Off-Path Attacking the Web

Yossi Gilad and Amir Herzberg
Department of Computer Science Bar Ilan University

Abstract
We show how an off-path (spoofing-only) attacker can perform cross-site scripting (XSS), cross-site request forgery (CSRF) and site spoofing/defacement attacks, without requiring vulnerabilities in either web-browser or server and circumventing known defenses. Attacker can also launch devastating denial of service (DoS) attacks, even when the connection between the client and the server is secured with SSL/TLS. The attacks are practical and require a puppet (malicious script in browser sandbox) running on a the victim client machine, and attacker capable of IP-spoofing on the Internet.

Our attacks use a technique allowing an off-path attacker to learn the sequence numbers of both client and server in a TCP connection. The technique exploits the fact that many computers, in particular those running Windows, use a global IP-ID counter, which provides a side channel allowing efficient exposure of the connection sequence numbers.

We present results of experiments evaluating the learning technique and the attacks that exploit it. Finally, we present practical defenses that can be deployed at the firewall level; no changes to existing TCP/IP stacks are required.

1 Introduction
TCP is the main transport protocol over the Internet, ensuring reliable and efficient connections. TCP was not designed to be secure against Man-in-the-Middle (MitM); in fact, it is trivially vulnerable to MitM attacks. However, it seems that man-in-the-middle and eavesdropping attacks are relatively rare in practice, since they require the attacker to control routers or links along the path between the victims. Instead, most practical attacks involve malicious hosts, without MitM capabilities, i.e., the attackers are off-path.

In our attacks, as well as in many other off-path attacks (e.g., SYN-flood [42], DNS-poisoning [22]), the attacker sends spoofed packets, i.e., packets with fake (spoofed) sender IP address. Due to ingress filtering [24, 15, 5] and other anti-spoofing measures, IP spoofing is less commonly available than before, but still feasible, see [2, 14]. Apparently, there is still a significant number of ISPs that do not perform ingress filtering for their clients (especially to multihomed customers). Furthermore, with the growing concern of cyberwarfare and cybercrime, some ISPs may intentionally support spoofing. Hence, it is still reasonable to assume spoofing ability.

However, there is a widespread belief that an ‘off-path’ spoofing attacker, cannot inject traffic into a TCP connection. The reasoning is that an incoming TCP packet must contain valid sequence number (or be discarded); the sequence number field is 32 bits long and initialized using randomness, therefore, it seems unlikely that an attacker can efficiently generate a spoofed packet which will be accepted by the recipient, i.e., inject data into the TCP stream.

This belief is also stated in RFCs and standards, e.g., in RFC 4953, discussing on TCP spoofing attacks (see [42], Section 2.2). Indeed, since its early days, most Internet traffic uses TCP - and is not cryptographically protected, in spite of warnings, e.g., by Morris [33], Bellovin [8, 10].

Of course, TCP injections are easy, for implementations using predictable initial sequence numbers (ISNs). This was observed already by Morris at 1985 [33] and abused by Mitnick [38]. Later, at 2001, Zalewski found that most implementations still used predictable sequence numbers [46].

However, by now, most or all major implementations ensure sufficiently-unpredictable initial sequence numbers, e.g., following [19] [20]. Does this imply that TCP injections are infeasible?

Zalewski [48] suggested that it may be possible to spoof a non-first fragment of a (fragmented) TCP packet, when the values of the fragment IP-IDs are predictable. In particular, existing implementations of Windows use
globally-incrementing IP-ID values, which are easy to predict. Zalewski’s attack may also be applicable to Linux, using the IP-ID prediction techniques in [17]. However, exploiting such non-first-fragment TCP injections seems challenging; furthermore, currently almost all TCP implementations use path MTU discovery [32][31] and avoid fragmentation completely. Hence, Zalewski’s attack (on TCP) will rarely work in practice.

Yet, we show that TCP injections are still possible. We present an efficient, practical technique, based on globally-incrementing IP-ID, allowing an off-path adversary, Mallory, to inject data into a TCP connection between two communicating peers: a client C and a server S. The attack is not immediate, and requires a connection lasting a few dozens of seconds. We present experimental results, showing that our techniques allow efficient, practical TCP injections. Furthermore, we show, that the attacks have significant potential for abuse. Specifically, we show how our TCP injection techniques, allow both circumvention of the Same Origin Policy [6][49] and devastating DoS attacks. Details follow.

Our TCP injection technique is related to a proposal for TCP injection, by klm [27]. The technique described by klm had some limitations, e.g., it did not work for clients connected by a firewall. More significantly, klm did not present experimental results; our experiments show that their technique, even with some improvements, results in low injection success rates, unless the attacker has low latency to the victim (as when they are on the same LAN). Our techniques avoid this limitation. We provide a detailed comparison between our technique to [27] in appendix A.

Like [48][17][27], our technique is based on the predictability of the IP-ID (e.g., in Windows); however, instead of using the predictable IP-ID to intercept or modify fragmented packets, as in [48][17], we use the changes in the IP-ID as a side channel. We use this side-channel to allow the attacker to detect difference in responses for crafted probe packets that she sends to the client; our implementation uses specific probes, but others may exist.

Previous works noted that the predictable IP-ID can be used as a side channel, allowing an attacker to use one connection to learn about events in another connection, which is undesirable. Gont [18] mentions several ways in which the side channel based on globally-incremented IP-ID can be abused. However, their impact is modest. In particular, the side-channel can be used to perform the idle (stealth) scan attack [47][26][29], and to count the number of machines behind NAT [9].

However, vendors continued using globally-incrementing IP-ID values, even after we presented our initial TCP injection results to them, and being aware of the previous attacks exploiting the globally incrementing IP-IDs. Their justification is that they believed that such attacks are impractical and too complex to be exploited in practice. They confirmed our results and agreed that they are feasible; they will address the problem in new releases.

We believe that this response is ‘too little, too late’, and that it is critical for the community to be aware of the threat and apply mitigations (Section 7). A more controversial conclusion is the need to apply more prudent approach to network security vulnerabilities, and respond to early indications of weakness, without waiting for a complete exploit; see discussion in Section 8.

Much of our work focused on analyzing what are the practical implications of the TCP injection. We study two approaches to exploit TCP injections: to circumvent address based server authentication, usually referred to as Same Origin Policy (SOP) [6][49]; and to launch a Denial of Service (DoS) attack. We discuss each of these briefly in separate subsections below, and in depth in Sections 2 and 3. All attacks are based on the same attacker and network assumptions which we now describe.

Attacker Capabilities and Network Model

All our attacks work in the same settings: an off-path, IP-spoofing attacker. We also assume that the attacker is able to control some puppets [4], i.e., scripts, applets or other restricted (sandboxed) programs, running on client machines accessing an adversarial web site. This is illustrated in Figure 1 where C enters a site controlled by the adversary, Mallory. This allows Mallory to run a malicious script within C’s browser sandbox. The script allows Mallory to (1) form the connection between C and S, and (2) probe C’s connection with S and avoid firewall filtering. The first allows Mallory to choose the victim server (S), we show how the second allows exposure of the TCP connection’s four tuple.

