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. The attacks are practical and require a puppet (malicious script in browser sandbox) running on a victim client machine, and an attacker capable of IP-spoofing on the Internet.

Our attacks are based on a technique that allows an off-path attacker to efficiently learn the sequence numbers of both the client and server in a TCP connection. This technique exploits the fact that many computers, in particular those running (any recent version of) 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. We also present practical defenses that can be deployed at the firewall level, either at the client or server end; 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, many 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, DNS-poisoning), the attacker sends *spoofed packets*, i.e., packets with fake (spoofed) sender IP address. Due to ingress filtering [18] and other anti-spoofing measures, IP spoofing is less commonly available than before, but still feasible, see [1, 11]. 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, 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 a 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 [30], Section 2.2). Indeed, since its early days, most Internet traffic is carried over TCP - and is not cryptographically protected, in spite of warnings, e.g., by Morris [23] and Bellovin [6, 8].

TCP injections are easy for implementations that use predictable initial sequence numbers (ISNs). This was observed already by Morris at 1985 [23] and abused by Mitnick [26]. Later, at 2001, Zalewski found that most implementations still used predictable ISNs [34]. However, by now, most or all major implementations ensure sufficiently-unpredictable ISNs, e.g., following [15]. Does this imply that TCP injections are infeasible?

We show that TCP injections are still possible. We present an efficient and 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 IP-ID field is specified in every IPv4 packet and allows the recipient to match fragments of an

IP packet during reassembly. In our attacks we assume that a globally incrementing IP-ID is employed by C, this IP-ID increments for every packet that the C sends[1]. A globally incrementing IP identifier is used in all Windows versions we tested (including XP, Vista and 7) and is also the default configuration in FreeBSD. However, it is not implemented in all operating systems; e.g., Linux machines use a different IP-ID counter for each destination and are immune to our attacks. The vast deployment of Windows on client machines, more than 70% according to browser user-agent based surveys, see [32], makes the IP-ID attack vector very practical.

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 *circumvention of the Same Origin Policy* [4, 36].

Our technique is based on the predictability of the IP-ID (e.g., in Windows); we use the changes in the IP-ID as a *side channel* to allow the attacker to detect difference in responses for crafted probe packets that she sends to the client.

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 [14] mentions several ways in which the globally-incremented IP-ID can be abused; but, their impact is modest. In particular, the side-channel can be used to perform the idle scan attack [35] (implemented in nmap), and to count the number of machines behind a NAT [7].

Our TCP injection technique improves upon the one presented by klm [20]. The technique described by klm had some limitations, e.g., it did not work for clients connected to the Internet by a firewall. More significantly, klm did not present experimental results; we experimentally compare our technique to [20]. The experiments show that their technique results in low injection success rates, unless the attacker has low latency to the victim (as when they are on the same LAN); it is doubtful that these results could allow significant exploits, as we were able to achieve.

1.1 Attacker 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* [3], 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

[1]Sever IP-ID implementation does not effect our technique.

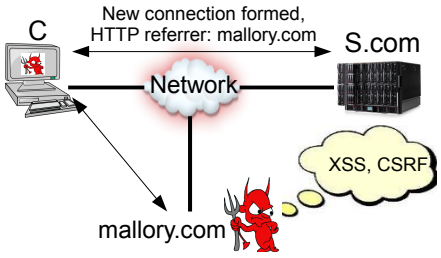

Figure 1: Network Model. C enters www.mallory.com, the adversarial web page. A script on that page forms a connection with www.s.com.

by the adversary, Mallory. This allows Mallory to run a malicious script in 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 (IP addresses and ports). Our attacks are browser independent, as we illustrate in experiments in the following sections.

1.2 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 [8]. 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 challenge-response authentication, possibly using cookies to avoid state-exhaustion on the server [10].

However, current web security still relies, to large extent, on the *Same Origin Policy* [4, 36], 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 identify *both* sequence numbers, as well as the IP ad-

dresses and ports of both parties.

To circumvent the same origin policy, 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 *cross site scripting (XSS)*. In contrast to known XSS attacks, our attack does not rely on server or browser vulnerability. Furthermore, our attack circumvents defenses against XSS as well as against *cross-site request forgery (CSRF)* [28], such as *Content Security Policy (CSP)* [27].

The XSS ability also allows advanced attacks. In particular, XSS can exploit use of password managers to learn the user password [29] and provides efficient means for detection of browsing history, more effectively than previous techniques, e.g., [21, 31].

