attacker and the victim are poor.

Moving receive window and challenge ACK window. So far, we have assumed that the connection is relatively idle, and the window does not change while the inference is in progress. This is likely to be the case in many real world scenarios, especially with long-lived connections. One example is the push notification connections on mobile platforms [2]. They are idle most of the time until a new push notification arrives. Even when a connection is not idle at one point, it is likely to become idle at some point and become more susceptible to the attack. Moreover, the traffic activity will mostly be concentrated on either uplink and downlink, rarely both. Typically, downlink traffic dominates; therefore, the attacker targeting at resetting the connection on the server side will experience less difficulty (client's sequence number increases very slowly). Tor network connections are also candidates as the end-to-end throughput is typically very low.

To support sequence number inference against (slow) moving receive windows, we implement a simple strategy which conducts a brute-force style sequence number guessing. Specifically, once a "left-most" in-window sequence number is inferred (which may become invalid in the next interval due to the ongoing activities), we send 20,000 RST packets with sequence numbers, with offset 1, 2, ..., 20,000 to the valid sequence number. As will be shown in §7.1.2, for low-activity connections, this strategy works well. We leave the exercise to come up with a strategy to target connections with heavier traffic to future work.

Per-connection rate limit. Since the Linux kernel version 4.0 (released in Apr 2015), in addition to the global challenge ACK rate limit, a per-connection rate limit was introduced. The idea is to reduce the impact of potential ACK loops [3] that may occur if client and server are de-synchronized. Theoretically, the per-connection rate limit provides an isolation between the victim connection and the attacker connection, and the side channel should be eliminated completely. For instance, even if the challenge ACK count limit is reached for the victim connection, it does not affect the limit on the attacker connection at all.

However, interestingly, the per-connection rate limit only applies to SYN packets or packets without any payload. The comment in the Linux kernel states "Data packets without SYNs are not likely part of an ACK loop", hinting that such packets do not need to be governed by the per-connection rate limit. It is evident that the developers assumed a benign scenario instead of an adversarial one. To get around this restriction, we simply send spoofed packets with a single byte of payload. For the spoofed SYN-ACK packets though, it is impossible to bypass the per-connection rate limit. Unfortunately, upon a closer look at the implementation, when a per-connection challenge ACK is sent out, it is also counted towards the global challenge ACK limit. Therefore, it is still possible to infer that the four-tuple of an ongoing connection has been guessed correctly by observing only 99 challenge ACKs at the end of the 1-second interval. In practice, the per-connection rate limit is 1 packet every 0.5 second, which does allow the attacker to proceed with the binary search approach outlined in §4.2. We have verified experimentally that it does work against the latest Linux kernels with per-connection rate limit.

Configurable maximum challenge ACK count. For simplicity, throughout the paper, we assume the challenge ACK count to be 100, which is the default value. Our test on a variety of Linux operating systems also confirmed the result. However, as proposed in RFC 5961, this value is configurable by a system administrator. According to the specification, the flexibility is provided to allow the tradeoff between resource (bandwidth and CPU) utilization and how fast the system cleans up stale connections. Fortunately, the exact configured value can be inferred quite easily with some simple steps (as long as it is not excessively large). After establishing a legitimate connection to the server, the attacker can send many RST packets, e.g., 1000 packets which is much larger than default value of 100, with in-window sequence values to trigger as many challenge ACKs as possible. The packets are sent in a very short period of time (say, 100 or 200 ms) to increase the likelihood that they end up in the same 1-second interval. The attacker then counts the total number of challenge ACKs returned. Finally, the attacker can wait for a short amount of time and repeat the process one more time to verify the number of received challenge ACK packets is the same; that value would be the actual limit set by the server. Note that this is only a one-time effort for each target.

7 Evaluations

To showcase the effectiveness of our attacks, we next evaluate them in terms of metrics such as success rate and the time to succeed.

7.1 Connection Reset Case Studies

There are two sets of experiments reported in this section viz., where (i) we reset an SSH connection and (ii) perform a Tor connection reset.