As mentioned above, our attacks require that the connection from C and S will not be short; since Mallory forms this connection (using the puppet), she can ensure that it does not terminate by sending periodic requests to the server. In persistent HTTP connections, all re-

![Figure 1: Network Model. C enters www.mallory.com, the adversarial web page. A script on that page forms a connection with www.s.com.](image-url)
quests are over the same (victim) connection and ensure it does not close. Persistent HTTP connections are the default configuration of apache servers and are also employed by many large web-servers (e.g., Facebook, Yahoo!, Google), but not all (e.g., live.com, the Japanese version of Yahoo!). Our attacks are browser independent, as we illustrate in experiments in the following sections.

1.1 Breaking SOP and Address-Based Authentication

TCP injection attacks were key to some of the most well known exploits, specifically, attacks against address based client authentication, e.g., see [38, 10]. However, as a result, address-based client authentication has become essentially obsolete, and mostly replaced with secure alternatives such as SSH and SSL/TLS. We believe that the only widely-deployed use of address based client authentication, is to identify clients involved in DoS attacks such as SYN flooding; and this threat can be dealt with by simple client-response authentication, possibly using cookies to avoid state-exhaustion on the server [13].

However, current web security still relies, to large extent, on the Same Origin Policy [6, 49], i.e., on address based server authentication; our results show that relying on addresses to authenticate the servers is also risky.

Using TCP injections to attack address based server authentication, e.g., to perform XSS attacks, is more challenging than using it to attack address based client authentication. In attacks on address based client authentication, the off-path attacker sends the initial SYN to open a new connection; hence, she knows the client’s sequence number, as well as the source and destination IP addresses and ports; she ‘only’ needs to predict the server’s sequence number. In contrast, to attack address based server authentication, the off-path attacker must guess both sequence numbers, as well as the IP addresses and ports of both parties; guessing the client port could also be challenging, as the client may choose it arbitrarily.

To circumvent the same origin policy [6, 49], the off-path attacker sends forged responses for requests that C sends to another server S. This attack is facilitated in two phases: first the puppet opens a connection to the victim server, allowing TCP injection into this connection; then the puppet requests an object, allowing the attacker to send the script in a (spoofed) response.

In particular, this allows powerful cross site scripting (XSS). XSS is one of the most critical attacks on web security, however, current XSS techniques depend on implementation vulnerabilities, usually of the site (e.g., [45, 49]), and sometimes of the browser [25]. In contrast to typical XSS attacks, our attack does not rely on a server side or browser vulnerability. Moreover, since Mallory can choose the server S, any persistent HTTP connection between C and a server is vulnerable (see above). Connections with HTTPS servers are not vulnerable to XSS since Mallory cannot inject content into the cryptographically sealed session.

Furthermore, the XSS ability allows injection of requests, i.e., cross-site request forgery (CSRF) [41]. This circumvents existing defenses, such as the use of cookies, referer header and hidden field, since all of these are available to the (injected) script (see [54]). The CSRF attack can be prevented by challenge response methods that require user involvement, such as password authentication or CAPTCHAs.

The XSS ability also allows advanced phishing attacks. In particular, this provides efficient means for detection of browsing history, more effectively than previous techniques, e.g., [23, 44].

1.2 Devastating DoS attacks

An off-path attacker can use the knowledge of TCP parameters (IPs, ports and sequence numbers) in several ways, to attack the availability of a communication service.

In 2004, Watson observed that BGP connections used a constant client port, and typically have very large windows; this makes it feasible for an off-path adversary to reset BGP connections [43] (despite random initial sequence numbers). However, appropriate countermeasures make this attack inapplicable today [13].

We show how a spoofing, off-path attacker, who controls a limited number of (weak) puppets, can deploy formidable DoS attacks, which so far were known to require stronger attacker capabilities; the Ack-Storm attack [1] and the Optimistic-Ack attack [37]. Both are DDoS attacks, which use TCP control plane to generate excess amount of traffic. The Ack-Storm attack [1], is usually performed by MitM adversaries, possibly with limited eavesdropping abilities. The Optimistic-Ack attack [37] typically requires client cooperation (zombie) and persuades the server to send data in a high capacity, more than that allowed by C’s link. Since in both these attacks, Mallory injects data only to the TCP layer (and not to the application), these attacks also work when the victim servers use SSL.

Launching these attacks simultaneously on multiple client-server pairs may allow Mallory to conduct an improved variant of the Coremelt attack [40] and congest a core link of the Internet, using only puppets.
1.3 Organization

The next two sections focus on the application of the TCP injection technique. In Section 2 we present our off-path attacks on the confidentiality and integrity (authentication) of the communication between client and server, including the XSS, CSRF and phishing attacks. In Section 3, we present our off-path DDoS attacks.

Sections 4–6 present the TCP injection technique itself. Section 4 presents the first step, which is exposing the server’s sequence number. Section 5 continues the attack, to expose the client’s sequence number as well. Section 6 discusses deployment challenges, improvements to meet these challenges, and experimental results.

Finally, Section 7 proposes defenses against the attacks, and Section 8 presents a concluding discussion.

2 Off-Path Data Integrity Attacks

In this and the following section we present and empirically evaluate exploits of TCP injections. We focus on long-lived-connection injection attacks, where an off-path attacker learns the sequence numbers of an existing, long-lived TCP connection, between a given TCP client and server (identified by their IP addresses and ports). Motivated by these exploits, in Sections 4–6 we present and evaluate the technique we employ to study the sequence numbers.

Specifically, we show critical exploits of long-lived connection injections. In this section we focus on two exploits: the first allows an off-path attacker to run a malicious script in the context of an arbitrary website of the attacker’s choice, without depending on a vulnerability of the server (e.g., bug in input sanitization) or of the browser; this is a new, devastating type of XSS attack [45, 25]. The second exploit allows the same attacker to present spoofed web-pages for clients. In the following section we show how off-path attackers can use long-lived connection injections to cause devastating DoS attacks. All exploits work in the same setting, illustrated in Figure 1.

2.1 Identifying the Victim Connection

To launch the long-lived-connection injection attacks, the attacker must identify a connection between the client and server which is defined by the IP addresses and ports of the participating peers.

The exploits use the puppet running on the client to open such (long-lived) connections. The server’s IP and port are, of course, known. To find the client’s IP and port, the puppet opens another TCP connection to the attacker and over it, sends packets to the attacker’s machine. These packets contain the client’s IP address.

The final challenge is to detect the client port. Many clients, in particular, those running one of the Windows family operating systems, assign ports to connections incrementally. We use the puppet to open a connection to the attacker’s remote site before and after opening the connection to the victim server, incremental port assignment allows the attacker to learn the client’s port; see step A in the attack process described in Section 2.2.1 below. Client port exposing may fail if the puppet communicates with the server or attacker via a NAT device that randomizes the client port (since ‘external’ ports are not incremental). In an ongoing research we investigate this problem and provide, details of a different technique that allows an off-path attacker to detect the client port in a connection.