1.3 Organization

Section 2 explains how an off-path attacker identifies the victim connection between the client and server. Sections 3-5 present the TCP injection technique itself: Section 3 presents the first step, which is exposing the server's sequence number. Section 4 continues the attack, to expose the client's sequence number as well. Section 5 discusses challenges, improvements to meet these challenges, and experimental evaluation.

Next, Sections 6 and 7 focus on the exploits of the TCP injection technique and 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.

Section 8 compares between the injection technique presented in this paper to the one in [20].

Lastly, Section 9 proposes defenses against the attacks and Section 10 presents a concluding discussion.

2 Identifying the Victim Connection

To launch the injection attacks, the attacker must first identify a TCP connection between the client and server; the connection is defined by the IP addresses and ports of the participating peers.

Our 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, the puppet sends a request to the attacker's site; this request contains the client's IP address.

The final challenge is to detect the client port. Many clients, in particular, those running Windows, assign ports to connections *sequentially*. We use the puppet to open a connection to the attacker's remote site before and after opening the connection to the victim server (S); sequential port assignment allows the attacker to learn the client's port: Mallory observes p_1 and p_2, the client ports used in the connection to her sites. If $p_2 = p_1 + 2$, then she identifies that the connection to S is via port $p_1 + 1$. For other client port allocation paradigms or when the puppet communicates via a NAT device that randomizes the client port, we use the technique that we presented in Section 3 of [13] to learn the client port.

3 Server Sequence Number Exposure

In this and the following section we describe the sequence exposure attack where an off-path adversary, Mallory, communicates with C and 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, σ, which S will use in the next packet sent to C. In the second phase, presented in the following section, we show how given σ Mallory efficiently learns 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 sections we assume that Mallory had identified C and S's IP addresses and ports as we described in Section 2.

3.1 The Server-Sequence Test

This subsection presents the *server-sequence test* that allows Mallory to test whether some sequence number, $\dot{\sigma}$, is in the flow control window (*wnd*) that C keeps for packets from S. The key observation is that when a TCP connection is in the established state, the recipient's handling of an *empty acknowledgment* packet (i.e., acknowledgment with no additional data) depends on the value of the 32-bit sequence number.

Empty-Ack 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; in the typical (i.e., legitimate) case, this duplicate Ack indicates to the sender that a packet loss occurred. However, if the sequence number is in *wnd*, then the receiver does not send any response; the reasoning is that 'Acking' the valid empty Ack packet will start a never-ending series of acknowledgments. This observation does not depend on other fields in the TCP header; in particular, the response to an empty-Ack packet does not depend on the actual value of the Ack field, which we show in the next section how Mallory learns.

The *server-sequence test*, illustrated in Figure 2, has three steps: in the first and third steps, Mallory sends a

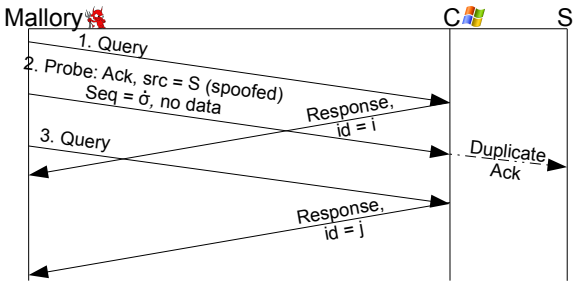

Figure 2: Server-Sequence Test.

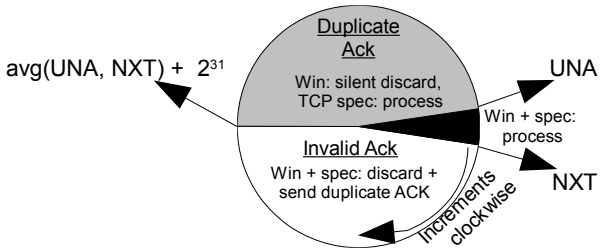

Figure 3: Ack Number Map. UNA is the lowest unacknowledged sequence number, NXT is the next sequence number that C will send. The 32-bit Ack field is cyclic.