Experimental setup. For the SSH experiments, we use a Ubuntu 14.04 host on the University of California - Riverside campus as the victim client. The victim SSH server is one of the instances we create on Amazon EC2 in different geographic locations, worldwide. The attack machine is a Ubuntu 14.04 host in our lab. For the Tor experiments, we target the connection between a Tor relay (set up in our campus) and a random peer relay. Our Tor relay is also a Ubuntu 14.04 host and has been running the service for several months. The attack machine is the same host as the one in the SSH experiments.

In both the SSH and Tor experiments, the attacker attempts to reset the connection on the server end by connecting to it and performing the inference attacks. The diversity of servers and the corresponding network paths help test the robustness of the attack. We assume that the 3-tuple <client IP, server IP, and server port> is known. Further, the attack machine is capable of spoofing the IP address of both the victim client and server.

7.1.1 SSH Connection Reset

Location	Success Rate	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time Cost (s)
US West 1	10/10	0	0	5000	48.00
US West 2	9/10	1.0	1.91%	5000	58.00
US East	10/10	0	0	5000	32.00
EU German	9/10	0.3	0.67%	5000	48.00
EU Ireland	10/10	0	0	5000	35.20
Asia 1	10/10	0	0	5000	51.00
Asia 2	9/10	1.7	5.34%	5000	36.67
South America	10/10	0	0	5000	45.70

Table 1: SSH connection reset results

Summary. We run the reset attack against 8 different Amazon EC2 servers in different geographical locations. They are all micro instances set up for our experiments only. We establish a connection from the victim client to each server, and have the attacker perform the off-path connection reset attack. For each server, we repeat the experiment 10 times and report the average. As shown in Table 1, the attack is highly effective: the average success rate is 97% over all runs, with an average time cost of 44.3s. Note that the overall time excludes the time for synchronization (recall §4.1) as it is a one-time effort for a server and can be done a priori. The bandwidth cost here is 5000 spoofed packets per second, which translates to 4Mbps. Note that the probing scheme has already built in packet loss detection using "parity bits" as described in §6. To show that the packet loss detection scheme works, we report the number of rounds and the percentage of rounds on average, when packet loss is detected. For instance, even when packet loss between the attack node and "Asia 2" server is frequent, we still manage to succeed 9 times out of 10.

Failures may still occur since the detection scheme is rudimentary and may fail to detect packet loss. In some cases, the failure can also be the result of the attacker and server becoming out-of-sync due to network delay variance. The success rate can be further improved by adding more redundancy and using better error detection schemes. However, we argue that the current success rate is already good enough to carry out effective DoS attacks.

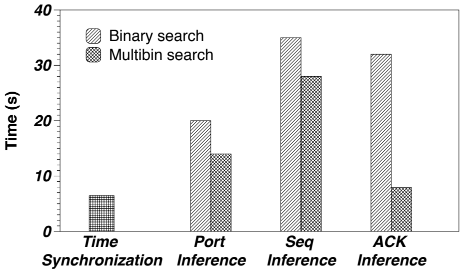


Figure 11: Time breakdown

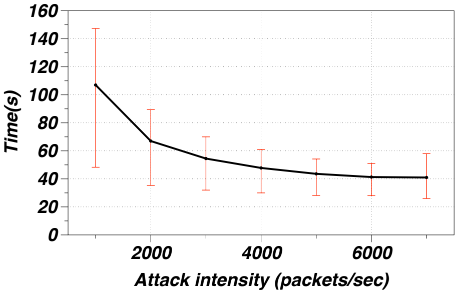


Figure 12: Attack intensity impact on time to succeed

Time breakdown. To understand where the time is spent in our attacks, we conduct another benchmark experiment against one of the SSH servers with both sequence number and ACK number inference. As shown in Figure 11, we break down the time spent into time synchronization and the three search phases of port number inference, sequence number inference, and ACK number inference. We also compare the optimized multi-bin search versus the regular binary search in each phase. Time synchronization takes around 7 seconds (optimization is not applicable). As discussed, it is only a one-time effort and therefore not on the "critical path". We see that with the optimized multi-bin search, the time spent on port search is fairly short (around 14 seconds). The time spent on sequence number search takes the most time due to the fact that the sequence number space is much larger. The time spent on ACK number inference is also fairly short (around 8 seconds) due to the fact that the challenge ACK window is extremely large and easy to locate.