It remains to describe, in the following subsections, the unique aspects to each of the exploits and evaluate their impact.

2.2 Off-Path Injection XSS (or: XSS of the Fourth Kind)

In a Cross-site scripting (XSS) attack, the attacker causes the browser to run malicious, attacker-provided script (or other sandboxed code), with the permissions of scripts within a victim server web-page.

In [25], Klein identifies three kinds of XSS attacks. In persistent/stored XSS attack, the script is received from the victim server, as part of the contents of a page stored by the victim server. In reflection XSS attack, the script is ‘reflected’ by the victim server to the client, after the server receives the script from the browser (typically visiting the malicious website). Finally, in a DOM-based XSS attack, the script is received by the browser directly from the attacking server; a browser vulnerability (bug) causes the browser to consider the script as coming from some other victim server.

Both reflection and persistent/stored XSS attacks, exploit ‘bugs’ in the web application. Well designed sites, using appropriate defenses such as Web Application Firewalls (WAF), should eliminate these attacks; see, e.g., [45]. DOM-based XSS attacks do not require any bug in the site, but depend on bugs in the browser; the relevant known bugs were quickly patched by browser vendors.

Long-lived-connection injection attacks, allow a new, fourth kind of XSS attacks: off-path injection XSS attacks. In these attacks, the malicious script is sent by the attacker to the browser, with (spoofed) source IP address of the victim server. If the script it injected correctly, with correct TCP/IP parameters and within correct HTTP context, then the browser executes it in the context of the
2.2.1 Attack Process
We next explain the technique we employ to use long-lived injection attacks to perform off-path XSS injections. Like our other exploits, we assume that the user visits a website controlled by the attacker from where he receives and executes a puppet (malicious script) [4]. The attack has five steps and proceeds as follows:

A. Form Connection, Expose Client Port
Puppet opens a new connection to a server controlled by the attacker, then a connection to the victim web server and finally another new connection to a server controlled by the attacker. Let the client port numbers that the attacker observes for the first and third connections be $p_1$, $p_3$. We use the counter property of Windows port assignment: if $p_3 = p_1 + 2$, then we assume that the client used port $p_1 + 1$ for the (middle) connection to the victim server. Otherwise, repeat.

B. Expose Connection Sequence Numbers
Puppet maintains the connection with the victim server alive by sending periodic requests for small objects. During this time, attacker runs the sequence exposure attack described in Sections 4, 5. If sequence exposing fails, restart entire attack.

C. Send ‘Dummy’ Request
Puppet sends the victim server a request for some web page (over the same persistent connection), e.g., using an iframe, and informs the attacker on that request. Note that the puppet runs in the context of the attacker site; hence, the attacker and puppet can communicate and coordinate the attack without restrictions.

D. Send Spoofed Response
Attacker sends spoofed response to the client, containing exact expected TCP parameters, and a web page containing the malicious script.

E. Script Execution
Browser receives the spoofed response as if it was sent by victim server, hence, executes script with permissions of the victim server. Figure 2 shows a successful run of this attack on the Mozilla Firefox browser.

2.2.2 CSRF Exploit
As indicated in [45, 41], once attackers succeed in an XSS attack, i.e., run a malicious script in the browser, in the context of a victim site, they can exploit it in many ways. In particular, such XSS attack allows attackers to send a forged (fake) request to the server on the user’s behalf, i.e., a cross site request forgery (CSRF) attack, circumventing all known defenses against CSRF attacks for non-secured connections, except for (few) defenses requiring extra user efforts for submission of each (sensitive) request; see [34].

Note that since the attackers (cross site) scripts can read the entire response that the user receives from the victim web-server, they would even be able to circumvent advanced proposed defenses, which require new browser mechanisms. In particular, they can foil the origin header proposed by Barth et al. against CSRF attacks [7], as well as policy-based defense mechanisms against XSS, e.g., Content Security Policy (CSP) [21, 39].

2.3 Experimental validation
In this subsection we evaluate the applicability of the XSS attack on web-users. The client machine in the following experiments is protected by Windows Firewall.

The success of the XSS attack depends on successfully exposing of the sequence numbers used in the connection the client has with the victim server. The success rate of the exposure technique that we employ (presented in Sections 4, 5) depends on the rate of packets that the client machine sends. In Section 6 we present another set of experiments that specifically evaluates the injection technique for different environments. In the measurements below, C sends 32 packets per second on average.

We tested whether connections with each of the top 1024 sites in Alexa ranking [3] are vulnerable to off-path XSS attacks: our client machine connects to the attacker (www.mallory.com), who then tries to run a script in context of one of the top sites. The script provides an indication of a successful injection by requesting an image from www.mallory.com. Note that our attacker
only communicates with the client, and does not have any interaction with the victim servers. In Figure 3 we compare the results for three common browsers and observe that the attack is not browser-dependent. The immune connections are generally of the following types: (1) secured with SSL (HTTPS), this prevents the attacker from injecting his script to the connection (step D in the attack); (2) sites that do not use the HTTP keep alive option, this prevents the attacker from keeping the long connection with the server that is required to expose the sequence numbers (step B in the attack). In Figure 4 we provide distribution of the top 1024 sites in Alexa ranking; showing that 80% percent of them appear vulnerable (line 3 in Figure 4). A comparison of this result to those presented in Figure 3 shows that the XSS attack was, in fact, successful on roughly 75% of the sites that appear vulnerable. Among the vulnerable sites on which we ran a successful attack are www.facebook.com, www.yahoo.com and www.amazon.com.

2.4 Web Spoofing/Phishing/Defacement

In addition to the XSS exploits, attackers can use TCP injections to perform web spoofing (which is key to phishing attacks). Namely, the attacker waits for the user to open a connection to some victim server, e.g., http://www.bank.com, and injects his data to the connection. In this attack, the attacker provides a spoofed version of the web-page to the client. This spoofing exploit can expose sensitive user-provided information such as passwords and may trick the user to download malware. An implicit assumption of this attack is that the initial web-page that the user receives, and which the attacker forges, i.e., http://www.bank.com, is not protected by SSL. This is the situation in most sites, which do not use SSL/TLS at all.

The attack also works for many sites which do use SSL/TLS, but only via a link, e.g., to the login page https://www.bank.com/login.php. This approach is common since it reduces the load on the server by delaying setup of SSL connections until these are required (in the banking example, for login); see line 4 of Figure 4. Web-spoofing can allow the attacker to circumvent the use of encrypted connections (SSL/TLS), using techniques/tools such as SSL-strip [30], i.e., replace links on the original page to phony pages (on the attacker’s site).