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 5.1 we show how Mallory can use the legitimate TCP connection that she has with C to implement queries and responses (since C has a TCP connection to 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 and 3 in Figure 2), she uses the IP-IDs that they specify, i and j, to learn $x = j - i$. Since the IP-ID implementation increments for every packet that C sends, x is the number of packets that C had sent between the two queries. Mallory learns that $\dot{\sigma}$ is in C's *wnd* if $x = 1$, i.e., C did not send any packet between the two queries.

3.2 Linear Search for σ

Mallory performs the server-sequence test for the sequences: 0, e_{wnd}, $2e_{wnd}$, etc, until she identifies a sequence number in C's *wnd*. The value e_{wnd} is an estimation of C's *wnd-size*. In our attacks (presented in Sections 6 and 7) we use the puppet to request for some large resource (or few small resources) over the connection with S before initiating the sequence exposure attack; S's response (i.e., the large object) increases C's *wnd-size*. We use this technique to increase *wnd-size* to approximately 2^{16}. Once a sequence number in *wnd* is detected, Mallory performs a binary search to identify the beginning of *wnd* (over the possible e_{wnd} sequence numbers), i.e., σ.

4 Client Sequence Number Exposure

In recent Windows client versions, from XP SP2 and onwards, the recipient uses the acknowledgment number, that is specified in 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 α, an Ack number that is in C's transmission window; i.e., Ack for new data that C had sent. The black area in Figure 3 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.

4.1 The Client-Sequence Test

Similarly to the test that 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 that Mallory specifies.

The test is derived from another observation from the TCP specification [25] (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 3.

Case 1: the packet contains a duplicate Ack (gray area in Figure 3), or acknowledges data that was sent, but not already acknowledged (black area in Figure 3). In this case, the recipient is supposed to continue processing the packet regularly (see [25]). However, a Windows recipient (i.e., C) silently discards the packet if it is in the gray area (since acknowledgment is invalid); otherwise (black area), the 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 3), the recipient discards the packet and immediately sends a duplicate Ack that specifies his current sequence number, i.e., NXT in Figure 3.

Hence, when C receives an acknowledgment packet that specifies an acceptable sequence number, i.e., in his flow control window (*wnd*), then: (1) in case that the specified Ack number is after UNA, C sends an acknowledgment; either since new data 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 that is tested and has two important properties derived from the observation above: (1) the probe packet specifies σ, a sequence number that is in C's *wnd* (discovered in the previous server sequence exposing phase); (2) the probe packet carries data ('non-empty' packet).

4.2 Binary Search for α

The client-sequence test allows Mallory to perform a binary search for the acknowledgment number that the client expects. If the client-sequence test for the acknowledgment number $\dot{\alpha}$ indicates that C did not send any packet between the two queries, then $\dot{\alpha}$ is below UNA (in the gray area in Figure 3). Otherwise, Mallory concludes that α is above UNA (in the black or white area in Figure 3).

The gray and white areas in Figure 3 are of equal size, and the black area (sent bytes without acknowledgment) is usually relatively small. This allows Mallory to perform a binary search for UNA; each time eliminating approximately half the possible numbers. UNA is the lowest number in the black area, i.e., it is a valid Ack number (α). The 32-bit length of the Ack field implies that there are 32 iterations.

5 Implementation and Evaluation of Sequence Numbers Exposure

In this section we discuss the implementation of the sequence exposure technique and its evaluation in practice; we assume the model presented in Section 1.1.

5.1 Implementing Test Queries/Responses

The server and client sequence tests which we described in Sections 3 and 4 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 that she has with C (since C is 'in' www.mallory.com): a query is some short data packet that Mallory sends to C, the response is C's TCP 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: queries and responses belong to the connection between C and Mallory; probes belong to the connection between C and S. Specifically, we found that Windows Firewall does not filter the queries, responses or probes that we use.

5.2 Detecting Packet Loss

In order to succeed in sequence exposing, Mallory must identify when test-packets (queries, responses or probes) are lost since the corresponding and following tests will yield a wrong result.

Mallory detects a lost probe by repeating tests which indicate that the client did not send a response (i.e., when the difference in response IP-IDs equals to one). There should be only few such tests: one when probing for the server's sequence number, where no response to the probe indicates that Mallory found a sequence number in the recipient flow control window. Additionally, approximately sixteen probes during the binary search for the client sequence number should not receive a response. Hence, repeating tests which indicate 'no response to the probe' does not significantly increase the time of the attack.