Compared to the results with binary search, we see that the optimized multi-bin search has greatly improved the search speed by more than 30 seconds overall. This is due to the fact that binary search significantly underutilizes the bandwidth resources and significantly increases the number of rounds of probes. The reason why the sequence number search does not benefit as much is because most of the time is spent on the initial linear search of the huge sequence number space. This step cannot be optimized with the multi-bin search.

Attack intensity vs. Time to succeed. Using the same experimental setup as before, we vary the attack intensity, i.e., the number of packets sent per second and show how this affects the time it takes to succeed. As shown in Figure 12, we plot the average, min, and max time to successfully conduct sequence number inference only (reset attack), as well as with the ACK number inference added (hijacking attack). Clearly, the higher the attack intensity the faster the attack. When the intensity is only $\approx$ 512 Kbps (1000 packets per second), the time to succeed is over 100 seconds, on average. When the intensity is $\approx$ 4 Mbps, (5000 packets per second), the average time reduces to $\approx$ 50 seconds for hijacking and only 30 seconds for reset. Note that an intensity $>$ 4 Mbps does not substantially improve the time to succeed because we begin to observe more packet losses, which cause additional rounds of probing. Of course, this is experienced on the specific network environment between the attack host and the server, which could differ elsewhere; if the network conditions are even better, the time to succeed can be further improved.

7.1.2 Tor Connection Reset

Node	Target	Success Rate	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time Cost(s)
62.210.x.x	FR	8/10	1.9	4.58%	4000	46.36
89.163.x.x	DE	9/10	4.0	7.97%	4000	49.08
178.62.x.x	GB	7/10	3.2	4.20%	4000	53.00
198.27.x.x	NA	10/10	0.8	1.45%	4000	59.86
192.150.x.x	NL	8/10	4.1	5.64%	4000	68.03
62.210.x.x	FR	6/10	2.5	5.85%	4000	49.57
89.163.x.x	DE	8/10	1.7	3.06%	4000	52.51
178.62.x.x	GB	8/10	6.0	8.15%	4000	78.35
198.27.x.x	NA	7/10	2.1	3.64%	4000	72.49
192.150.x.x	NL	6/10	5.5	7.14%	4000	79.42

Table 2: Tor connection reset results (first half under browsing traffic and second half under file downloading traffic)

To conduct a realistic experiment, we use a Tor relay set up in our campus and have a user using it as an entry relay. The entry relay establishes connections with an arbitrary middle relay (anywhere in the world). For ethical reasons, we do not perform attacks against arbitrary relay nodes that are not connected to our node.

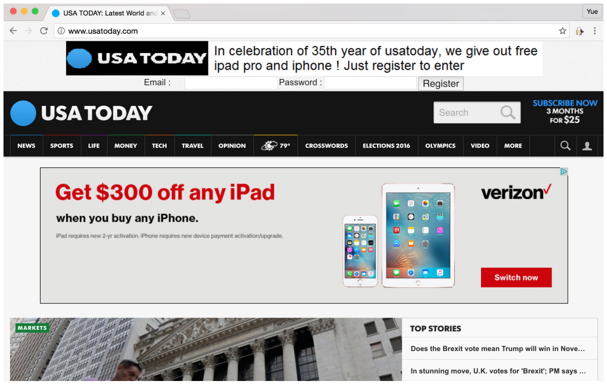
To understand how the attack performs against mostly idle connections, we test it against connections between our own Tor relay and 40 other Tor relays throughout the world. The attack node has to connect to these Tor relays that are far away to perform attacks. In each case, we repeat the reset experiment 10 times. First, we discover that 16 of them do not appear vulnerable to the side channel attacks, even though they appear to be Linux hosts. We suspect that this is because of certain firewalls that drop our spoofed packets. For the remaining 24 hosts, the average success rate is 88.8% and the average time to succeed is 51.1s. We find these results to be slightly worse than those in the SSH experiments because of higher packet loss rates.