To succeed in a web-spoofing attack, the attacker would best send the spoofed page as a response to a request made by the user (since then the page appears authentic to the user); hence, the attacker should be able to detect the request for the page and send a response. We solve this problem by having the puppet open a connection to the victim server in advance providing sufficient time to expose the sequence numbers used in the connection. We leave the connection open (by sending ‘dummy’ requests periodically); and probe for user activity by identifying a change in the client’s sequence number. In order to detect this change that indicates that the client had sent a request to the server, the attacker periodically conducts a client-seq-test; this test is a building block of the sequence numbers exposing technique and we provide its details in Section 5. Briefly, the test allows the attacker to test whether the client sequence number is above some value; testing using the exposed (i.e., last known) value of client sequence number allows to detect such change. Once we detect such activity over the connection, we assume that the user had sent a request to the server for the home page and send a spoofed (modified) page.

This web spoofing technique assumes that the user opens the page for the victim-server while the puppet is still running, e.g., in a different tab of the same browser or in a zero-size iframe. Furthermore, it assumes that the browser employs connection sharing between different tabs, i.e., one TCP connection is used to communicate with the same server via several tabs of the browser. TCP connection sharing is employed by the current versions of Internet Explorer, Firefox and Chrome (and possibly other browsers).

Footnote 1: Line 4 of Figure 4 counts sites which have persistent connections, to both http and https. Note that some of these sites may not often use https, while others may use https but in a different domain.
Figure 5: Web Spoofing/Defacement Attack. Mallory waits for the user to enter J.P. Morgan bank website, when he enters he injects a phony page. In this figure Mallory added a devil image.

Another assumption is that the user receives the attacker’s response before the server’s; this appears as a race that would be difficult to win for an attacker far from the client machine. However, the attacker can avoid this race by injecting data to the client (as the server) in advance: the injected data artificially increments the sequence number that the client expects from the server while the true server would still use the ‘normal’ sequence number, causing the client to reject all data sent by the server.

2.4.1 Example: Spoofing J.P. Morgan

The J.P. Morgan bank website is an example of a sensitive site that is vulnerable to this spoofing/phishing attack; it uses HTTP keep alive option and its homepage is not protected by SSL. Hence, this website is vulnerable to the web spoofing attack above. Figure 5 shows the result of a successful web spoofing attempt: here the client has two tabs open in his browser. The current tab (in focus) shows the J.P. Morgan homepage that Mallory provided; the devil image (that does not exist in the original page) indicates that this page is spoofed. J.P. Morgan homepage contains a client log-on link that in the original site switches to SSL. In the spoofed version, the link is to a web-page in Mallory’s site. In the other tab, the victim is in www.mallory.com, this allows Mallory to monitor the requests that the user (may) send J.P. Morgan and identify the correct time to inject the spoofed page.

3 Off-Path Denial-of-Service by Puppets

We next describe possible exploits of long-lived-connection injection attacks to disrupt network communication. Namely, we show how attackers can build on these attacks, to launch devastating Distributed Denial-of-Service (DDoS) attacks. As before, we consider an off-path IP-spoofing attacker, who also controls a significant number of ‘puppets’, i.e., scripts running in browsers of unsuspecting users. As argued in [4], it is relatively easy for attackers to control a large number of puppets in this way.

However, puppets have limited ability to launch DDoS attacks. One reason are limitations placed by the browsers on the number of concurrent connections opened by the same web-page. Another reason is that, due to their execution within a sandbox, puppets can only use standard TCP connections. In particular, if the attack succeeds in causing congestion and loss, then TCP connections, including those of the puppets, will significantly reduce their window size (and hence their traffic rates). Therefore, while puppets can be used for DDoS attacks, their impact is much less than that of ‘regular’ malware (zombies/bots).

In contrast to the data integrity attacks in the previous section, the attacks below use TCP control plane to cause congestion. Since there is no attempt to inject data to the application layer, even SSL/TLS protected sites are vulnerable. Hence, these attacks are applicable to websites that support persistent HTTP connections, with or without SSL/TLS, e.g., to about 80% of the 1000 most popular sites (see line 2 in Figure 4).

3.1 Off-Path Optimistic Ack Attack

We first describe Off-path Optimistic Ack; this is a variant of the Optimistic Ack DoS attack [37]. These attacks cheat TCP’s congestion control mechanism, causing senders to believe that most of the data they sent was already received, and hence that they can send more data (congestion window is not full), and also to increase the size of the congestion window. This can result in huge amplification factors, see [37], unless servers use per-connection bandwidth quotas or use other defenses.

In both Optimistic Ack attacks (ours and the original [37]), the attacker sends to the server acknowledgment packets (Acks), as if the client received all packets sent by the server (although packets are still in transit or even lost). As a result, the server continues sending information, with increasing window sizes (and hence rates).

In the original attack [37], the client must run malware, with ability to send “raw IP” packets (i.e., not according to the TCP specifications). This is a significant requirement; recent operating systems make it harder for malware to obtain such ability. Furthermore, the original attack may be blocked by a firewall on the client side, by detecting the unusual high rate. Note that since by requiring only puppets, attacker is more likely to control enough clients to succeed in the attack, in spite of countermeasures such as per-connection quotas.

Our long-lived-connection injection attack, allows an
off-path attacker to perform an off-path variant of the Optimistic Ack attack, as follows. The attacker only needs a puppet on the client machine to open the TCP connection with the victim server and learn the client port and sequence numbers. Following this, the puppet requests some large object from the server and is no longer needed; the attacker sends Ack packets as done by the client in the original attack, and - if not using SSL/TLS - the attacker can even send new request(s) if needed. Note that even if the client’s firewall (detects the attack or for some other reason) begins blocking packets on this connection from both directions, this does not help since we provide the Acks to the server from the off-path attacker. Furthermore, RFC-compliant RST packets that the firewall (or client) may send, would be out of the server’s window and hence ignored, and would not tear the connection. Server-induced verifications, by intentionally dropping or reordering packets periodically, as suggested in [87], seem one of the best (or only) defenses.

### 3.2 Off-Path Ack-Storm DoS Attack

In the original Ack-Storm DoS attack [1], the attacker needs to have some (limited) ability to eavesdrop on packets. From these packets, the attacker learns the TCP parameters (IP addresses, ports, and sequence numbers). Using these, the attacker sends two spoofed data packets, one to each of the two ends of the connection. According to the TCP specification, and in most TCP implementations, upon receiving an Ack for data that was not yet sent, TCP sends back a ‘duplicate Ack’ - i.e., resends the previously sent Ack. As a result of receiving the pair of spoofed data packets, one at each peer, both peers begin sending acknowledgment packets to each other. Since these Acks acknowledge data which was actually sent by the attacker, not by the peer, then each of these Acks will only result in another duplicate Ack returned, and this process will continue indefinitely.

The attacker can send additional data packets to the peers, causing additional ping-pong exchanges, quickly filling-up the channel capacity. This causes increased load on the networks; the fact that the packets involved in the attack are very short (just Acks), makes the load on routers and switches even higher. Eventually, this causes packet losses, and legitimate TCP connections sharing the same links significantly reduce their rate.