Mallory detects lost queries and responses by using TCP congestion control. Since we implement the queries as data sent over the TCP connection between C and Mallory, we are able to detect a lost query similarly to TCP congestion control mechanism: if a query does not arrive (to C), then Mallory receives a duplicate Ack for the following query; similarly, if a response does not arrive (to Mallory), then the following response is an accumulative Ack. In these cases, Mallory performs again the corresponding tests.

5.3 Errors in Tests

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, independent of the probe, between responding to the Mallory's test-requests, then that packet would increment the IP-ID. This event will appear to Mallory as the case where C responded to her probe; i.e., provide a false indication. As discussed in the previous subsection, there are only few tests where the probe does not yield a response, i.e., where such an error is possible.

We handle errors in the server and client sequence exposure phases differently. During the server sequence exposure phase, Mallory tests many possible sequence numbers; however, only one of these tests can yield an error result (the one that tests for a valid sequence number, i.e., in C's *wnd*). Hence, the probability of an error in this phase is low (since there is only one 'critical' test). We identify that such error had occurred after Mallory tests the entire sequence space and all tests indicate a negative result; in this case we restart the attack.

During the client sequence number exposure phase,

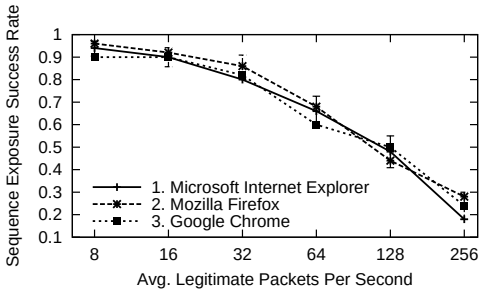

Figure 4: TCP sequence exposure success rate. Each measurement is the average of 50 runs, error bars mark standard deviations.

we perform only 32 tests (binary search for the Ack number); since approximately 16 of these tests should indicate that the probe did not cause C to send a packet, the probability for an error is greater than in the previous phase. However, since the number of tests is low in this phase, we cope with possible errors by repeating the tests which indicate that C responds to the probe without adding a significant overhead to the entire sequence number exposure process.

5.4 Empirical Evaluation

In this set of measurements we provide the adversary with the IP addresses and ports that describe the victim connection and evaluate the sequence exposure technique (presented in Sections 3 and 4); in Section 6 we evaluate the full attack which requires to identify the victim connection, expose the sequence numbers and perform different successful 'meaningful' injections. The server in these measurements runs Apache (version 2.2.14), and the client is an up to date Windows machine (protected by Windows Firewall).

Figure 4 illustrates the probability for successful exposure for different packet rates and when the puppet runs on different browsers. The attacker and client bandwidths are respectively 1 and 10 mbps and the round trip time between Mallory and C is 100 milliseconds. The average time for a successful sequence exposure is 102 seconds (standard deviation 18 seconds); this is the estimated that time we require the client to stay in the attacker's site (www.mallory.com) to perform the XSS and CSRF attacks described in the following sections. In Section 8 we provide a detailed comparison between our sequence exposure technique to the one previously presented in [20].

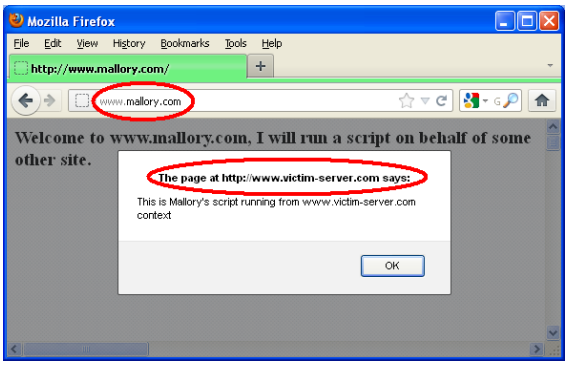

Figure 5: An XSS Attack. Mallory runs a script in context of www.victim–server.com within Mozilla Firefox sandbox. The address bar indicates the user is at www.mallory.com, but the message box context indication shows that the script (that Mallory provided) runs from www.victim–server.com.

6 XSS and CSRF Exploits

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 client and a server (identified by their IP addresses and ports).

We focus on two exploits: the first, presented in this section, 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 type of XSS attack [19, 33]. The second exploit, which we present in the following section, allows the same attacker to present spoofed webpages for clients. We evaluate these attacks on connections with popular websites.

All exploits work in the same setting, illustrated in Figure 1.

