Off-Path TCP Injection Attacks

Yossi Gilad[†] and Amir Herzberg[‡]
Department of Computer Science, Bar-Ilan University
[†]mail@yossigilad.com, [‡]amir.herzberg@gmail.com

We present practical off-path TCP-injection attacks for connections between current, non-buggy browsers and web-servers. The attacks allow *web-cache poisoning* with malicious objects such as spoofed web-pages and scripts; these objects can be cached for a long period of time, exposing any user of that cache to *cross-site scripting*, *cross-site request forgery* and *phishing* attacks.

In contrast to previous TCP injection attacks, we do not require MitM capabilities or malware running on the client machine. Instead, our attacks rely on a weaker assumption, that the user only enters to a malicious web-site, but does not download or install any application. Our attacks exploit subtle details of the TCP and HTTP specifications, and features of legitimate (and very common) browser implementations. An empirical evaluation of our techniques with current versions of browsers shows that connections with most popular websites are vulnerable.

We conclude this work with practical client and server end defenses against our attacks.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms: Security

Additional Key Words and Phrases: Web and Network Security; Browser Security

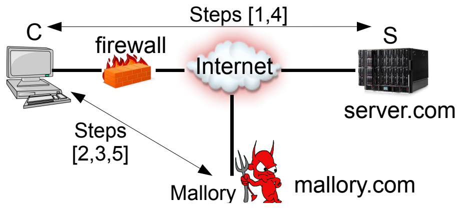
ACM Reference Format:
ACM Trans. Info. Syst. Sec. V, N, Article A (January YYYY), 30 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

TCP is the main transport protocol over the Internet, ensuring reliable and efficient connections. TCP is trivially vulnerable to man-in-the-middle (MitM) attackers; they can intercept, modify and inject TCP traffic [Joncheray 1995]. Despite significant possible threats, a common assumption is that MitM capabilities are difficult to obtain; this assumption is demonstrated by OWASP's list of top ten security risks [The Open Web Application Security Project (OWASP) 2013] where the majority of attacks do not require MitM capabilities.

Even without cryptographic defenses against MitM, network protocols should be secure against weaker - but common - *off-path* attackers. Off-path attackers are weaker than MitM attackers since they cannot eavesdrop on packets sent to others; however, they can send 'spoofed' packets, i.e., packets containing fake source IP address.

There is a widespread belief that it is not feasible for an off-path attacker to *inject* traffic into a TCP connection. The reasoning is that (modern) TCP implementations randomize the 32-bit sequence number [Gont and Bellovin 2012], and most implementations also randomize the 16-bit client port [Larsen and Gont 2011]; hence, in order to successfully inject data to the TCP stream, the off-path adversary seems to have



Steps:
1. Puppet establishes a `victim-connection' between C and S by embedding an object from www.server.com (HTTP referrer is www.mallory.com).
2. Mallory learns the client port in victim-connection.
3. Mallory learns the server's sequence number in victim-connection.
4. Puppet requests to embed some HTML page from www.server.com (e.g., in an iframe).
5. Mallory impersonates S and injects a spoofed HTML page with script in response to the puppet's request in step 4.

Fig. 1. Network Model and Attack Outline.

to guess valid values to both fields. This belief is stated in RFCs and standards, e.g., in RFC 4953, discussing on TCP spoofing attacks (see Section 2.2 of [Touch 2007]). 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 [Morris 1985] and Bellovin [Bellovin 1989; 2004].

We present attacks that allow an off-path adversary to learn the client port and sequence numbers, and thereby inject traffic to the TCP connection. More specifically, we present *four* attacks, two for learning the port, and two for learning the sequence numbers. We show that the attacks can be used modularly, i.e., each of the port-learning attacks can be used with each of the sequence-number learning attacks. This flexibility is important, since different attacks have different prerequisites as well as different performance properties.

Our attacks exploit subtle properties of legitimate and popular implementations of the IP, TCP and HTTP protocols. In particular, consider the two sequence-number learning attacks. The first attack relies on the client usage of *global IP-ID counter*, where the sender chooses the IP-identifier field using a global counter. This allows the off-path attacker, who is able to communicate with the client (victim), to learn when the client sends a packet to another destination. We use this side-channel to infer on the sequence number of the TCP connection. Although the global IP-ID counter side-channel is specific to the client's operating system, this side-channel is common, e.g., it exists in all recent versions of the Windows operating system.

In contrast, our second sequence-number learning attack avoids the global IP-ID counter assumption. Instead, it exploits a common - and legitimate - behavior of browsers (according to the HTTP specification), which was introduced in the early versions of browsers and still exists in the modern browsers (up-to-date versions of Chrome, Firefox and Internet Explorer). This TCP injection technique is independent of the victim's operating system.

TCP injections allow many attacks. In particular, we show how to circumvent the basic browser's same origin policy defense and inject spoofed objects such as scripts and web-pages to a connection with a victim web-site. We also show how the attacker can force these objects into the client's cache (e.g., at the browser or proxy) for a long time (even years), and form a persistent cross-site scripting or web-spoofing attack.

1.1. Network Settings and Attack Outline

Figure 1 illustrates our network model and outlines our attacks. Mallory, the attacker that we consider, is an *off-path (spoofing) attacker*. Mallory cannot observe traffic sent to others; specifically, she cannot observe the traffic between a client C and a server S. However, Mallory can send *spoofed packets*, i.e., packets with fake (spoofed) sender IP address. Mostly due to ingress filtering [Baker and Savola 2004; Ferguson and Senie 2000; Killalea 2000], IP spoofing is less commonly available than before, but it is still

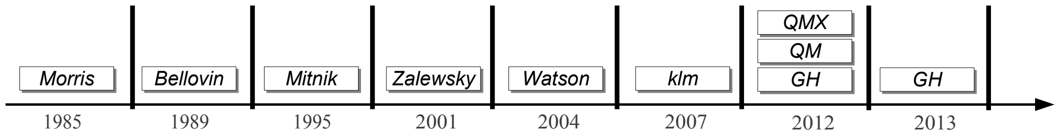


Fig. 2. A Time-Line of TCP Injection Attacks.

possible with many ISPs[1], see [Advanced Network Architecture Group 2013; Beverly et al. 2009; Ehrenkranz and Li 2009]. Mallory can connect to the Internet via an ISP that allows IP-spoofing; hence, the spoofing attacker model is realistic.

Our attacks require that the user enters to Mallory's web-site. This allows Mallory to run a restricted script in the user's browser sandbox. Specifically, this script is restricted by *same origin policy* [Barth 2011; Ruderman 2001] and can only communicate via the browser, i.e., request (and receive) HTTP objects (no access to IP, TCP or HTTP packet headers). Following [Antonatos et al. 2008], we refer to such attacker-controlled scripts as *puppets*. Puppets are usually easier to obtain and control compared to *zombies*, since browsers normally run scripts automatically upon opening a web-site, while zombies require installation (of malware).

Organization. The reminder of this section discusses related works and summarizes contributions. Section 2 presents a modular overview of our attacks, and compares to related attacks. Section 3 presents and evaluates our client-port de-randomization techniques. Sections 4 and 5 present the sequence number learning techniques, Section 6 compares between them and provides an empirical evaluation. Sections 7 and 8 present exploits, Section 9 presents defenses, and Section 10 concludes.

1.2. Related Works

1.2.1. TCP Injection Attacks. In Figure 2 we present a 'time-line' of important off-path attacks and security improvements for TCP, which we discuss below.

The time-line begins in 1985 with Morris' TCP injection attack [Morris 1985] and Bellovin's seminal paper from 1989 [Bellovin 1989]. Both works showed that predictable sequence numbers allow to perform off-path attacks on TCP communication.

In spite of these warnings, until 1995, most TCP implementations still used trivially-predictable *initial sequence number (ISN)*. This changed only after the infamous attack by Mitnick on Shimomura [Shimomura and Markoff 1995], that utilized a TCP injection. After this attack, many implementations randomized their ISN choices. However, at 2001, Zalewski found that most implementations still used 'sufficiently predictable' ISNs [Zalewski 2001], allowing TCP injections. Since then, most or all major implementations were fixed to ensure sufficiently-unpredictable ISNs, e.g., following [Gont and Bellovin 2012].

In 2004, Watson presented the first off-path injection attack which worked for random ISN [Watson 2004]. This attack injected a 'RST' packet, breaking up a connection; the idea was that it suffices for a RST packet to have a sequence number in the 'window', not necessarily exactly the next-expected sequence number. Watson's attack was, therefore, relevant for long-lived connections using known client (and server) ports and addresses, and in particular, against BGP. Prevention of this attack was one of the main reasons that many TCP implementations, with the notable exception of Windows, began to also randomize client ports, as recommended in [Larsen and Gont 2011].

[1]Apparently, there is still a significant number (16%-22%) of ISPs that do not perform ingress filtering and allow their clients to spoof an arbitrary, routable source address [Advanced Network Architecture Group 2013; Beverly et al. 2009].

The first 'proof of concept' showing that off-path attackers may still be able to inject data to TCP connections, even with randomly-chosen client ports and ISNs, was in [klm 2007]. This attack worked only against Windows machines, connected directly to the Internet, rather than via firewall, and did not handle concurrent connections.