The off-path Ack-Storm DoS Attack works exactly like the regular Ack-Storm DoS Attack, except for using the long-lived connection injection technique to allow the off-path attacker to learn the TCP parameters. Hence the attacker can run this attack, without requiring the ability to eavesdrop.

### 3.3 Off-path Coremelt Attack

Since the two DoS attacks described above can be launched using only puppets, it follows that even relatively weak attackers may be able to cause large amounts of traffic from many clients spread around the network. This can cause high load on servers, routers and links.

In particular, by choosing well the pairs of clients and servers between which the attacks are launched, the attackers can cause huge amounts of traffic to flow over specific ‘victim’ backbone routers and links. These backbone networks, connecting large core ISPs (autonomous systems), have very high capacities; by sending enough traffic to a particular destination, attacker can cause queuing and losses in the connecting router. As a result, Internet connectivity may break - first for TCP connections and then even for UDP applications. We use the term Off-path Coremelt Attack for the resulting attack on core Internet connectivity, since it is an off-path variant of the Coremelt attack [40]. The Coremelt attack uses a large botnet, sending large amounts of traffic between pairs of the bots, with the pairs chosen intentionally so that huge amounts of traffic will flow over specific victim link/router. As shown by simulations in [40], this can result in congesting the victim link/router, and even in breaking connectivity in the Network.

In the Off-path Coremelt attack, the attacker will also congest a core link; however, instead of depending on pairs of zombies (bots), here the attack just requires a puppet at one end. The other end of the connection is a legitimate server, and to cause huge amounts of traffic, the attacker uses one of the two off-path DoS attacks described above.

This attack has three advantages compared to the original Coremelt attack: (1) controlling a sufficiently large and correctly-dispersed set of puppets is easier than controlling a comparable set of zombies; (2) we only need to control one end of each connection (the puppet), not both ends; and (3) since we use adversary-chosen servers, these can have very high bandwidth, higher than available to most bots.

### 3.4 Experimental Evaluation

We used the topology illustrated in Figure 6 to test the off-path denial of service attacks we presented. In our tests we assume that Mallory runs a puppet on C and can inject data to the TCP connection between C and S (a connection that Mallory caused C to establish).