6.1 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. Known XSS attacks, exploit 'bugs' in the web application or in the browser [19], which were (mostly) fixed.

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 is injected correctly, with correct TCP/IP parameters and within correct HTTP context, then the browser executes it in the context of the victim site.

6.1.1 Attack Process

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) [3]. Our puppet code is available online at [12] with explanations and documentation that refers to the text below, which describes the five steps of the attack:

A. Establish a connection from the client to the victim server, identify client port (see in Section 2).

B. Expose connection sequence numbers. Puppet keeps the connection with the victim server alive by periodically sending requests for small objects. During this time, attacker runs the sequence exposure attack described in Sections 3 and 4.

C. Send a 'dummy' request. Puppet sends the victim server a request for some web page (over the same persistent connection), e.g., using an iframe (see our code [12]), and informs the attacker on that request. Note that the puppet runs in the context of Mallory's site; hence, Mallory and puppet can communicate and coordinate the attack without restrictions.

D. Send spoofed response. Attacker sends a 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 5 shows a successful run of this attack on Mozilla Firefox.

6.2 CSRF Exploit

As indicated in [28, 33], 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 [24].

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 [5], as well as policy-based defense mechanisms against XSS, e.g., *Content Security Policy (CSP)* [16, 27].

6.3 Empirical Evaluation

In this subsection we evaluate the applicability of the XSS attack on web-users. The client machine in the following experiments is as in the evaluation of the sequence exposure technique presented in Section 5.4.

The success of the XSS attack depends on successfully exposing the sequence numbers used in the connection that the client has with the victim server. The success rate of the sequence exposure technique (presented in Sections 3, 4) depends on the rate of packets that the client sends (see Section 5.3 for details). In the measurements below, C sends 32 packets per second. In Section 5 we presented another set of experiments that specifically evaluates the injection technique in different environments.

We tested whether connections with each of the top 1000 sites in Alexa ranking (see [2]) are vulnerable to off-path XSS attacks: our client 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 machine, and does not have any interaction with the victim servers.

In Figure 7 we compare the results for three common browsers and observe that the attack is browser independent. The immune connections are generally to sites of the following types:

1. do not support persistent HTTP connections, i.e., do not use the HTTP keep alive option. This prevents the attacker from keeping the long connection with the server, which is required to expose the sequence numbers (attack step B).

2. secured with SSL (HTTPS). This prevents the attacker from injecting her script to the connection (attack step D).

In Figure 6 we provide distribution of the top 1000 sites in Alexa ranking; showing that 80% of them appear vulnerable (line 1 in Figure 6). A comparison of this result to those presented in Figure 7 shows that the XSS attack was 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`.

The reason that the attack does not succeed for all potential victim connections is that in some attempts our attacker failed to identify the correct client port (e.g., if the browser re-used a port allocated in the past for the

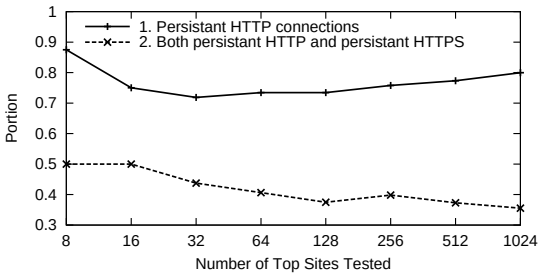

Figure 6: The applicability of the injection attacks on various sites.

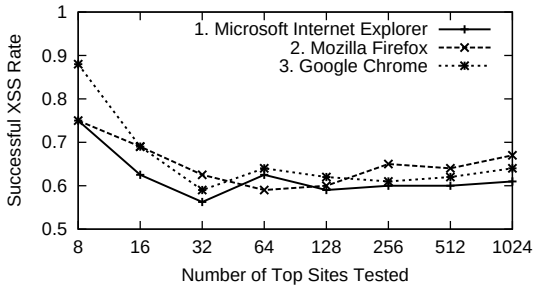

Figure 7: Rate of successful XSS attacks on connections to popular sites.

same server). Employing the more elaborate technique in [13] to identify the victim connection can improve our results.