The attacks in [Qian and Mao 2012; Qian et al. 2012] require malware running on the client machine (albeit with limited privileges). These attacks use the malware to identify a 'victim connection' by monitoring the client's system (e.g., by executing 'netstat'); then, the attacker learns the server's sequence number by sending spoofed packets with various sequence numbers, the malware identifies when a packet is accepted by the client (has valid a sequence number) by reading system counters. A significant challenge in practice, that was not considered in [Qian and Mao 2012; Qian et al. 2012], is that many clients connect to the Internet via NAT devices; in this case, the external port (allocated by the NAT) is likely to differ from the one observed by the malware, which runs on the client. Moreover, assuming a local malware agent to perform web-spoofing (injection of false content) is a strong requirement. In fact, a malware can often display false content to the user and trick him or her into believing that it is genuine (without complex TCP injections); this is a common attack vector. The off-path attacks that we present in this paper require only a puppet running on the victim machine, in contrast to [Qian and Mao 2012; Qian et al. 2012].

This paper consolidates and extends our two previous, preliminary publications [Gilad and Herzberg 2012a] and [Gilad and Herzberg 2013b]. We extend the scenarios where the attacks can be employed, improve the performance of the attacks considerably, and extend the empirical evaluation on connections with popular servers.

1.2.2. Other related works. TCP injection attacks were key to some of the most well known exploits, specifically, attacks against address-based *client* authentication, e.g., see [Shimomura and Markoff 1995; Bellovin 2004]. As a result, address-based client authentication has become essentially obsolete, and mostly replaced with cryptographic alternatives such as SSH and SSL/TLS.

Web security still relies, to large extent, on the *Same Origin Policy (SOP)* [Barth 2011; Ruderman 2001], i.e., on domain/address-based *server* authentication. Many attacks are based on circumventing SOP; however, these attacks are usually based on *implementation bugs*, mostly in the sites, and some in the browsers or middle-boxes; see [Zalewski 2011]. Especially related to our attacks is the HTTP response splitting attack [Klein 2004], which exploits the loose separation between HTTP responses.

De-randomization of client port selection is also relevant to UDP communication. Assuming a zombie agent behind the same NAT of the victim, [Herzberg and Shulman 2012] presents port de-randomization attacks. In contrast, in this paper we require only a puppet rather than a zombie agent, do not assume that the client connects to the network via a NAT, and of course de-randomize a TCP port rather than a UDP port.

1.3. Contributions

The basic contribution of this work is in showing that TCP injections are very practical, in terms of both efficiency and of requirements: our attacks do not depend on malware, and two of our attacks (one for learning the port and the other for learning the sequence number) do not assume a specific implementation of TCP/IP, in contrast to previous works.

In the short term, patches should be deployed to prevent our attacks; we present such defenses in this paper. As a long term result, we hope that this work will help to promote the use of cryptographic defenses, providing strong security even against MitM attackers, rather than assuming that attackers only have off-path capabilities.

Table I. Off-Path TCP Injection Attacks: Building Blocks. In brackets: the main requirements.

	Learn Connection Four-tuple	Learn Sequence Numbers	Exploit
[klm 2007]	Active probing for connection (Windows client, no firewall)	Exploit global IP-ID counter impl., both seq. # obtained (Windows client)	None
[Qian and Mao 2012] [Qian et al. 2012]	Monitor connections, e.g., with netstat (Malware)	Read client system counters, server's seq. # obtained (Malware; in [Qian and Mao 2012] also seq. # checking firewall)	XSS, CSRF, phishing (no TLS/SSL)
This Work	Establish connection, exploit sequential port allocation impl. (Puppet, Windows client)	Exploit global IP-ID counter impl., both seq. # obtained (Puppet, Windows client)	As above plus web-cache poisoning (Puppet, no TLS/SSL)
	Establish connection, client port de-randomization (Puppet, client behind firewall)	Exploit browser behavior, server's seq. # obtained (Puppet, no TLS/SSL, 'tolerant' browser)	

We identify the main challenges for off-path TCP injections, and build our attacks modularly, with independent modules handling different phases and tasks. This allows our modules to be used independently in other attacks.

Lastly, our web-cache poisoning exploit significantly enhances known exploits for TCP injections, allowing long-term effects on the victims (even for years).

2. A MODULAR ATTACK SCHEME

In this section we present a modular scheme for a TCP injection attack, breaking the attack into three separated tasks:

◇ Learn Connection Four-Tuple. The attacker learns the four parameters of a TCP connection between a client and a server, i.e., their respective IP addresses and ports.

◇ Learn Sequence Number(s). The attacker learns the current sequence number, for packets sent from the server to the client. In some attacks, the attacker also learns the sequence number for packets from the client to the server.

◇ Exploit. The final task of the attack, is to successfully exploit the exposed TCP parameters, and inject traffic in a meaningful way. This task is far from trivial and may also depend on the properties of the previous tasks, e.g., required length of connection.

Existing works on off-path TCP injections [klm 2007; Qian and Mao 2012; Qian et al. 2012] discussed these tasks, but did not present them as modules addressing different tasks required for TCP injections. By explicitly identifying these three modules/tasks, it is easier to understand and compare new TCP injection modules. It also becomes easy to identify cases where a new module, improving the solution to one task, can improve an earlier attack. On the other hand, protocols and systems should be designed to make *each* phase (task) infeasible.

The following subsections present the three tasks; for each task, we present two possible implementations which are novel to this work. We also compare with implementations of the same task in previous attacks. Table I summarizes our discussions below.

2.1. Learn Victim-Connection's Four-Tuple

The first task is to identify a TCP connection to attack, i.e., a 'victim-connection'. In [klm 2007], the adversary actively scans the client machine for an existing connection with a particular server. As indicated in [klm 2007], this technique is typically detected and blocked by firewalls. In [Qian and Mao 2012; Qian et al. 2012], the attacker runs a rogue application (malware) on the client machine. The malware monitors connections that the client has with servers, e.g., by executing netstat.

The attacks presented in this article rely on a weaker assumption: that the user's browser runs a *puppet*, i.e., a malicious script, executed within a sandbox, after being automatically downloaded from the attacker's web-site (mallory.com) to which the user had innocently entered. This puppet *establishes* the victim connection (step 1 in Figure 1). Therefore, Mallory (the attacker) knows the client and server IP addresses, as well as the server's port. It is only left to identify the client port (step 2 in Figure 1).

We present two implementations for this module. The first implementation is simple, but limited: it assumes *sequential port allocation*; this allows the attacker to guess the correct client-port of the connection that the puppet establishes. Sequential port allocation is used by many client machines, in particular those running Windows.

However, many operating systems attempt to avoid predictable port allocation; indeed, this is recommended in RFC 6056 [Larsen and Gont 2011], which presents several recommended algorithms for 'unpredictable port allocation'. Our second port derandomization technique successfully attacks (de-randomizes) the *Simple Hash-Based Port Selection* (SHPS) algorithm from [Larsen and Gont 2011], implemented in Linux. Being the default Linux implementation, this algorithm applies to many clients, e.g., mobiles running Android or clients that connect to the Internet through NATs (which often run the Linux kernel). This technique is more complex, but also very efficient; we refer details to Section 3.

2.2. Learn Sequence Numbers

The next task required for TCP injection, after identifying the victim-connection (including client port), is to expose one or both of the connection's sequence numbers; this is step 3 in Figure 1. Knowledge of the server's (or client's) sequence number allows the attacker to inject spoofed data to the connection, impersonating as the server (or client, respectively). Observing the sequence numbers directly from traffic requires an on-path attacker (i.e., eavesdropping capability). Off-path TCP injection requires a technique to infer the sequence number(s).

2.2.1. Operating-System Specific. In the attack of [klm 2007], the adversary exploits the global IP-ID counter implementation in Windows. The attacker sends 'ping' packets to the client and observes the difference in the IP-ID field in the 'pong' responses; this difference tells the number of packets that the client had sent to other parties (since each packet increments the global IP-ID counter). In this attack [klm 2007], the attacker sends to the client spoofed probe packets (that appear to be from the server); the client responds to a probe only if it specifies an invalid server sequence number, i.e., outside the client's flow-control window. The client sends the responses to the server, but the attacker learns whether the client responded by observing the IP-ID field in the 'pong' packets that she receives from the client.

The first implementation for the 'sequence number learning' module that we present in this paper resembles to the attack of [klm 2007], but improves it to work in two important cases: (1) when the client connects via a firewall device which blocks incoming 'ping' packets; and (2) when there is benign traffic between the client and multiple servers, such traffic makes using the side-channel more challenging since each packet that the client sends increments the IP-ID, regardless of the destination. We describe our implementation of this module in Section 4.

2.2.2. System Monitoring. In the sequence number inference attacks [Qian and Mao 2012; Qian et al. 2012] the attacker runs a malware on the client machine and uses that malware to monitor system variables. In these attacks the attacker sends spoofed packets to the client machine. Each packet specifies a different sequence number. The observation in [Qian and Mao 2012] is that if the sequence number is not close to the value that the client expects, then some network firewalls will discard the packet. The observation in [Qian et al. 2012] is that the client will respond to the packet only if its sequence number is in the flow-control window. Both attacks use the malware to read system counters, which tell whether the client received the attacker's packet ([Qian and Mao 2012]) or responded to it ([Qian et al. 2012]).

2.2.3. Browser Behavior. The second sequence-number learning technique that we present, called ‘Inject and Observe’, uses a different approach than the previous attacks. Inject and Observe, assumes a standard TCP/IP stack and does not rely on an operating system specific leakage, e.g., via the IP-ID field (cf. to [klm 2007]). Additionally, since we assume only a puppet running on the client machine, Mallory (the attacker) cannot read system files (cf. to [Qian and Mao 2012; Qian et al. 2012]).

In the Inject and Observe technique, described in Section 5, Mallory sends to the client data which is spoofed as coming from the server in response to queries that the puppet sends. Inject and Observe relies on a very common, tolerant, browser behavior while parsing HTTP responses; this behavior allows the puppet to retrieve the injected data when buffered in the flow-control window (maintained by the browser). The data contains the server’s sequence number that Mallory guessed; hence, when read by the puppet, Mallory learns a valid sequence number.