We evaluate the attacks by measuring the degradation of service that they cause to other legitimate connections. We consider different round-trip times (RTTs) for the legitimate connections we measure: the longer the RTT is, the greater the congestion windows are. Since every loss
halves the congestion window, a more significant effect is observed when RTT is high. Furthermore, a higher RTT implies that it will take more time for the sender to detect a packet loss and retransmit. The base line to which we compare the effect of these attacks is line 1 in Figure 7, which illustrates the time it takes C* to receive a 50MB file from S* under normal conditions.

```
In this attack we aimed to clog the S’s link: Mallory uses C to request some large file from S and then performs the Optimistic Ack attack, persuading S to send data to C at high a rate. We evaluated the effect of this attack by measuring the degradation of service in a connection that S has with some other client C* who tries to download a 50MB file. Lines 1 and 2 in Figure 7 illustrate our results. Notice the significant difference in attacker and server link capacities; the amplification ratio measured in this attack is 78 (for every byte that Mallory sends, S sends approximately 78 bytes). Furthermore, the attack also clogs C’s link, as shown by line 4.

3.4.1 Off-Path Optimistic Ack Evaluation

We use the Ack-Storm attack to congest C’s link and measure the effect on a different connection that he has with some other server S*. In order to congest the link, Mallory creates a new Ack ‘ping-pong’ every 100 ms; to create each ping-pong Mallory sends only two short (40B) packets. In the connection with S*, C tries to download a 50MB file (similar to the previous experiment). Lines 1 and 3 compare the file transfer time at a normal time, to that when the Ack-Storm attack takes place. This attack requires much less effort and lower bandwidth than the Opt-Ack attack, i.e., has much higher amplification ratio; but is limited by the client bandwidth (which is typically lower than the server’s) this limitation is illustrated by line 5 which is very similar to line 1 (normal conditions).

4 Server Sequence Number Exposure

In this and the following section we describe the sequence exposure attack where an off-path adversary, Mallory, learns the current sequence numbers of a TCP connection between C and S.

We present a two phase attack: first, in this section we describe how Mallory learns the server’s sequence number, sn, which S will use in the next packet sent to C. In the second phase, presented in the following section, we show how given S’s sequence number (sn), Mallory efficiently obtains the acknowledgment number that C expects; this acknowledgment number is the sequence number that C will next use in packets sent to S. In both phases Mallory communicates only with C and aids a side channel feedback that she receives from C.

The attacks that we describe in both sections assumes that Mallory had previously identified C and S’s IP addresses and ports; see details on how Mallory exposes these parameters in Section 2.1.

4.1 The Server-Sequence Test

This subsection presents the server-sequence test that allows Mallory to test whether some sequence number, sn, is within the flow control window (wnd) that C keeps for packets he receives from S. The key observation is that when a connection is in the established state, the recipient’s handling of an empty acknowledgment packet (i.e., acknowledgment with no additional data) differs in case that it is not within wnd. The difference depends on the 32-bit sequence number and allows Mallory to search for a valid sequence number. The following paragraphs explain how:

Packets that specify an invalid sequence number (i.e., outside the recipient’s wnd) cause the recipient to send a duplicate Ack (for the last valid packet the recipient received). However, if the sequence number is within wnd, then the receiver does not send any response; the reason is that replying with an Ack in this case would start a never-ending series of acknowledgments.

The server-sequence test, illustrated in Figure 8, has
three steps: in the first and third steps Mallory sends a query to C; this is some packet that causes C to send a response packet back to Mallory who then saves the IP-ID value in the response. In Section 4.1 we show how Mallory can use the legitimate TCP connection she has with C to implement queries and responses (since C is in www.mallory.com). In the second step, Mallory sends C a probe: this packet is spoofed and appears to belong to C’s connection with S. The probe in this test is an empty Ack packet that leverages the observation above.

When Mallory receives the responses (for steps 1, 3); she uses the IP-IDs they specify, i and j, to learn x = j − i, the number of packets that C had sent between the two queries. Since the Windows IP-ID implementation is a single counter for all destinations (incremented on every packet that C sends), Mallory learns that sn is within C’s wnd if x = 1, i.e., C did not send any packet between the two queries (see Figure 8).

Figure 8: Server-Sequence Test. The dashed arrow marks the duplicate Ack that C sends only in case that sn is not within the flow control window. If sn is within C’s wnd, then C does not send that packet and j − i = 1.

4.2 Learning Process

The probability that a sequence number that Mallory tests is within C’s wnd is \( \frac{\text{wnd-size}}{2^{32}} \) (since wnd is 32 bits long); this resembles to the TCP RST forgery attack [13] where the attacker sends reset packets with arbitrary sequence numbers. The value of wnd-size is therefore important: Gont also points out the significance of the flow control window size in his security assessment for TCP [20].

Mallory conducts the server-sequence test until he identifies a sequence number within C’s wnd. Each test is for a different sequence number which is of distance \( e_{\text{wnd}} \) from the previously tested one, where \( e_{\text{wnd}} \) is an estimation of C’s wnd-size. Namely, Mallory tests sequence numbers \( 0, e_{\text{wnd}}, 2e_{\text{wnd}}, \text{etc.} \). Generally, if \( e_{\text{wnd}} < \text{wnd-size} \), then Mallory would test redundant sequence numbers, and if \( e_{\text{wnd}} > \text{wnd-size} \), then Mallory may not find a valid sequence number. In our attacks (presented in Sections 4.5) we use the puppet to request some large resource (or few small resources) over the connection with S before initiating the sequence exposure attack; the response increases C’s wnd-size. We increase wnd-size until it is approximately \( 2^{16} \). Once a sequence number within wnd is detected, Mallory conducts a binary search (over the possible \( e_{\text{wnd}} \) sequence numbers) to identify the exact beginning of wnd, which is the next server sequence number.

In Section 5 we provide an empirical evaluation of the complete sequence exposure technique.

5 Client Sequence Number Exposure

In recent Windows client versions (from XP SP2 and onwards) the recipient uses the acknowledgment number, that is specified within TCP packets, together with the sequence number to verify that a packet is valid. In order to inject a packet to the TCP stream, Mallory must specify an Ack number that is within C’s transmission window; i.e., Ack for new data that C had sent. The black area in Figure 9 represents the ‘acceptable’ acknowledgment numbers (transmission window). In this section we show how to take advantage of Ack number validation to expose the client’s sequence number.

5.1 The Client-Sequence Test

Similarly to the test we presented in the previous section, we build a three step client-sequence test where the first and last steps provide Mallory with the current value of C’s IP-ID. In the second step Mallory sends a spoofed probe, C’s response to this probe depends on the Ack number Mallory specifies.

The test is derived from another observation from the TCP specification [35] (Section 3.9, page 72). The relevant statement refers to an acknowledgment packet that carries data and contains a valid sequence number; i.e., success in the previous server sequence exposing phase is required to initiate this phase. The specification distinguishes between two cases regarding the acknowledgment number in the packet, see illustration in Figure 9.

Case 1: the packet contains a duplicate Ack (gray area in Figure 9), or acknowledges data that was sent, but not already acknowledged (black area in Figure 9). In this case the recipient is supposed to continue processing the packet regularly (see [36]). However, a Windows recipient (e.g., C) silently discards the packet if it is in the gray area (since acknowledgment is invalid); otherwise (black area) its data is copied to the received buffer for the application.

Case 2: In the complementary case that the acknowledgment number is for data that was not yet sent (white area in Figure 9), the recipient discards the segment and immediately sends a duplicate Ack that specifies his current sequence number, NXT.
Hence, when C receives an acknowledgment packet that specifies an acceptable sequence number, i.e., within his flow control window (wnd), then: (1) in case that the specified Ack number is after UNA, C sends an acknowledgment; either since data had arrived (black area), or since the packet acknowledges unsent data (white area). (2) In case that the Ack number is before UNA (gray area), then C (running Windows) discards it.

The probe which we use in the client sequence test specifies the acknowledgment number to be tested (an) and has two important properties: (1) the probe packet specifies sn, a sequence number that is within C’s wnd (discovered in the server sequence exposing phase); (2) the probe packet specifies data.

In this section we discuss the applicability of the sequence exposure technique in practice; we assume the model presented in Section 1.

6.1 Implementing Test Queries/Responses