7 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 browse to some website, e.g., `www.bank.com`, and injects her data to the connection. In this attack, the attacker provides a spoofed version of the website to the client. This exploit can expose user-provided information such as passwords and may trick the user to download malware. A requirement of this attack is that the initial web-page that the user receives, and which the attacker forges, is not protected by SSL; i.e., `http://www.bank.com`. This assumption holds for 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 (e.g., for login); see line 2 of Figure 6. Web-spoofing allows the attacker to circumvent the use of encrypted connections (SSL/TLS), using techniques/tools such as SSL-strip [22], i.e., replace links on the original page to phony pages (on the attacker's site). This technique can be made unnoticeable for typical users by presenting them a spoofed web-page with the original content, i.e., attacker just modifies targeted links (e.g., for the login page) to point to her website. However, an alert user who follows such a modified link might notice the change in domain and detect the attack.

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 periodically sending 'dummy' requests); and probe for user activity by identifying an increment in the client's sequence number.

In order to detect this change, which indicates that the client had sent a request to the server, the attacker periodically performs a *client-seq-test* which we presented in Section 4. The test allows the attacker to identify 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 user activity over the connection, which we assume is a request for the server's 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).

Another assumption is that the user receives the attacker's response before the server's; at first glance, 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 'dummy' 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.

The reminder of this section presents the implementation of the web-spoofing attack on the J.P. Morgan

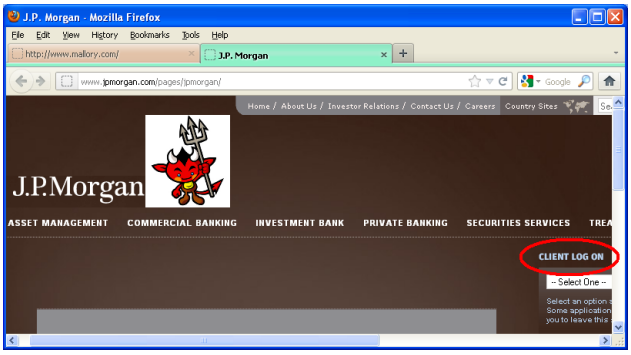

Figure 8: Web Spoofing/Defacement Attack. Mallory waits for the user to enter J.P. Morgan bank website, when he enters he injects a phony page. In this figure Mallory added a devil image.

bank homepage. We have also confirmed this attack to work on the following banks web-pages: Goldman Sachs (`http://www.goldmansachs.com/`), Morgan Stanly (`http://www.morganstanley.com/`) and The Royal Bank of Scotland (`http://www.rbs.co.uk/`). All of these banks use a HTTP homepage (and persistent connections) and only switch to HTTPS when the client clicks the login button.

7.1 Example: Spoofing J.P. Morgan

The J.P. Morgan bank website is an example of a sensitive site that uses HTTP keep alive option and its homepage is not protected by SSL (but the login page is protected). Hence, this website is vulnerable to the web spoofing attack above. Figure 8 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 (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, this link is to a web-page in Mallory's site. In another 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.

8 Performance Comparison of TCP Injection Techniques

In this section we compare the TCP injection technique presented in [20] to the one presented in this paper. The significant difference is that [20] injects data to a legitimate *existing* connection between two peers (C and S) where in this paper we make and additional assumption: that the attacker runs a puppet on the victim machine; we use that puppet to *create* the victim connection. This difference has three implications which we describe below.

First, the attacker must identify the connection between C and S and expose its parameters (IP addresses and ports). In [20], the attacker is assumed to have previous knowledge of the client and server addresses as well as the server's port. In order to expose the client's port, in [20] 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 which is spoofed as if sent by 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.

This technique for probing the client port has a few challenges: (1) 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. (2) 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 expose the connection parameters (see Section 2).

Second, the attacker in [20] must cope with traffic over the victim connection. Such traffic disrupts the search for the client sequence number (see Section 4) since this phase requires specifying a valid sequence number, which keeps changing due to traffic over the connection between the client and the server. Moreover, [20] 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 puppet communicates with 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 queries (see Section 5).

In Figure 9 we compare the success rates of our sequence exposure technique to that described in [20] where the victim connection in while running the attack in [20] has only a modest 10 kbps traffic rate. 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 [20] does not specify how to implement the queries, we used our method, i.e.,

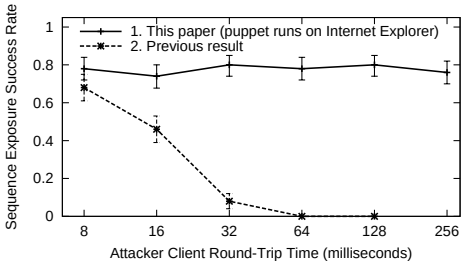

Figure 9: Comparison of sequence exposure techniques. Each measurement is the average of 50 runs, error bars mark standard deviations.

on a TCP connection between the client and the attacker. We assume that the attacker in [20] 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.