The tolerant behavior is allowed by the HTTP 1.1 specification [Fielding et al. 1999] and implemented in the current versions of Chrome, Firefox and Internet Explorer.

2.3. TCP Injection: Exploits

Once attackers identified the victim connection and one or both of its sequence numbers, they can inject spoofed data to the communication stream. The final building block of the attack is an application of the injection capability (steps 4 and 5 in Figure 1), typically to inject a malicious web-object (script or page) into the connection. The malicious object may be cached, and the attacker can make sure that it stays in cache (theoretically forever), as we show and empirically evaluate in this article.

Web-cache poisoning allows the attacker long-term use of many exploits, including cross-site scripting (XSS), cross-site request forgery (CSRF) and phishing, bypassing state of the art defenses such as the content security policy [Stamm et al. 2010]. We suggested the XSS, CSRF and phishing exploits in [Gilad and Herzberg 2012a; 2012b] and they were also suggested in parallel in [Qian and Mao 2012]. In Section 7 we present the XSS and CSRF exploits, Section 8 presents the persistent variant of these exploits and how it can be used to perform phishing attacks.

3. LEARN VICTIM-CONNECTION’S FOUR-TUPLE

The first phase in performing an off-path TCP injection is to identify the victim-connection. As described in Section 2.1, Mallory (the attacker) uses the puppet to establish the victim-connection; therefore, Mallory knows the client’s address as well as the server’s address and port[2]. In this section we describe how Mallory learns the fourth parameter of the TCP connection four tuple: the client port. We describe two techniques: first for the simpler case that the client operating system uses sequential port allocation (implemented in Windows), and then focus on de-randomization of the Simple Hash-Based Port Selection algorithm, a common and standardized method to allocate client ports (implemented in Linux).

3.1. Sequential Port Allocation

In some systems, such as Windows, learning the client port is trivial, since port numbers are assigned consecutively (for all destinations). Mallory uses the puppet to open a connection to her remote sites, a.mallory.com and b.mallory.com, 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_a and p_b, the client ports used in the connection to her sites. If $p_b = p_a + 2$, then she identifies that the connection to S is via port $p_a + 1$ (otherwise she retries).

[2]Our initial discussion in this section assumes that the server has one IP address; in practice, large servers often have multiple addresses, we refer to this issue in Subsection 3.5.

3.2. Randomized Port Allocation and SHPS

It is widely accepted that sequential port allocation is insecure, and that the client port should be 'unpredictable' to an off-path attacker. RFC 6056 [Larsen and Gont 2011] presents five recommended client port selection algorithms to secure against off-path adversaries.

We focus on the third recommended algorithm in [Larsen and Gont 2011]: 'Simple Hash-Based Port Selection' (SHPS). SHPS is the default port selection algorithm of the Linux OS kernel since version 2.6.15, i.e., since the year 2006; it is therefore embedded in all Android versions and many NAT devices. Extensive deployment at the NAT level makes SHPS the de facto port selection algorithm for many clients, even if the client machine does not use this algorithm.

SHPS chooses a pseudo-random initial port for each destination (server) IP-address; a new connection between the client and that destination uses the current port which is then incremented, i.e., *a per destination port-counter*. SHPS is expected to be secure against off-path adversaries, since these are not aware of the initial port. However, in this section we show how an off-path attacker (Mallory) *can* predict the next port assignment by SHPS.

In Subsection 3.3 we present *port elimination and testing*. This is a simple technique, where Mallory *eliminates* (or 'marks') a port p, and the puppet *tests* if the next-assigned port was supposed to be p. By repeating this for many ports, eventually a match happens, allowing the puppet to predict the next-assigned port. Then, in Subsection 3.4, we present a *meet in the middle* optimization method, which applies elimination and testing *concurrently* to multiple ports, improving the efficiency of the de-randomization technique. We complete this section with Subsection 3.5, which discusses practical challenges and presents an empirical evaluation.

3.3. Port Elimination and Testing

We now describe a method for eliminating a client port p, and then testing if p is the next port to be assigned by the client's port-selection algorithm. Specifically, Mallory sends a spoofed SYN packet, specifying the client's IP address and port p as the source, to the server (S). This causes S to open a (pending) connection with port p of the client. As a result, the server will *refuse* additional SYN packets from port p of the client, namely, port p is *eliminated*. After port p was eliminated, the puppet tries to establish a new connection with S; the *response time* indicates whether the selected client port was eliminated. We now provide the details of our technique, illustrated in Figure 3.

In the first step (see Figure 3), Mallory sends to S a spoofed TCP SYN with the source address of C and from port p, which is the port that Mallory tests. When S receives this SYN, it creates a connection entry for $\langle$C:p,S:server-port$\rangle$ and assigns it the SYN-Received state; S then sends a SYN+ACK response to C. We assume that the client connects to the Internet via a firewall; such defenses are very common and often deployed at the host level (often as an integrated part of the operating system, or software suites for end-point security), or the gateway level, e.g., many modern NAT devices (even low-end) are integrated with a firewall. A firewall that connects C to the network (see illustration in Figure 1) will discard the unsolicited SYN+ACK packet from S (as C did not send a matching SYN)[3]; as a result, S stays at the SYN-Received state for a relatively long time (in our experiments below, this was typically $10 - 20$ seconds for popular web-servers).

In the second step (see Figure 3), the puppet establishes a TCP connection with S, by requesting the browser to embed an image from S in the puppet's web-page. The

[3]This is typically a default firewall rule; without it, the client may be exploited in scanning attacks such as the idle-scan [Sanfilippo 1998].

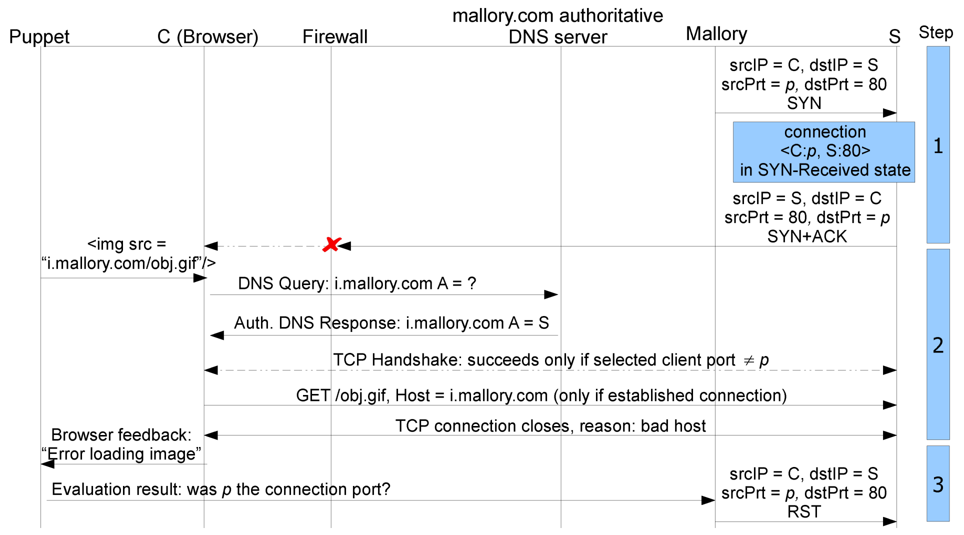

Puppet
C (Browser)
Firewall
mallory.com authoritative DNS server
Mallory
S
Step
srcIP = C, dstIP = S
srcPrt = p, dstPrt = 80
SYN
connection
<C:p, S:80>
in SYN-Received state
1
srcIP = S, dstIP = C
srcPrt = 80, dstPrt = p
SYN+ACK
<img src =
"i.mallory.com/obj.gif"/>
DNS Query: i.mallory.com A = ?
Auth. DNS Response: i.mallory.com A = S
2
TCP Handshake: succeeds only if selected client port ≠ p
GET /obj.gif, Host = i.mallory.com (only if established connection)
TCP connection closes, reason: bad host
Browser feedback:
"Error loading image"
Evaluation result: was p the connection port?
srcIP = C, dstIP = S
srcPrt = p, dstPrt = 80
RST
3

Fig. 3. SHPS De-Randomization, Elimination and Testing Technique.

puppet requests an image from domain i.mallory.com, an attacker-controlled domain that is mapped to the IP address of S. The prefix counter i ensures that each request uses a unique sub-domain; this prevents reuse of an existing connection.

In the third step (see Figure 3), the puppet evaluates whether p was the connection port and informs Mallory. Evaluation is based on the response time, which is very different in the two cases - when the client tried to use the 'eliminated' port p, and when it used a different port.

If the client port selected by the operating system is not p, then C and S will establish a TCP connection, over which the browser will request the image. Usually the servers will refuse the request immediately, since the browser specifies in the a HTTP request's 'Host Header' a sub-domain of mallory.com and not the server's domain (e.g., s.com), due to the DNS mapping in step 2 (of i.mallory.com to S's IP address). Hence, most servers will close the connection (others might return a HTTP not found message), and the puppet will receive an error feedback from the browser after roughly two C-S round-trip times (RTTs), which is normally much less than one second.

In contrast, if the operating system selects p as the client port, then C will try to establish a connection, i.e., send a SYN packet, with the same source port (p), but, almost always, with a different sequence number than that set by Mallory in the first step. Therefore, S will discard this packet, see TCP specification [Postel 1981] page 69; the TCP connection will not be established. The client operating system will retry to establish the connection several times and return a feedback to the puppet only after several seconds. Often, this feedback will be due to exceeding the maximal number of retransmission attempts; alternatively, the connection may be established, but again only after several seconds, when the server closes the (spoofed) pending connection. In both cases, the delay is much larger than in the case that the client used a port different from p.