In addition, we pick 5 random relays and simulate background traffic with browsing and file downloading, and conduct the same experiment as above. Here, to deal with moving windows, we use the simple brute-force strategy described in § 6. The results are shown in Table 2. The average success rate is now down to 77% and the average time to succeed is 60.9s. Upon further inspection, the increased failure rate is exactly due to the moving window problem i.e., it interferes with the sequence number search. Nevertheless, we think the result is acceptable as we have not designed a robust solution specifically for dealing with a moving window (this is left for future work).

In general, we believe that a DoS attack against Tor connections can have a devastating impact on both the availability of the service as a whole and the privacy guarantees that it can provide. The default policy in Tor is that if a connection is down between two relay nodes, say a middle relay and an exit relay, the middle relay will pick a different exit relay to establish the next connection. If an attacker can dictate which connections are down (via reset attacks), then the attacker can potentially force the use of certain exit relays.

7.2 TCP Hijacking Case Study

Our attack does not require any assistance from client-side or server-side malware or puppet (which are required in prior studies [23, 14]). Therefore, our target is any long-lived TCP connection that does not use SSL/TLS. There are several attractive targets: video, advertisements, news, and Internet chat rooms (e.g., IRC). Depending on the implementation, one can envision the following possibilities: 1) the client periodically initiates a request and asks for responses, or 2) the server proactively pushes notification messages. In both cases, our attack can inject malicious messages to the client and induce a variety of classic attacks such as phishing or cross-site scripting.



USA TODAY: Latest World and

www.usatoday.com

In celebration of 35th year of usatoday, we give out free ipad pro and iphone ! Just register to enter

Email : Password : Register

Figure 13: USAToday screenshot with phishing registration window

Here, we pick a news website *www.usatoday.com* which has a long-lived TCP connection that periodically retrieves news updates every 30 seconds. This gives ample idle time for our sequence number and ACK number inference. The attacker machine and the victim client are Ubuntu 14.04 hosts in our lab (as in the other case studies). Once the numbers are inferred, we perform a desynchronization attack [4] by sending a spoofed request to the server that will force it to send a response to the client. Since the request was never sent by the client, it will not accept the response as the response packet contains an invalid ACK number (acknowledging data that have not been sent). Later, when the client itself initiates a real request, the server would no longer accept it as the packet is considered to be data with an old sequence number. Now that the client and server become de-synchronized, the attacker no longer needs to worry about a race condition where the response to the victim client is sent back by the server first. During all this, the attacker simply sends spoofed responses periodically every few seconds with ACK numbers properly acknowledging the client's requests. If such spoofed responses arrive before the client sends a request, they will simply be dropped without any adverse effect (because the ACK numbers are acknowledging data that has not been transmitted yet).

We implement the attack end to end, and successfully hijack the connection and inject a phishing registration window to solicit email and passwords at the top of the webpage as shown in Figure 13. We repeat the experiment 10 times and summarize our results in Table 3. The attack first infers sequence and ACK numbers before injecting the malicious payload. Success rate 2 quantifies the rate of inferring the sequence and ACK numbers correctly. However, USAToday occasionally switches the

Success rate 1	Success rate 2	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time Cost (s)
7/10	9/10	2.22	3.63%	5000	81.05

Success rate 1 = success rate of injecting the phishing registration window
Success rate 2 = success rate of inferring the correct sequence and ACK number

Table 3: USAToday injection results

HTTP request from one type to another and therefore the injected payload will not match the request. Success rate 1 quantifies the rate of injecting the response that matches the request, which is strictly lower than success rate 2, but is still reasonable in our experiments. In addition, the time to succeed is longer than in the case of SSH and Tor experiments mostly because of the extra steps of ACK number inference and data injection.

8 Discussion and Defenses

Vulnerabilities in other OSes: We examine if the studied vulnerability exist in the latest Windows and FreeBSD OSes (The latter TCP stack is also used by Mac OS X). In brief, these OSes are not vulnerable to the attack. First of all, neither Windows nor FreeBSD has implemented all three conditions that trigger challenge ACKs according to RFC 5961. More importantly, the ACK throttling is not found for Windows or MAC OS X. Ironically, not implementing the RFC fully, in fact is safer in this case.