The server and client sequence tests we described in Sections 4 and 5 use packets that Mallory receives from C to learn the effect of the (spoofed) probe packet. Mallory can persuade C to send her such packets by using the legitimate TCP connection she has with C; a query is some short data packet that Mallory sends to C, the response is the C’s acknowledgment sent back to Mallory.

This method allows Mallory to bypass typical firewall defenses since all packets in the test appear to belong to legitimate connections (requests to C-Mallory connection, probe to C-S connection). Specifically, Windows Firewall does not filter this technique.

6.2 Detecting Packet Loss

In order to succeed in sequence exposing, Mallory must identify when test packets are lost since the corresponding test will yield a wrong result. For instance, if a probe is lost, then its test will indicate that C respond to the probe and mislead Mallory.

Mallory detects a lost probe by repeating tests that indicate the client did not send a response (i.e., when \( j - i = 1 \)). There should be only few such tests: one when probing for the server’s sequence number and about sixteen during the binary search for the client sequence.

Mallory detects lost queries and responses by employing TCP congestion control. Since we implement the queries as data sent on a TCP connection, we are able to detect a lost query similarly to TCP congestion control mechanism: if Mallory receives several duplicate Acks, then she assumes that the corresponding query (a data packet) was lost. In this case Mallory repeats all the tests that she performed between the corrupt test and until its
detection. Mallory detects a lost response by identifying that no Ack was received for one of the queries (instead an accumulative Ack was received); in this case she just repeats the invalid test.

6.3 Test Errors

The sequence exposure process uses the global IP-ID to determine whether a probe caused C to respond. However, since every packet that C sends increments the IP-ID, errors may occur.

Such errors can appear only in tests where C does not respond to the probe: if C sends a packet in response to the probe, then the IP-ID is incremented, and the difference in IP-ID values of the responses that Mallory sees is at least 2; i.e., in this case Mallory always concludes that C had sent a packet.

Hence, there are only few tests where an error is possible: during server sequence exposure only one test should indicate that no packet was sent, i.e., that Mallory found a sequence within the recipient flow control window. Mallory then conducts a binary search over the values in the flow control window to find the exact sequence number. There are approximately 16 iterations to this binary search, on average, half of these indicate that C does not send a packet in response to the probe. Similarly, the binary search for the client-sequence includes 32 iterations, on average 16 tests should indicate the C did not send a packet.

6.4 Experiments

In this set of measurements we evaluate the sequence exposure technique; in Sections 2 and 3 we evaluate the full attack (that requires sequence exposing and different successful ‘meaningful’ injections). The server in these measurements is runs Apache, and the client is an up to date Windows machine (protected by Windows Firewall).

Figure 11 illustrates the success probability for different packets per second averages and when the puppet runs on different browsers. The average time for a successful sequence exposure is 102 seconds (standard deviation 18 seconds); this is the estimated time we require the client to stay in the attacker’s site to conduct cross site scripting and initiate denial of service attacks (see Sections 2 and 3). Attacker and client bandwidths are respectively 1 and 10 mbps.

We assume that Mallory can send these packets with (small) inter-packet delay, such that reordering in the test packets is a rare.

The web spoofing attack presented in Section 2.4 requires that the victim will stay in the attacker’s site until he accesses the web-page that attacker wishes to spoof.
Acked packets (‘in transit’) allows testing few sequence numbers.

We can further improve detection of the sequence exposure attack for connections that employ the TCP selective Ack option. This option is used by most modern browsers, including Internet explorer, Firefox and Chrome. A selective Ack specifies the sequence numbers of out-of-order data that was received by the sender and allows to distinguish between a duplicate Ack generated by a network loss and a duplicate Ack due to sequence exposure attempt.

The second rule is enabled when the selective Ack option is used; it verifies that no two sequential Ack packets that the server receives are identical. Normally, a duplicate Ack is a result of a packet, $p$, that arrives out-of-order. In this typical case, $p$’s sequence number must still be in the recipient’s flow control window; hence, $p$’s data is queued and the recipient sends a duplicate Ack to the source. The selective Ack attached to this feedback notifies that $p$’s data was received. However, in the case of a sequence exposure attack, most of the probes that the attacker sends are out of the recipient’s (client’s) flow control window and are discarded. Therefore, the duplicate Ack response to a probe is identical to the previous Ack that the server had received.

After several indications of an attack, the firewall tears down the connection. Note that abuse of this mechanism to cause the server to close a legitimate connection with one of his clients requires the adversary to send to the client a probe that specifies the correct connection four tuple. However, since in contrast to our attacks, the off-path attacker does not create the legitimate connection (that she tries to tear down), it is challenging to expose the connection parameters (addresses and ports).

### 7.2 Client-End Defense

In this subsection we propose modifying the IPv4 identifier at the client’s firewall (to replace the global counter). Since the identifier is only used by the recipient to match packet fragments, when a packet arrives at the sender’s firewall, it can modify the IP-ID field without any implications on the sender or recipient (even if the packet will be fragmented later on the route). When a packet arrives in fragments at the firewall, then it must map all fragments of the same packet (those that specify the same reassembly four tuple - IP addresses, protocol and IP-ID) to the same identifier.

The first, intuitively appealing direction seems to be using random identifiers. However in IPv4 this is not recommended, according to the birthday paradox, in roughly $1.2\sqrt{216} = 1.2 \cdot 2^{8} = 307$ packets there will be a repetition (since the field is 16 bits long), which would cause fragments of one packet to be mis-associated with others, and hence cripple performance.

The IP standards $^{38,11}$ specify that IP fragments are associated with a packet according to four parameters: source and destination addresses, transport layer protocol (e.g., TCP), and identifier. Therefore, a simple solution would be that each source, destination, protocol tuple will be associated with a different identifier counter, initialized by a keyed pseudo random function $f$, i.e., the initial identifier is $f_{i}(source, dest, protocol)$. In Linux, the choice of IP-ID is similar, but is only based on the source and destination addresses.

FreeBSD supports using random IPv4 IDs which are permuted locally: a packet is assigned with a random IP-ID that was not specified in one of the recent (8192) packets that were sent. Both Linux and FreeBSD approaches immune the TCP connection to our attacks.

### 8 Conclusions

In this work, we show that the folklore belief that TCP is secure against spoofing-only, off-path attackers is unfounded. We show practical, realistic injection attacks. We further show that this allows crucial abuses, breaking the same-origin policy defense which is critical to web security, and allowing devastating DoS attacks.

One important conclusion is that Bellovin $^{10}$ was right: TCP was never designed for security, and should not be expected to provide it. To ensure authentication and confidentiality, even against (only) spoofer, we should use secure protocols such as SSL/TLS $^{12}$ or IPsec $^{23}$. SSL/TLS may not suffice to prevent (lower-layer) attacks such as the DoS attacks presented in this paper (Section 3); to prevent these too, we should use lower-layer security mechanisms, preferably IPsec or other mechanisms, e.g., see $^{42,16}$.

A potentially more controversial conclusion is that basic vulnerabilities should be investigated and fixed, even before demonstration of a complete, practical, exploit. In this paper, we went into great length to prove the practicality of the vulnerability, since earlier results were considered as ‘impractical’. We believe that the network security community should adopt a more prudent approach, publishing and addressing issues and potential vulnerabilities, without waiting for a complete exploit. Compare the approach in the cryptographic community, where even yet-theoretical attacks are taken into account, published and motivate design of improved ciphers.

---

$^{4}$The default FreeBSD configuration uses a globally incrementing IP-ID, as in Windows.
References

[1] Raz Abramov and Amir Herzberg. TCP ACK Storm DoS Attacks. In Proceedings of The IFIP 26th International Information Security Conference, IFIP SEC 2011, pages 29–40, June 2011.

[2] Advanced Network Architecture Group. ANA Spoofer Project. http://spoofer.csail.mit.edu/summary.php, 2012.

[3] Alexa Web Information Company. Top Sites. http://www.alexa.com/topsites, 2012.

[4] Spiros Antonatos, Periklis Akritidis, Vinh The Lam, and Kostas G. Anagnostakis. Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure. ACM Transactions on Information and System Security, 12(2):12:1–12:15, December 2008.

[5] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704 (Best Current Practice), March 2004.

[6] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), December 2011.

[7] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, ACM Conference on Computer and Communications Security, pages 75–88. ACM, 2008.