In order to 'clean up' after testing port p, Mallory sends a reset (RST) packet that corresponds to her spoofed SYN; this releases the server's resources in case that these are still allocated to the connection.

If the port p was indeed the port that the client tried to use (in connecting to S), then the attacker can now predict that on the next connection-open by the puppet to S,

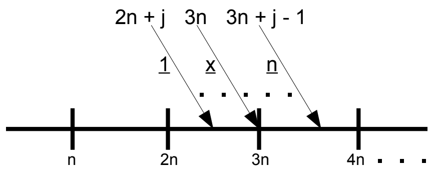


Fig. 4. SHPS De-Randomization, Meet-in-the-Middle Optimization. At the top are ports allocated by the operating system, illustrated by the arrows; numbers with underscore mark the connection number. At the bottom are ports that Mallory eliminates.

port $p+1$ will be used. Otherwise, the attacker can repeat the process, until eventually successful. This would work - but not efficiently; since the port field is 16 bits long, the expected number of attempts until success is 2^{15}.

3.4. A Meet-in-Middle Optimization

In this subsection we present a *meet-in-middle optimization*, that significantly reduces the time and communication involved in the SHPS de-randomization process. In order to improve de-randomization performance, Mallory uses the puppet to establish multiple connections to the server and eliminate ports simultaneously.

Let π denote the number of possible ports for a connection between C and S. Since the port field is 16-bits long, $\pi \leq 2^{16}$ (π is often smaller than 2^{16}, see next subsection).

In the first phase of the de-randomization process, Mallory performs port-elimination (described above) on $n = \lceil\sqrt{\pi}\rceil$ ports, specifically, the ports $\{i \cdot n\}_{i=0}^{n-1}$. In this phase, the puppet establishes n connections to S, which we number by the order of establishment. See illustration in Figure 4.

In order to use the puppet to establish multiple connections to the server S, Mallory must circumvent the fact that browsers which support HTTP 1.1 would normally send multiple requests to the same server using the same 'persistent' connection. Circumvention of this mechanism is performed by mapping DNS records of attacker controlled domains to the server IP, as described in the previous subsection: the puppet requests objects from $\{i.\text{mallory.com}\}_{i=0}^{n-1}$, Mallory controls the DNS records for these domains and maps them to the IP address of S (see illustration in Figure 3). Browsers use domain-names to identify servers and not IP addresses; hence, this technique, which we verified on Chrome, Firefox and Internet Explorer, opens n new connections to S.

We assume that during the de-randomization process the client does not create an independent connection with S. According to the SHPS algorithm, client port allocation is sequential; therefore, exactly one of these connections will use a port eliminated by Mallory, the wait-time for feedback from that connection is significantly longer than in other connections and this is identified by the puppet; let x denote the number of that connection. Since the puppet had established $n - x$ connections after connection x, the current value of the operating system's client-port counter is $kn - x$ for some integer $1 \leq k \leq n$. This completes the meet in the middle phase.

The following phase of the de-randomization process is an exhaustive search that is performed in iterations to identify the current port from the remaining n possibilities. In each exhaustive search iteration, Mallory performs the elimination process simultaneously on half of the remaining ports and the puppet requests only a single object. If the port allocated by the operating system is one of those tested by Mallory, then the feedback from S to the puppet is delayed. Since each iteration eliminates half the possibilities, the exhaustive search requires $\lceil\log_2 n\rceil$ iterations to complete.

3.4.1. Analysis. The maximal number of *simultaneous* connections that the puppet may open depends on the version of the browser; this value is at least 15 in all modern browsers and typically increases with new releases, see [Browserscope 2012]. Based on this, we estimate the amount of transmitted data and time required to perform SHPS de-randomization.

Mallory sends $n = \lceil \sqrt{\pi} \rceil \leq \sqrt{2^{16}} = 256$ spoofed SYNs during the meet in the middle phase and a similar number of SYNs during the exhaustive search phase, i.e., overall at most 512 packets of 40 bytes each, in total 20KB.

The puppet requests $n \leq 256$ objects during the meet in the middle phase; since the browser allows simultaneous requests for 15 objects, the number of 'request-iterations' during the meet in the middle phase is at most $\lceil \frac{256}{15} \rceil = 18$. Each iteration takes roughly two C-S round-trip times (RTTs). In total, the iterations take 36 RTTs, which are 3-7 seconds (for typical Internet RTTs of 100-200 milliseconds). The exhaustive search phase has $\lceil \log_2 n \rceil \leq 8$ iterations which perform one after the other, i.e., requiring 16 RTTs, i.e., typically about 1.5-3 seconds.

3.5. Real-World Challenges and Evaluation

This subsection describes practical challenges in performing SHPS de-randomization and presents an evaluation of our technique on connections with popular web-servers.

3.5.1. Challenges.

A. Multiple Server IP Addresses. Large web-sites often map their domain names to multiple IP addresses; this allows load distribution on several server-machines and shorter round-trip time to the client, who connects to a physically close server. However, this induces a difficulty on our attack since we wish to learn the port-counter associated with the specific server IP address that the client uses.

Usually, the attacker can identify a small set of possible IP addresses just by the client's physical location or ISP (e.g., our ISP provides six addresses for www.google.com). These possibilities are eliminated with a short validation phase at the end of the de-randomization process: after Mallory learns the value of the port-counter for some server IP, she sends a spoofed SYN to the server using the next port; the puppet tries to retrieve an object from the server's domain (cf. to attacker controlled domains as described in Subsection 3.3). If the puppet receives a feedback after a relatively long delay, indicating that the port allocated by the client operating system is eliminated, then Mallory de-randomized the port counter for the correct server IP address; otherwise, Mallory performs the de-randomization process again for another IP address.

B. SYN Flooding. Our SHPS de-randomization technique requires sending n SYN packets during the meet in the middle phase, i.e., create up to 256 'half-open' connections. This might be identified by some web-servers as a SYN flooding attack [Eddy 2007], i.e., an attempt to clog the server's connections backlog; we now discuss the defenses suggested in [Eddy 2007] that might be triggered and influence our technique. We note that the situation where a single client IP address has multiple connections to a single server is very common; in particular, in the case of a clients'-network behind a single NAT device, dozens and even hundreds of clients may share the same external IP address (of the NAT).

The first defense is to filter connections from the client's IP address. This defense blocks our attack, but fails to mitigate SYN flooding when the attacker can spoof her address. Moreover, this defense seems inappropriate to support the common NATed-network scenario (described above) and may be abused by such IP-spoofing attackers to deny service from legitimate clients by sending spoofed SYNs using their addresses.

The second defense is to use SYN-cookies [Bernstein 1996], i.e., avoid state keeping at the web-server until the TCP handshake completes. In this case, the server will reply to the client's SYN even if it uses a port that was 'eliminated' by Mallory, thereby foiling our technique. SYN-cookies encode the connection state in the server's sequence number, which is returned to the client in the SYN+ACK packet; this allows the server to reconstruct its state when receiving the following ACK packet from the client. However, SYN-cookies are not widely used since they come 'at a high price': they allow the server to use only one of four options for maximal segment size (MSS), which may degrade service for clients. Furthermore, SYN-cookies reduce the entropy in the server's sequence number which may allow an attacker to guess its value, see [Kaminsky 2011].

Finally, the server may reduce its TCP timers; this will release server's resources faster, but may deny service from clients with long response time. This defense does not prevent our attack, but forces a tighter time constraint on it: the puppet must perform all requests until timeout or Mallory must 'refresh' her spoofed SYN packets.

All the defenses above have disadvantages which may discourage servers from deployment. A typical solution, suggested in [Eddy 2007] and [Lemon 2002], is a hybrid approach: the server keeps a small state for each connection, e.g., using SYN cache [Eddy 2007], and employs one of the defenses described above when it identifies a SYN flooding attack. Indeed, the majority of servers in our experiments did not employ IP filtering or SYN-cookies even after we sent the required 256 spoofed SYNs; this allowed us to de-randomize the client port with high success rates (see next).

3.5.2. Evaluation.

Setup. We evaluated our technique on connections with popular web-sites, specifically, the top 1024 sites in the Alexa ranking [Alexa Web Information Company 2013]. We used a Linux client (kernel version 3.2.0) with a local IP-tables host level firewall (version 1.4.12). The Linux kernel uses the range $[32768, 61000]$ for choosing client ports; this is a significantly smaller range than all possibilities for the 16-bit port field. This observation helps to improve the run time of the de-randomization technique.

We placed the attacker and client machines in the same network, which allowed the attacker to send packets to the Internet using the client's IP address (in reality, the attacker would connect through an ISP that does not perform ingress filtering, see discussion in Section 1.1). The client and attacker connect through different physical interfaces of a network switch, this prevents the attacker from observing packets to/from the client, i.e., attacker is off-path. The client and attacker connect through 10Mbps link to the Internet.

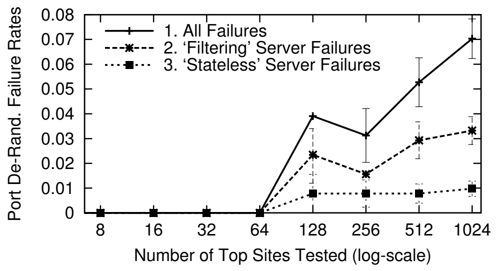


Fig. 5. Evaluation of SHPS De-Randomization. *Failure rates* as a function of web-site popularity. Rates are the average of two runs: one when puppet runs on Firefox and the other on Chrome. Error-bars mark standard deviations.

We performed our experiments when the puppet runs in Mozilla Firefox (version 16.0.2) and Google Chrome (version 23.0.1271.64). We verified our port prediction by executing netstat on the client side and observing the selected client port in the following connection.