Defenses. As highlighted earlier, the root cause of all the attacks described is the side channel associated with the global challenge ACK count. This side channel can leak various types of information about an ongoing TCP connection. In general, as asserted in previous studies [21], network protocols are not designed rigorously to guarantee the non-interference property. In our study, we discover that the design and implementation of RFC 5961 has actually introduced an information flow that leaks TCP connection state through the shared challenge ACK counter, and is highly exploitable.

The best defense strategy is to eliminate the side channel (the global challenge ACK count) altogether. One can still enable the per-connection rate limit as long as each connection has a completely separate counter that does not interfere with those of other connections. The downside of this strategy is that if the number of connections in a system increases, the aggregate challenge ACK count can go up without any bound. There is currently no evidence to suggest that this worst case scenario is likely to ever happen. However, if one is really concerned about wasting resources on sending challenge ACKs, we suggest a second solution which is adding noise to the channel. This is a common defense strategy in mitigating side channel attacks [10, 27]. Specifically, instead of having a fixed global challenge ACK count of 100 in all intervals, we can add random values (either positive or negative) for each interval. This will essentially confuse the attacker during the search process. In fact, even if the attacker repeats the probe many times, the result will always differ over time. To ensure that the added randomness is theoretically sound, one can even apply differential privacy to systematically introduce noise, as was done recently in [28]. We leave the design of the exact scheme to add randomness to future work. We also plan to propose the defenses to the Linux community.

9 Related Work

Previous work on off-path TCP sequence number inference heavily relies on executing malicious code on the client side [22, 23, 14, 16, 17, 1], either in the form of malware [22, 23] or malicious javascript [14, 16, 17]. They share the same scheme of "guess-then-check" based on some side channels observable by the malicious code on the client side. They include OS packet counters [22, 23, 9], global IPID [14, 1], and HTTP responses [16]. In contrast, our off-path TCP attack eliminates the requirement completely, which makes the attack much more dangerous. The only prior study that shares the same threat model is the one reported by lkm in phrack magazine in 2007 [1]. The authors exploit the well-known global IPID side channel on Windows hosts to perform such attacks. Unfortunately, the IPID side channel is extremely noisy and the attack can take close to 20 minutes to succeed, as reported by the authors. Furthermore, as reported in [14], the success rate of such an attack is very low, unless the attacker has a low latency to the victim (e.g., on the same LAN). In comparison, our newly reported attack finishes much faster and is significantly more reliable.

Besides the TCP sequence number, it has been shown that other types of information can be inferred by an off-path or blind attacker [12, 21, 11, 29, 5, 15]. For instance, Ensafi *et al.* [12] show that, by leveraging the SYN cache and RST rate limit on FreeBSD, one can infer if a port is open on a target host through bouncing scans off of a "zombie" FreeBSD host. Knockel *et al.* [21] demonstrate the use of a new per-destination IPID side channel that can leak the number of packets sent between two arbitrary hosts on several major operating systems with a bootstrapping time of an hour on average. Alexander *et al.* [5] can infer the RTT between two arbitrary hosts with reasonable accuracy within minutes. Gilad *et al.* [15] are also able to infer if two hosts have established a TCP connection identified by a specific four-tuple, by utilizing the same noisy global IPID side channel. Compared to the newly discovered side channel, it has the following limitations: 1) requires the presence of stateful firewall or NAT which may not be universally present; 2) has a low success rate even when the tests are repeated multiple times (e.g., for more than a minute). Utilizing the new side channel, we can do this much faster.

Many of the side channels can be abused and cause unwanted information leakage. However, in some cases, they can also be used legitimately for network measurements. For instance, the global IPID side channel has been used to infer a network's port blocking policy [24]. The same side channel has also been used to count how many hosts are behind a NAT [6]. In addition, even though commonly considered a vulnerability, ISPs that allow IP spoofing are still prevalent according to the latest reports in 2009 [7] and 2013 [8]. Further, very recently, IP spoofing has also been used in legitimate applications such as reverse traceroute [20], detecting Interdomain Path changes [19], and detecting routing policy violations [13].