The third difference between our approach to [20] 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 her 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 paper initiate the request using the puppet and inject the response (see Section 6).

9 Defense Mechanisms

The attacks in this paper relay on successful exposure of the sequence numbers; the technique that we presented for this task uses the global counter property of the IP-ID implementation in Windows machines. Deployment of IPv6 mitigates this attack vector since the IPv6 fragmentation header (that specifies the IP-ID) is only present in fragmented packets. In most implementations, TCP employs path MTU discovery to avoid IP-fragmentation. Hence, TCP connections over IPv6 are usually immune to our attacks.

In this section we propose defenses that prevent off-path sequence exposing. Our mechanisms are of two types, those deployed at the client-end, and those deployed at the server-end. Each mechanism blocks the attack even if the other peer is vulnerable; i.e., servers and clients can independently protect themselves.

9.1 Server-End Defense

This defense uses feedback that the client machine (that runs the puppet) involuntarily sends as a side effect of the ID-exposing process. For every wrong guess of the server sequence number, the client sends to the server a duplicate Ack (see Section 3 and Figure 2) with no data. A firewall can monitor these empty Ack packets and verify that: in no point in time the server received more 'empty-Acks' than the number of un-Acked data packets that he had sent to the client. The firewall tears down the connection after several alerts by this rule.

9.2 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, then when a packet arrives at the sender's firewall, the firewall 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).

The first, intuitively appealing direction seems to be using random identifiers; however, this will often cause IP-IDs of packets 'in-transit' to collide. Such collisions may cause packet loss in case that these packets are fragmented since fragments of different packets will be misassociated together.

The IP standards specify that IP fragments are associated with a packet according to four parameters: source and destination addresses, transport layer protocol (e.g., TCP), and the IP identifier. The global IP-ID side channel can be eliminated by assigning each source, destination, protocol tuple a different identifier counter, initialized by a keyed pseudo random function f (e.g., keyed hash function); i.e., the initial identifier is $f_k(\textit{source, dest, protocol})$, where k is a secret key. 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[2].

Both Linux and FreeBSD approaches immune the TCP connection to our attacks.

10 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.

One important conclusion is that Bellovin [8] was right: TCP was never designed for security, and should not be expected to provide it. To ensure authentication and confidentiality, even against (only) spoofers, we should use secure protocols such as SSL/TLS [9] or IPsec [17].

[2]However, the default FreeBSD configuration uses a globally incrementing IP-ID, as in Windows.

Acknowledgments

Many thanks to Amit Klein, Daniele Perito and the anonymous referees for their invaluable comments. This work was supported by grant 206703 of the Israeli Science Foundation.

References

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

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

[3] 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.

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

[5] 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.

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

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

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

[9] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

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

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

[12] Yossi Gilad and Amir Herzberg. Puppet Code (Java Script). `http://u.cs.biu.ac.il/~herzbea/security/code/puppet-example.js`, 2012.

[13] Yossi Gilad and Amir Herzberg. Spying in the Dark: TCP and Tor Traffic Analysis. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

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

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

[16] 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.

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

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

[19] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Technical report, Web Application Security Consortium: Articles, July 2005.

[20] klm. Remote Blind TCP/IP Spoofing. Phrack magazine, `http://www.phrack.org/issues.html?id=15&issue=64`, 2007.

[21] 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.

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

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

[24] 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.

[25] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981.

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

[27] 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.

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

[29] Ben Toews. Abusing password managers with xss. http://labs.neohapsis.com/2012/04/25/abusing-password-managers-with-xss/, 2012.

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

[31] 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.

[32] Wikipedia. Usage Share of Operating Systems. `http://en.wikipedia.org/wiki/Usage_share_of_operating_systems`, December 2011.

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

[34] Michal Zalewski. Strange Attractors and TCP/IP Sequence Number Analysis. `http://lcamtuf.coredump.cx/newtcp/`, 2001.

[35] Michal Zalewski. *Silence on the Wire: A Field Guide to Passive Reconnaissance and Indirect Attacks*. No Starch Press, 2005.

[36] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.