Results. Figure 5 shows the failure rates as a function of web-site popularity. SHPS de-randomization failed for approximately 7% of the 1024 websites that we tested (see Figure 5 line 1); i.e., a 93% success rate.

We also measured the deployment and effect of the SYN flooding defenses described above: when failed to de-randomize the port, we tested whether the web-site allows new connections from the client, i.e., whether the client's IP address is filtered; these 'filtering' servers were approximately 3% of the servers (see Figure 5 line 2). If the client's IP address was not filtered, we tested whether the client can connect to the server using an 'eliminated' port; if he can, then the server either has a short timer, that had elapsed by the time we tested the correct port, or uses the SYN-cookies defense (i.e., server does not 'remember' the spoofed SYNs); these 'stateless' servers were approximately 1% of the servers (see Figure 5 line 3). SHPS de-randomization for other servers (approximately 3%) failed due to other errors; e.g., sometimes we were not able to retrieve the server's IP address (due to DNS filtering at the network or ISP level).

4. SEQUENCE NUMBER LEARNING: EXPLOIT THE IP-ID SIDE-CHANNEL

We now proceed to the second building block (and attack phase), as in the design presented in Section 2; namely, we show how to learn the sequence numbers of the victim-connection. We assume that Mallory (the attacker) has the parameters of the victim-connection, in particular, that she identified the client port; e.g., by executing one of the techniques described in Section 3 (or other methods, see Table I).

Many client machines run the Windows operating system, which uses a global IP-ID counter implementation[4]. This implementation allows an off-path attacker, who is able to communicate with the client, to learn the number of packets that the client sends to other destinations; this is performed by observing the difference in the IP-ID value in packets that the attacker receives from the client. We utilize this IP-ID based side channel to learn the server's sequence number. Furthermore, recent Windows clients run a non-standard variant of TCP, where the acknowledgment number is also employed to validate packets (this mechanism exists since XP SP2); we extend our technique to identify the acknowledgment number as well.

We present a two phase attack: first, in Subsection 4.1 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 Subsection 4.2, 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 Subsection 4.3 we discuss implementation and real world challenges of the technique.

Section 6 provides an empirical evaluation and compares with our second technique for learning the server's sequence number (presented in the following section).

4.1. The Sequence Number Test

This subsection presents the *sequence number test* that allows Mallory to learn whether some sequence number, $\tilde{\sigma}$, is in the flow-control window (*wnd*) that C keeps for bytes received 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 a sequence number *outside* the recipient's *wnd* are invalid, and cause the recipient to send a duplicate Ack for the last valid packet that he received (see TCP specification [Postel 1981] bottom of page 69); this duplicate Ack allows the sender to re-synchronize with the current sequence number that the recipient expects. In contrast, if the sequence number is *within wnd*, then the receiver does not send any response; the reasoning is that 'Acking' a valid empty Ack packet will start a never-ending series of acknowledgments.

The *sequence number test*, illustrated in Figure 6, 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 observes the IP-ID value in the response.

[4]The IP-ID field is attached to every IPv4 packet.

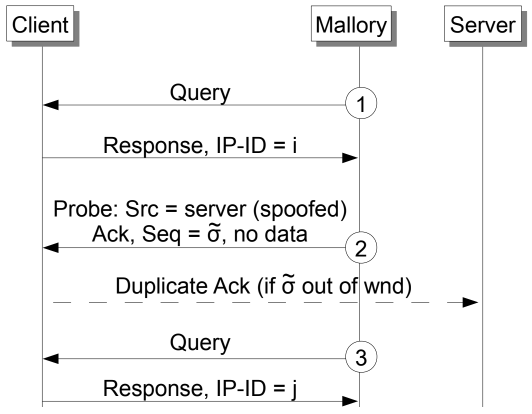

Client
Mallory
Server
Query
1
Response, IP-ID = i
Probe: Src = server (spoofed)
Ack, Seq = $\tilde{\sigma}$, no data
2
Duplicate Ack (if $\tilde{\sigma}$ out of wnd)
Query
3
Response, IP-ID = j

Fig. 6. Global IP-ID Counter Side Channel. Sequence number test.

In Subsection 4.3 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 mallory.com). In the second step, Mallory sends to 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 6), 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 $\tilde{\sigma}$ is in C's *wnd* if $x = 1$, i.e., C did not send any packet between the two queries.

Analysis. Mallory performs the sequence number test for $\tilde{\sigma} \in \{i \cdot |wnd|\}_{i=1}^{\frac{2^{32}}{|wnd|}}$ until she identifies a sequence number in C's *wnd*; i.e., a linear search. The typical size of *wnd* is 2^{16}, therefore, at most 2^{16} tests are required. Once a sequence number in *wnd* is detected, Mallory performs a binary search to identify the beginning of *wnd*, i.e., the current sequence number.

Each empty Ack packet (probe) is 40B long; query packets are also short, 41B long (see implementation in Subsection 4.3). In each test Mallory sends $40+2\cdot 41 = 122$B (one probe and two queries). Therefore, in this phase Mallory sends at most $2^{16} \cdot 122\text{B} = 8\text{MB}$ of data.

4.2. The Acknowledgment Number Test

In recent versions of Windows (since XP SP2) 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 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 7 illustrates the 'acceptable' acknowledgment numbers (i.e., C's transmission window).

Similarly to the sequence number test (above), we construct a three step *acknowledgment number test* where the first and last steps provide Mallory 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 [Postel 1981] (page 72). The relevant statement refers to an acknowledgment packet that carries data ('non-empty' packet) and has a valid sequence number; i.e., success in the previous server sequence-number learning phase is required to initiate this phase. The TCP specification distinguishes between two cases regarding the acknowledgment number in the packet, see illustration in Figure 7.

Case 1. The packet contains a stale Ack (gray area in Figure 7), or acknowledges data that was sent, but not already acknowledged (black area in Figure 7). In this case according to the specification, the recipient is supposed to continue processing the packet regularly (see [Postel 1981]). 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 7), the recipient discards the packet and immediately sends a duplicate Ack that specifies his current sequence number, i.e., NXT in Figure 7.

Hence, when C receives a packet that specifies an acceptable sequence number, i.e., in C's TCP 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) silently discards it.

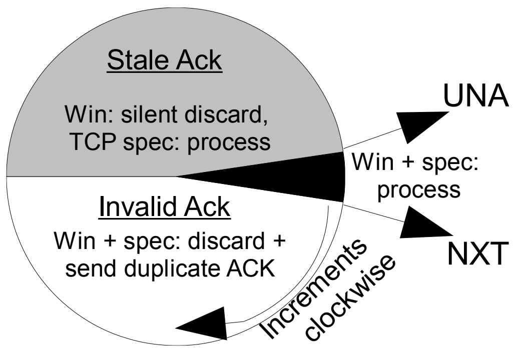


Fig. 7. Ack Number Map. UNA is the lowest unacknowledged sequence number, NXT is the next sequence number that C will send. The 32-bit Ack field is cyclic.

The probe which we use in the test specifies the acknowledgment number that Mallory tests and has two important properties derived from the observation above: (1) the probe packet specifies a sequence number that is in C's *wnd* (discovered in the previous server sequence-number learning phase); (2) the probe packet carries data (i.e., a 'non-empty' packet).

Analysis. Let $\tilde{\alpha}$ denote the acknowledgment number that Mallory tests. If the test indicates that C did not send any packet between the two queries, then $\tilde{\alpha}$ is below UNA (i.e., in the gray area in Figure 7). Otherwise, Mallory concludes that $\tilde{\alpha}$ is above UNA (i.e., in the black or white area in Figure 7). The gray and white areas in Figure 7 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 un-acknowledged sequence number, it is the Ack number C expects.

The 32-bit length of the Ack field implies that there are 32 iterations. Each packet that Mallory sends in this phase is 41B long (an Ack packet with 1 byte of data); therefore, Mallory sends less than 1KB of data in this phase.

4.3. Implementation and Real-World Challenges

In this subsection we discuss the implementation of the tests presented above and the practical challenges that we cope with.

4.3.1. Implementing Test Queries. The sequence and acknowledgment number tests which we described above 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 the user 'surfed' to mallory.com): a query is a short packet that Mallory sends to C, e.g., containing 1B of data (the total length is 41B), the response is C's TCP ACK packet sent 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.

4.3.2. Detecting Packet Loss. Mallory must identify when test-packets (queries, responses or probes) are lost, since otherwise the corresponding and following tests will yield a wrong result.

Mallory detects a lost probe by repeating all tests that indicate that C 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's flow-control window. Additionally, approximately 16 of the 32 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 and data transmitted during 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.

4.3.3. Errors in Tests. Both tests use the global IP-ID counter 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 occur only in tests where C does not respond to the probe: if C sends a packet, independent of the probe, between responding to Mallory's test-queries, 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 above, there are only few tests where the probe does not yield a response, only in these tests such an error is possible.

We handle errors in the two phases differently. During the sequence number learning phase, Mallory tests 2^{16} possible sequence numbers; however, only one of these tests can yield an error result (the one that tests a valid sequence number, i.e., in C's *wnd*). Hence, the probability for 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 acknowledgment number learning phase, 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 attack.

5. SEQUENCE NUMBER LEARNING: EXPLOIT TOLERANT BROWSERS

In this section we present a complementary technique to the one presented in Section 4. In contrast to Section 4, we assume a standard TCP client, i.e., complaint with the TCP specification ([Postel 1981]), such clients include Linux and Android devices. We exploit an *under-specification of HTTP 1.1* [Fielding et al. 1999], which allows browsers (compliant with the HTTP specification) to render un-parsable web objects.

At the end of this phase Mallory learns the 32-bit server's sequence number; this allows her to send spoofed data to the client (impersonating as the server).