10 Conclusions

To conclude, we have discovered a subtle yet critical flaw in the design and implementation of TCP. The flaw manifests as a side channel that affects all Linux kernel versions 3.6 and beyond and may possibly be replicated in other operating systems if left unnoticed. We show that the flaw allows a variety of powerful blind off-path TCP attacks. Finally, we propose changes to the design and implementation of TCP's global rate limit to prevent or mitigate the side channel.

Acknowledgement

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. The work is also supported by National Science Foundation under Grant #1464410.

References

[1] Blind TCP/IP Hijacking is Still Alive. `http://phrack.org/issues/64/13.html`.

[2] Cloud Messaging. `https://developers.google.com/cloud-messaging/`.

[3] [tcpm] mitigating TCP ACK loop ("ACK storm") DoS attacks. `https://www.ietf.org/mail-archive/web/tcpm/current/msg09450.html`.

[4] Abramov, R., and Herzberg, A. Tcp ack storm dos attacks. *Journal Computers and Security* (2013).

[5] Alexander, G., and Crandall, J. R. Off-Path Round Trip Time Measurement via TCP/IP Side Channels. In *INFOCOM* (2015).

[6] Bellovin, S. M. A Technique for Counting Natted Hosts. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment* (2002).

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

[8] Beverly, R., Koga, R., and k claffy. Initial Longitudinal Analysis of IP Source Spoofing Capability on the Internet. In *Internet Society Article* (2013).

[9] Chen, Q. A., Qian, Z., Jia, Y. J., Shao, Y., and Mao, Z. M. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *CCS* (2015).

[10] Chen, S., Wang, R., Wang, X., and Zhang, K. Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy* (2010).

[11] Ensafi, R., Knockel, J., Alexander, G., and Crandall, J. R. Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels. In *PAM* (2014).

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

[13] Flach, T., Katz-Bassett, E., and Govindan, R. Quantifying Violations of Destination-based Forwarding on the Internet. In *IMC* (2012).

[14] Gilad, Y., and Herzberg, A. Off-Path Attacking the Web. In *USENIX WOOT* (2012).

[15] Gilad, Y., and Herzberg, A. Spying in the Dark: TCP and Tor Traffic Analysis. In *PETS* (2012).

[16] Gilad, Y., and Herzberg, A. When tolerance causes weakness: the case of injection-friendly browsers. In *WWW* (2013).

[17] Gilad, Y., Herzberg, A., and Shulman, H. Off-Path Hacking: The Illusion of Challenge-Response Authentication. *Security Privacy, IEEE* (2014).

[18] Han, B., and Billington, J. Termination properties of TCP's connection management procedures. In *ICATPN* (2005).

[19] Javed, U., Cunha, I., Choffnes, D., Katz-Bassett, E., Anderson, T., and Krishnamurthy, A. Poiroot: Investigating the root cause of interdomain path changes. In *SIGCOMM* (2013).

[20] Katz-Bassett, E., Madhyastha, H. V., Adhikari, V. K., Scott, C., Sherry, J., Van Wesep, P., Anderson, T., and Krishnamurthy, A. Reverse Traceroute. In *NSDI* (2010).

[21] Knockel, J., and Crandall, J. R. Counting Packets Sent Between Arbitrary Internet Hosts. In *FOCI* (2014).

[22] Qian, Z., and Mao, Z. M. Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security. In *IEEE Symposium on Security and Privacy* (2012).

[23] Qian, Z., Mao, Z. M., and Xie, Y. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *CCS* (2012).

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

[25] R. Braden, Ed. Requirements for Internet Hosts - Communication Layers. rfc 1122, 1989.

[26] Ramaiah, Anantha and Stewart, R and Dalal, Mitesh. Improving TCP's Robustness to Blind In-Window Attacks. rfc5961, 2010.

[27] Song, D. X., Wagner, D., and Tian, X. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security* (2001).

[28] Xiao, Q., Reiter, M. K., and Zhang, Y. Mitigating storage side channels using statistical privacy mechanisms. In *CCS* (2015).

[29] Zhang, X., Knockel, J., and Crandall, J. R. Original SYN: Finding Machines Hidden Behind Firewalls. In *INFOCOM* (2015).