[8] S. M. Bellovin. Security Problems in the TCP/IP Protocol Suite. Computer Communication Review, 19(2):32–48, apr 1989.

[9] Steven M. Bellovin. A Technique for Counting Natted Hosts. In Internet Measurement Workshop, pages 267–272. ACM, 2002.

[10] Steven M. Bellovin. A Look Back at "Security Problems in the TCP/IP Protocol Suite". In ACSAC, pages 229–249. IEEE Computer Society, 2004.

[11] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. Updated by RFCs 5095, 5722, 5871, 6437.

[12] Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0. RFC 2246, January 1999.

[13] W. Eddy. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational), August 2007.

[14] Toby Ehrenkranz and Jun Li. On the State of IP Spoofing Defense. ACM Transactions on Internet Technology (TOIT), 9(2), 2009.

[15] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 (Best Current Practice), May 2000. Updated by RFC 3704.

[16] Yossi Gilad and Amir Herzberg. Lightweight Opportunistic Tunneling (LOT). In Michael Backes and Peng Ning, editors, ESORICS, volume 5789 of Lecture Notes in Computer Science, pages 104–119. Springer, 2009.

[17] Yossi Gilad and Amir Herzberg. Fragmentation Considered Vulnerable: Blindly Intercepting and Discarding Fragments. In Proceedings of USENIX Workshop on Offensive Technologies, Aug 2011.

[18] F. Gont. Security Assessment of the Internet Protocol Version 4. RFC 6274 (Informational), July 2011.

[19] F. Gont and S. Bellovin. Defending against Sequence Number Attacks. RFC 6528 (Proposed Standard), February 2012.

[20] Fernando Gont. Security Assessment of the Transmission Control Protocol TCP. Internet draft, Expired: July 25, 2011, 2011.

[21] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, Proceedings of the 16th International Conference on World Wide Web, pages 601–610. ACM, 2007.

[22] Dan Kaminsky. It’s The End Of The Cache As We Know It. In Black Hat conference, August 2008. http://www.doxpara.com/DMK_BO2K8.ppt

[23] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.

[24] T. Killalea. Recommended Internet Service Provider Security Services and Procedures. RFC 3013 (Best Current Practice), November 2000.

[25] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Technical report, Web Application Security Consortium: Articles, July 2005.
[26] Amit Klein. OpenBSD DNS Cache Poisoning and Multiple O/S Predictable IP ID Vulnerability.  http://www.trusteer.com/docs/dnsopenbsd.html, October-November 2007.

[27] klm. Remote blind TCP/IP spoofing. Phrack magazine,  http://www.phrack.org/issues.php?issue=64&id=15, 2007.

[28] Gunnar Kreitz. Timing Is Everything: The Importance of History Detection. In Vijay Atluri and Claudia Díaz, editors, ESORICS, volume 6879 of Lecture Notes in Computer Science, pages 117–132. Springer, 2011.

[29] Gordon Lyon. Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning.  http://nmap.org/book/, 2009.

[30] M. Marlinspike. New Tricks for Defeating SSL in Practice. In BlackHat DC, February 2009.

[31] M. Mathis and J. Heffner. Packetization Layer Path MTU Discovery. RFC 4821 (Proposed Standard), March 2007.

[32] J.C. Mogul and S.E. Deering. Path MTU discovery. RFC 1191 (Draft Standard), November 1990.

[33] Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. Technical report, AT&T Bell Laboratories, February 1985.

[34] Paul Petefish, Eric Sheridan, and Dave Wichers. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet.  https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet, 2011.

[35] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[36] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[37] Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud. Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse. In Catherine Meadows and Paul Syverson, editors, Proceedings of the 12th ACM Conference on Computer and Communications Security, pages 383–392, pub-ACM:adr, 2005. ACM Press.

[38] Tsutomu Shimomura and John Markoff. Take-down: The Pursuit and Capture of Kevin Mitnick, America’s Most Wanted Computer Outlaws - by the Man Who Did It. Hyperion Press, 1st edition, 1995.

[39] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the Web with Content Security Policy. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, Proceedings of the 19th International Conference on World Wide Web, pages 921–930. ACM, 2010.

[40] Ahren Studer and Adrian Perrig. The Coremelt Attack. In Michael Backes and Peng Ning, editors, ESORICS, volume 5789 of Lecture Notes in Computer Science, pages 37–52. Springer, 2009.

[41] The Open Web Application Security Project (OWASP). Cross-Site Request Forgery (CSRF).  https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF), 2010.

[42] J. Touch. Defending TCP Against Spoofing Attacks. RFC 4953 (Informational), July 2007.

[43] P. Watson. Slipping in the Window: TCP Reset Attacks. presented at CanSecWest,  http://bandwidthco.com/whitepapers/netforensics/tcpip/TCPResetAttacks.pdf, October 2004.

[44] Zachary Weinberg, Eric Yawei Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks. In IEEE Symposium on Security and Privacy, pages 147–161. IEEE Computer Society, 2011.

[45] Jeff Williams and Jim Manico. XSS (Cross Site Scripting) Prevention Cheat Sheet.  https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, January 2012.

[46] M. Zalewski. Strange Attractors and TCP/IP Sequence Number Analysis.  http://lcamtuf.coredump.cx/newtcp/2001.

[47] M. Zalewski. Silence on the wire: a field guide to passive reconnaissance and indirect attacks. No Starch Press, 2005.

[48] Michal Zalewski. A New TCP/IP Blind Data Injection Technique? BugTraq mailing list post,  http://lcamtuf.coredump.cx/ipfrag.txt, 2003.

[49] Michal Zalewski. The Tangled Web: A Guide to Securing Modern Web Applications. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.
A Comparing Performance of TCP Injection Attacks

In this appendix we compare the existing approach and technique for TCP injection presented in [27] to those presented in this paper. The significant difference between the two approaches is that [27] injects data to a legitimate existing connection between two peers (C and S) where in this paper we use a puppet to create the victim connection. This difference has three implications we describe below.

First, the attacker must identify the connection between C and S and expose its parameters (IP addresses and ports). In [27] attacker is assumed to have previous knowledge of the client and server addresses as well as the server’s port; in this paper we assume only knowledge of the server’s address and port which are usually available. In order to expose the client’s port, in [27] the attacker performs a variant of the idle scan, indirectly scanning all possible client ports. The scan is as follows: the attacker sends a SYN to the server spoofed as if from the client; if there is already a connection through the client port specified in the SYN packet, then the server ignores the spoofed SYN. Otherwise the server sends a SYN/ACK packet to the client who will respond in RST. The attacker uses the global IP-ID to test whether the client sent a packet in response.

Implementing this method for probing the client port has a few challenges: (a) this technique is filtered by typical client firewalls (e.g., Windows Firewall) that will discard the SYN/ACK server response in case that the client did not first send a SYN. (b) attacker must run a synchronized attack, querying for the client IP-ID, then assume that the server probe had arrived and query for the IP-ID again; if during this time C sends a packet or server SYN/ACK does not yet arrive then the test is invalid.

In contrast, we create the connection using the puppet and identify the client port by using an insight on Windows port allocation paradigm. This allows us to form a connection with an ‘interesting’ server and efficiently detect its parameters (see Section 2.1).

Second, the attacker in [27] must cope ongoing traffic over the victim connection itself. Such traffic fails the binary search for the client sequence number (see Section 5) since this phase requires specifying a valid sequence number (which keeps changing due to traffic on the connection). Moreover, [27] does not describe how to implement the queries to (1) avoid firewall filtering and (2) detect network losses. In the approach presented in this paper, the attacker controls the connection since her puppet makes the requests for the server. Hence she is able to avoid traffic on the connection while exposing the sequence numbers. The legitimate TCP connection with the client is used to implement the queries (see details in Section 6).

In Figure 12 we compare the success rates of our attack to that described in [27] where the victim connection in while running the attack in [27] has only a modest 10 kbps traffic rate (since attacker does not control the traffic rate in this case). The comparison is for different network delays between the client and attacker; the longer the delay, the more time until the attacker receives feedback and the more traffic that passes on the connection. Since [27] does not specify how to implement the queries, we used our method, i.e., on a TCP connection between the client and the attacker. We assume that the attacker in [27] successfully detects the client port (despite the challenges above). We also assume that the client sends an average of 32 packets per second to other peers.

Figure 12: Comparison of sequence exposure techniques. Each measurement is the average of 50 runs, error bars mark standard deviations.

The third difference between our approach to [27] regards to the practical challenge of performing a ‘meaningful’ injection. That is, after a successful exposure of sequence numbers, the attacker should identify the right time to inject his data; For example, to perform the XSS attack, the spoofed response must arrive after the client had sent a request; it is hard for an off-path attacker to detect that time. In contrast, the attacks in this initiate the request using the puppet and inject the response (see Section 2).