Subsection 5.1 provides required background, explaining how browsers handle HTTP responses that they receive. Subsection 5.2 describes our sequence number learning technique. Subsection 5.3 describes how to set an important parameter of the attack and evaluates its value in practice. Section 6 provides an empirical evaluation and compares with our previous technique.

5.1. HTTP Request/Response Handling

As of HTTP 1.1 [Fielding et al. 1999], clients can send multiple requests to the same server in a single ('persistent') HTTP connection; furthermore, clients can send these requests in *pipeline*, i.e., without waiting for response to one request before sending the next request. In order to allow browsers to match between each response and the corresponding request, the responses are sent by the server, exactly in the order in which the client had sent the requests.

More specifically, the browser (client) keeps a FIFO queue of pending HTTP requests for each connection, and handles them one by one, as follows. In order to handle the (oldest) request, the browser reads the bytes in TCP's flow-control window (allocated per-connection) when they become available. The browser expects to find the matching response in the beginning of TCP's flow-control window. Next, the browser parses the response as per [Fielding et al. 1999], embedding it in the web page. This process continues until there are no more requests awaiting reply from the particular connection.

Unfortunately, the HTTP standard [Fielding et al. 1999] does not specify what the browser should do when the flow-control window contains data which is *not* a valid HTTP response. We tested the current versions of the three most popular browsers (Internet Explorer, Firefox and Chrome), and *all* of them handled this situation as follows: the browsers treat *all available data* in the flow-control window as payload of a response with the following 'default' HTTP header:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=us-ascii
Content-Length: available-data-size
```

The browser returns this 'response' to the requesting module, normally, the browser's rendering engine or a script/applet. The browsers *do not break* the existing TCP connection, and continue processing responses to requests sent over it[5]. The following subsection explains how we exploit this behavior to learn the server's sequence number.

5.2. Inject and Observe

In this subsection we present the server sequence number learning technique which is illustrated in Figure 8. The technique has two steps: *(1) Inject* and *(2) Observe*.

Inject step. In this step, Mallory injects data into the stream of HTTP responses sent from the server (S) to the client (C). This data is 'observed' (read) in the following step, which allows Mallory to determine the server's sequence number.

Let wnd denote the browser's flow-control window for the connection and $|wnd|$ denote its size. In order to inject the data, Mallory sends to the browser $\frac{2^{32}}{|wnd|}$ packets,

[5]This behavior may have been adopted to simplify 'debugging' of servers that implement HTTP pipelining incorrectly.

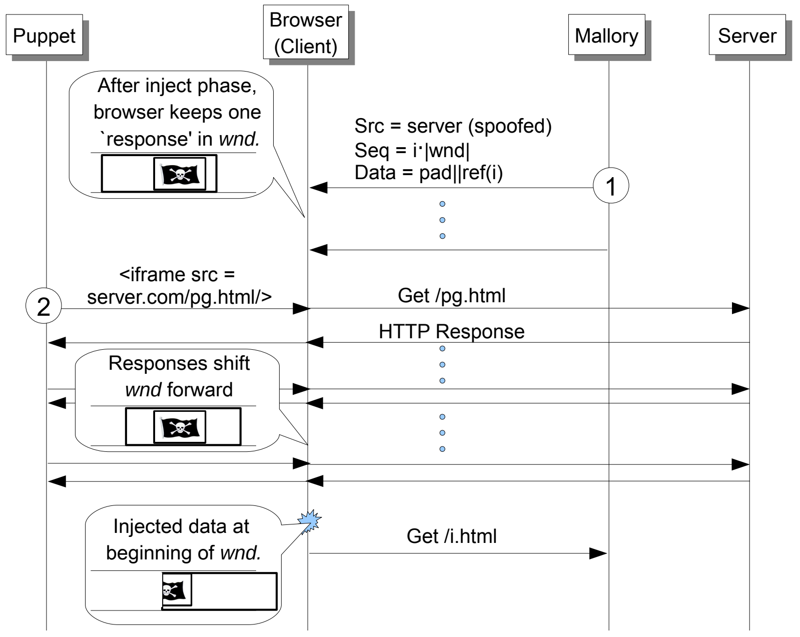

Puppet
Browser (Client)
Mallory
Server
After inject phase, browser keeps one 'response' in *wnd.*
Src = server (spoofed)
Seq = i·|wnd|
Data = pad||ref(i)
1
<iframe src = server.com/pg.html/>
2
Get /pg.html
HTTP Response
Responses shift *wnd* forward
Injected data at beginning of *wnd.*
Get /i.html

Fig. 8. The Inject and Observe Technique.

spoofed to appear to be from S (on its victim-connection with C). The i^{th} packet has server sequence number $i \cdot |wnd|$, and contains as payload $pad||ref(i)$, where *pad* is an easily-removable 'pad'[6] and $ref(i)$ is an iframe reference to a web-page at Mallory's site:

```
<iframe src = "mallory.com/i.html" />
```

Hence, exactly *one* of these packets contains a 'valid' server sequence number, which falls within *wnd*; all the other packets are discarded by C.

Actually, this description was a bit simplified, since TCP also validates the *acknowledgment number* specified in received packets. Specifically, TCP ignores packets whose acknowledgment number is for data not yet sent (relative to the cyclic space of acknowledgment numbers), see TCP specification [Postel 1981] page 72 and Figure 7. Therefore, if we select the acknowledgment number randomly, there is a 50% chance that it would be ignored. The solution is simple: Mallory sends *two* packets for each sequence number, one specifies $Ack = \alpha$ for some $\alpha \in \{0, 2^{31}\}$, and the other specifies $Ack = \alpha + 2^{31}$; this ensures that the Ack number in *one* of the two packets is valid (gray or black areas in Figure 7). Hence, exactly one of the packets will contain 'good' sequence and acknowledgment numbers, and its data is saved in the flow-control window (*wnd*). Figure 8 illustrates C's victim-connection *wnd* after the inject step.

During the 'inject' step, the puppet ensures that there is always at least one request waiting for reply in the browser's queue; this by generating two initial requests and sending a new request when a response arrives (Figure 8 does not show the simultaneous requests for readability). The reason that one request must always be enqueued is that when there are no pending requests, some browsers clear the flow-control window (those will discard the injected data).

[6]The length of the pad and its use will become clear when we present the following 'observe' step.

Observe step. In this step, the puppet makes prevalent requests to the server, until it reaches the data injected by Mallory in the previous step. Similarly to the previous 'inject' step, the puppet maintains at least one request enqueued until this phase completes (see Figure 8). Each response that arrives at C shifts *wnd* forward; once a sufficient number of responses arrive, such that there is no gap of unreceived bytes between the injected data (buffered in *wnd*) and the last response, the browser will also read the injected response, expecting it to match the following request; see illustration of C's victim-connection *wnd* during the observe step in Figure 8. In fact, the last response would (usually) overwrite part of the pad at the beginning of the injected data; assume that the pad is at least as long as the server's response[7], hence, some of the pad and all of $\mathit{ref}(i)$ would remain in *wnd* and be read and *rendered* by the browser; see illustration of C's victim-connection *wnd* after the observe state in Figure 8. As described in the previous subsection, in all browsers that we tested, the remaining injected data is handled as a regular response with a default header.

When the browser renders the injected response, it will attempt to resolve *ref(i)* and retrieve *i.html* from Mallory's web-site (see Figure 8); providing to Mallory the value of i. This allows Mallory to compute the next server sequence number that C expects.

5.3. Pad Length

The length of the pad attached to each 'inject-step' packet is a significant factor in the applicability of the attack: the shorter the pad is, the less data that Mallory sends and the shorter the attack time is. However, the pad must be at least as long as the responses that the puppet receives from the server (in the observe step) in order to guarantee success (the attack may succeed with shorter pads, see analysis below). In this subsection we evaluate the length of the pad that Mallory should use in practice to launch the Inject and Observe attack on connections with popular web-servers[8].

In order to identify requests with short responses we combine three techniques:

Browser-Cache. The puppet requests an object from the web-server prior to the attack; this object is then cached by the browser. When the puppet requests again the same object (during the observe step), the browser will send a HTTP If-Modified-Since request to the web-server and receive a short HTTP Not Modified response.
Non-Existing Objects. The puppet sends requests to non-existing objects, the web-server replies to these requests with HTTP Not Found responses.
Web-Site Crawling. Find the shortest object available on the web-site prior to the attack; the puppet requests this object during the observe step[9]

We measured the minimal response size in popular web-sites. Figure 9 shows that for many popular websites (more than 80% of the 1024 most popular sites) a pad of 600B guarantees success. Of the three techniques above, we found that for more than 90% of the web-sites exploiting the browser cache produced the shortest response.

5.3.1. Analysis. The Inject and Observe technique requires Mallory to send a number of packets that is linear to the number of sequence numbers. Specifically, Mallory sends during the 'inject' step $2\frac{2^{32}}{|wnd|}$ packets. For typical flow-control window size ($|wnd|$) of 2^{16}B, we find that Mallory sends 2^{17} packets. The number of additional packets sent during 'observe' step is negligible.

Each packet is perpended with a pad, assume that the pad length is l while the length of the response that the puppet receives (during the observe step) is r. The

[7]The pad may, for example, be of the form $\{0\}^m||1$, where m is the length of the longest possible response.
[8]We use Alexa's web-site popularity rank [Alexa Web Information Company 2013].
[9]In order to avoid caching of the short object, the puppet specifies a random query string in the request URL.

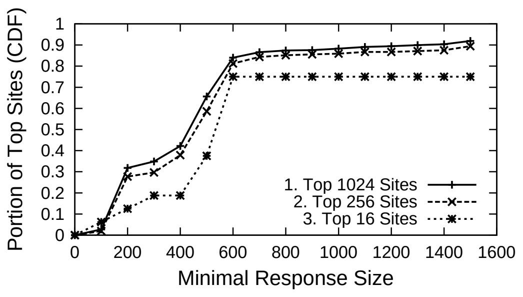


Fig. 9. A CDF of Minimal HTTP Response Length for Popular Websites.

probability for success depends on l and r: (1) if $l \geq r$, then the server's response will not overrun the reference (iframe) to Mallory's site, which guarantees success. (2) If $l < r$, then the response will not overrun the reference to Mallory's site only if the current sequence number (that the server uses in the connection) is lower than the sequence number that Mallory specified in her injected response by at least $kr - l$ bytes and no more than kr bytes (for some integer k): meaning that after the server sends k responses (of r bytes), the client's flow-control window reaches to Mallory's response, but does not overrun the reference to her web-page. The probability for success for this case is $\frac{r-(r-l)}{r} = \frac{l}{r}$. Summarizing both scenarios, the probability for success in the entire attack is $\min\{1, \frac{l}{r}\}$.

Assume that Mallory uses a pad of at most $600B$ (the pad is shorter if the response length is shorter than 600B); according to our measurements (illustrated in Figure 9) and analysis above, the probability for success using Inject and Observe on connections with the 1024 most popular websites is 91%, and on average Mallory sends 47MB in each attack. In Section 6.3 we empirically evaluate the success rate of Inject and Observe on connections with these websites.

6. COMPARISON AND REAL WORLD EVALUATION

In this section we evaluate and compare the two sequence number learning techniques presented in Sections 4 and 5. Table II summarizes our comparison.

6.1. Comparison

Requirements. The most significant difference between the two sequence number learning techniques that we presented is their assumption on the client-side: the applicability of the IP-ID Side Channel technique in Section 4 depends on the client operating system (requires a Windows client), whereas the Inject and Observe technique in Section 5 depends on the client browser (requires a 'tolerant' browser, such as Chrome, Firefox or Internet Explorer, see evaluation in Section 6.3). Furthermore, as we discussed in Section 4.3 and show in our evaluation in Section 6.2, the greater the client's communication rate is, the greater the probability for error and failure of the IP-ID Side Channel technique (since the IP-ID increments faster).

In terms of their requirements from the server-side, the IP-ID Side Channel technique in Section 4 does not assume anything about the server-end; however, in many cases TCP servers close idle connections. Therefore, Mallory uses the puppet to send requests over the victim-connection to the server in order to avoid 'timeout tear-down' and perform the attack, which requires a significant time to launch (see evaluation below). In the common case, that the server is a 'web-server', using the puppet to send such multiple requests over one connection requires a persistent HTTP connection,

Table II. Comparison of Sequence Number Learning Techniques.

	Client Requirements	Server Requirements	Learned Seq. Number	Time and Data	Success Rate
IP-ID Side Channel	Windows client	Persistent connection	Both client and server	48 seconds, 8MB	71%[10]
Inject and Observe	'Tolerant' browser	Persistent connection, no SSL/TLS	Server only	2.5 minutes, 47MB	78%

namely, that the server does not close the connection after handling a request (this is the default HTTP configuration according to the specification [Fielding et al. 1999]).

The Inject and Observe technique in Section 5 assumes a browser client and a HTTP connection. Furthermore, the technique relies on two additional assumptions on the web-server side:

(1) Use HTTP without cryptographic protection (i.e., no HTTPS). SSL/TLS defenses will not allow Mallory to inject data to the application (browser will discard the spoofed data before HTTP parsing).
(2) Support persistent HTTP connections. This allows the puppet to send several requests over the same connection; if the server does not support this property, then the connection will close after the first response arrives.

Empirical Measurements. Figure 10 illustrates the portion of popular Internet servers that conform with these requirements. We found that *all* of the 1024 most popular web-sites[11] support HTTP connections (without HTTPS protection), see line 1. Furthermore, we found that 85% of the HTTP servers use persistent connections, this is illustrated by line 2 (this is the potential success rate of the attacks). Finally, we note that 3% of HTTP web-servers automatically redirect their users to a HTTPS version of the website (e.g., facebook, gmail), see line 3; however, most of these web sites (in particular, those listed above) send the redirect to HTTPS over a *persistent HTTP connection*, compare line 3 to line 4; such web-sites are *vulnerable* to our attacks, for example, as we show in Section 8.2 this allows Mallory to redirect users to her own malicious web site.

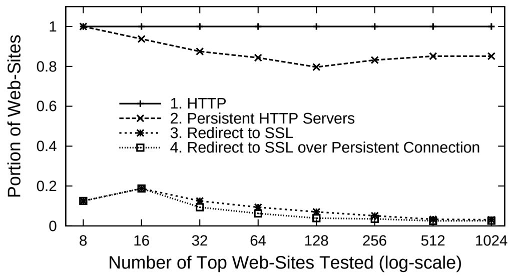


Fig. 10. Applicability of Our Sequence Number Learning Attacks. Measurements on connections with popular websites.

Learned Sequence Number. The two sequence number learning techniques produce different results: the IP-ID Side Channel technique in Section 4 provides both sequence numbers; this allows Mallory to inject spoofed data to the connection, impersonating as either the client or server. In contrast, the Inject and Observe technique in Section 5 only provides the server's sequence number. Although the exploits that we present in the following sections of this article only require injection to the client-side (i.e., require knowledge of the server's sequence number), the ability to inject data to the server's side may allow other types of exploits.

Time and Transmission-Volume. The time required to learn the sequence numbers is of practical importance: since both techniques rely on the aid of a puppet agent running

[10]Success rate depends on client transmission rate (see Figure 11); this measurement is while the average client transmission rate is 32 packets per second.
[11]We use Alexa's popularity rank in our evaluations [Alexa Web Information Company 2013].

in the user's browser, the user must stay in the malicious site (mallory.com) while the attack executes. We measured the average completion time of successful executions on connections with popular servers: the IP-ID Side Channel technique requires 48 seconds to complete (standard deviation is 14 seconds), whereas the Inject and Observe technique completes in approximately 2.5 minutes (standard deviation is 21 seconds). We believe that an attacker can often display content that would persuade users to stay in her site for the required amount of time while the puppet executes the attack in the background.

The time difference between the two results is since Mallory sends on average 47MB in an Inject and Observe attack, where packets are padded, and 8MB in the IP-ID Side Channel technique. See analysis in Sections 4 and 5.

6.2. IP-ID Side Channel: Evaluation

We evaluated the IP-ID Side Channel sequence number learning technique in two sets of measurements. We first evaluated the success rate of the technique as a function of the rate of independent packets that the client sends. As discussed in Section 4.3, the higher this rate is, the faster the global IP-ID counter increments and the lower the probability for success. Next, we evaluated the success rate of the IP-ID based technique on connections with popular web-servers.

Setup. In the first set of measurements, aimed to evaluate the success rate with benign traffic, we used our own server (in order to mitigate the uncertainty factor of server-side behavior). In this set of measurements we provided the adversary with the IP addresses and ports that describe the victim connection; the server in these measurements runs Apache (version 2.2.14). The setup in the second set of measurements is as in Section 3.5.2.

In both sets of measurements the client machine is an up-to-date (fully-patched) Windows machine, protected by Windows Firewall with default configuration.

To identify a successful exposure of the sequence numbers, we used the puppet to request a non-existing object from the server (after the learning process completes) and injected a spoofed HTTP OK response. If the puppet identifies the spoofed OK response, then we determined that the injection attack was successful, hence the attacker obtained the correct sequence numbers. Notice that since this evaluation method injects a response to the connection, it requires a non-SSL connection; as illustrated by line 1 in Figure 10, all 1024 popular web servers accept such (non-secure) HTTP connections.

Results. Figure 11 illustrates the probability for successful learning of sequence numbers for different client transmission rates and when the puppet runs on different browsers. The results show that this technique is independent of the client's browser, and effective for reasonable client transmission rates.

We next evaluated the success rate of the entire attack on communication with the 1024 most popular servers. We preformed this experiment while the client sends an average rate of 32 packets per second (independent of the attack), according to the Poisson distribution; we performed three executions of each attack using different browsers: Chrome (v23), Firefox (v16) and Internet Explorer (v9). Figure 12 illustrates success rate of the IP-ID based technique (lines 2-3) and compares with the potential success rate (HTTP servers that use persistent connections) which is illustrated by line 1. The figure compares the success rate of the sequence number learning technique in two scenarios: (a) when the client connects directly to the network, in this case the client-port is obtained by exploiting Windows incremental port allocation, see line 2; (b) when the client is connected through an IP-tables NAT (version 1.4.12) which uses the SHPS port selection algorithm, in this case the client-port is obtained by executing the port de-randomization technique (described in Section 3), see line 3.

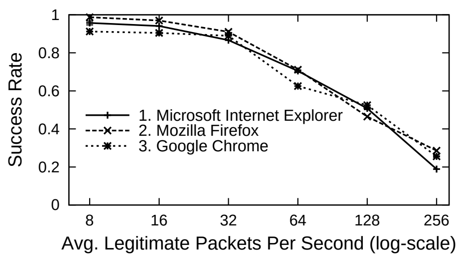


Fig. 11. IP-ID Sequence Numbers Learning Technique. Success rate as a function of client's transmission volume; each measurement is the average of 50 executions, error bars mark standard deviations.

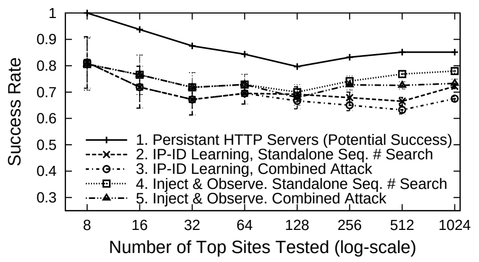


Fig. 12. Sequence Number Learning Techniques Comparison. Success rates as function of web-site popularity. Each measurement is the average of executions using different browsers (see text), error bars mark standard deviations.

6.3. Inject and Observe: Evaluation

The browsers Chrome (v23), Firefox (v16) and Internet Explorer (v9) conform with the ‘tolerant’ behavior which we exploit, namely, they render invalid HTTP responses by attaching a default header as described in Section 5.1. Similarly to the evaluation of our previous technique, we measured the success rate in two scenarios: (a) as a standalone component (the client-port is obtained using netstat); (b) as part of a complete injection attack (port de-randomization followed by Inject and Observe).

In order to identify a successful execution, similarly to the evaluation of the previous technique, after Inject and Observe completes the puppet requests a non-existing object from the web-server and Mallory injects a HTTP OK response; this allows the puppet to identify whether the response was successfully injected.

Setup. Our network setup is as in Section 3.5.2. The client machine which we used is a Nexus 10 tablet device, running Android version 4.2.2 which is based on the Linux kernel version 3.4.5. We used an Android device since Android is popular a platform for mobile devices. We empirically evaluated the Inject and Observe technique with Chrome and Firefox[12].

Padding Length. Mallory sends padded packets during the inject phase, the length of the pad effects the probability for success as well as the overhead of the attack, see discussion and analysis in Section 5.3. We first measured the required length of padding for each website in our tests, our measurements are described in Section 5.3 and illustrated in Figure 9. For our evaluation, we used padding of at most 600B for each connection: if we identified a response that is shorter than 600B from a website, then the pad length equals to the length of that response (while attacking a connection with that website), otherwise the pad length is 600B.

Results. Figure 12 compares the success rates (average of measurements using Chrome and Firefox) for both scenarios. The indicated success rate of Inject and Observe as a standalone component is approximately 78%, see line 4. The difference between our analysis in Section 5.3 (that indicated 90% success rate) and empirical result is since not all websites use persistent HTTP connections, which is a requirement of the Inject and Observe technique; this requirement is illustrated by line 1 (potential success). The combined attack, see line 5 has a similar success rate to that of standalone Inject and Observe, since our port de-randomization technique has a high probability for success (see evaluation in Section 3.5.2).

[12]Internet Explorer does not have a version for Android.

7. XSS AND CSRF ATTACKS

In this and the following section we present two types of exploits for TCP injections. The first, presented in this section, allows an off-path attacker (Mallory) to run a malicious script in the context of an arbitrary website of her choice; this is a new type of XSS attack [Klein 2005]. The second exploit, which we present in the following section, allows the same attacker to present spoofed web-pages for users. All exploits work in the same setting, illustrated in Figure 1.

7.1. XSS Injection (or: XSS of the Fourth Kind)

In a *Cross-Site Scripting (XSS)* attack, the attacker runs a malicious script with the permissions of scripts from a victim website. Klein identifies three kinds of XSS attacks [Klein 2005]. In *stored* XSS attack, the script is received from the victim website, 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 while the client visits a malicious website). Finally, in a *DOM-based* XSS attack, the browser receives the script directly from the attacking website; a browser vulnerability (bug) causes the browser to consider the script as coming from some other victim website.

Off-path TCP injection attacks allow a new, fourth kind of XSS attacks: *XSS injection*. In these attacks, the attacker *injects the cross-site script* to the connection between the browser and a third-party website. 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 website.

7.1.1. Attack Process. The XSS injection attack process is illustrated in Figure 1. The attack has five steps which we now describe, our puppet code is available online at [Gilad and Herzberg 2013a] with explanations and documentation that refer to the steps:

1. Establish a victim-connection from the client to a website (S). In order to establish the connection, the puppet requests to embed an object from the website.

2. Identify the four parameters of the victim connection; see Section 3.

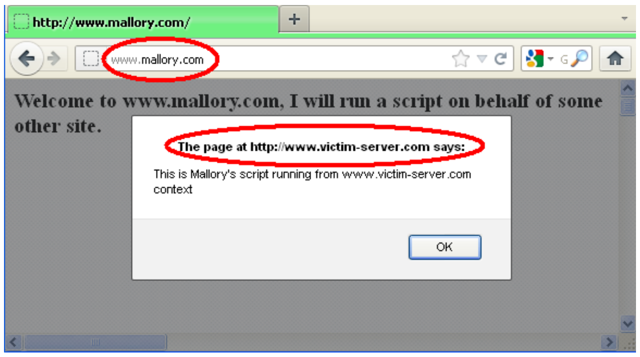
3. Learn connection sequence numbers. Puppet keeps the connection with S alive by periodically sending requests for small objects. During this time, Mallory learns the sequence numbers; see Sections 4 and 5.

4. Send a 'dummy' request. Puppet sends to S a request for some web page (over the same persistent victim-connection), e.g., using an iframe (see our code [Gilad and Herzberg 2013a]), and informs Mallory 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.

5. Send spoofed response. Mallory sends (injects) a spoofed response to the client, using the learned TCP parameters. The response contains a web page with the malicious script. The browser receives the spoofed response as if it was sent by S, hence, executes script with permissions of S. Figure 13 shows a successful run of this attack on Mozilla Firefox.

7.1.2. Vulnerable Connections and Websites. There are two types of immune websites to the XSS injection attack:

(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 learn the sequence numbers (attack step 2).
(2) Secured with SSL (HTTPS). This prevents Mallory from injecting her script to the connection (attack step 5).



Welcome to www.mallory.com, I will run a script on behalf of some other site.

The page at http://www.victim-server.com says:

This is Mallory's script running from www.victim-server.com context

Fig. 13. An XSS Attack. The Firefox 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.

These two requirements from the server-side are identical to those of Inject and Observe and were quantified in our evaluation of the sequence number learning techniques in Sections 5 and 4. In particular, line 2 in Figure 10 shows the potential success of the attack for connections with the 1024 most popular servers (according to Alexa's rank [Alexa Web Information Company 2013]). In fact, we used injection of spoofed HTTP response in order to evaluate the success rate of the sequence number learning techniques; therefore, Figure 12 illustrates the success rate of the XSS attack using both sequence number learning techniques that we presented in this paper.

Circumventing XSS-Defenses. The XSS injection attack allows Mallory to provide her own content, hence input sanitation techniques (common against 'stored' and 'reflected' XSS attacks) are ineffective against this attack. Furthermore, since Mallory specifies her own HTTP headers to the injected response, she can circumvent advanced proposed defenses which require new browser mechanisms. In particular, Mallory can circumvent the content security policy (CSP) defense [Jim et al. 2007; Stamm et al. 2010]: in this defense, the server specifies the CSP HTTP header which limits the execution of scripts; Mallory's spoofed response omits this header.

7.2. CSRF Exploit

As indicated in [The Open Web Application Security Project 2010], 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 suggested defenses against CSRF attacks, except for (few) defenses requiring extra user efforts for submission of each sensitive request (such as solving a CAPTCHA); see [Paul Petefish et al. 2011].

Note that since the attackers' (cross site) scripts can read the entire response that the user receives from the victim website, they are even able to circumvent advanced proposed defenses, which require new browser mechanisms. In particular, they can foil the *origin* header defense against CSRF attacks [Barth et al. 2008], as well as the common attachment of randomized hidden fields in the submission form.

8. WEB CACHE POISONING AND PHISHING ATTACKS

The exploits presented in Section 7 are limited: they can only run at the present moment and in the current victim-connection between the client and server. This motivates a persistent *web-cache poisoning* attack [Klein 2011; The Open Web Application

Security Project 2009]. Mallory can cache spoofed responses (for requests made by the puppet) at the browser, as well as at intermediate network proxies that will provide these spoofed responses to other users in the network. Mallory poisons the cache by specifying the cache control and related HTTP headers in her spoofed response; for example, the following HTTP headers cache the spoofed response for a day, and impede the browsers from refreshing the page:

```
Last-Modified: now
Cache-Control: public
Expires: tomorrow
```

Mallory can poison the web-cache with spoofed pages containing malicious scripts, that users of the web-cache will receive when they access the (poisoned) websites. These scripts will execute automatically and in the domain-context of the victim website; i.e., this is a persistent variant of the XSS exploit.

8.1. Replacing Existing Objects in Cache

Mallory may attempt to cache a page whose genuine version is already cached. In this case Mallory cannot use the puppet to request this page (as described above) since such a request will not generate a HTTP request 'over-the-wire' (since the cache will provide the response); hence Mallory will not be able to provide and cache a spoofed response. We now describe how Mallory can overwrite an existing object in the cache.

In the first step, the puppet requests a non-existing web-page, using a random URI, from the victim server (S); since the page does not exist, it is not in the cache. Hence, the request for the web-page is sent to S, and Mallory injects a spoofed web-page with a script in response; this script executes in context of S's domain and can therefore send a request to S with customized HTTP headers. In the second step, this script requests the object (e.g., page) that Mallory wishes to spoof, and includes the following HTTP header:

```
Cache-Control: max-age=0
```

This header orders the browser, as well as caching devices such as proxies, to avoid using their cache and send the request for the object to S. In the third step, Mallory injects a response that contains the spoofed object; the injected response includes the headers required to cache the object (see above). Since the injected object is more recent than the one that was already in-cache, it supersedes the existing object in the cache.

8.1.1. Evaluation. We tested the cache poisoning technique above (while replacing an object in cache) on Chrome (v23), Firefox (v16) and Internet Explorer (v9), as well as Squid (v3.3.4), a common open-source HTTP proxy. We were able to replace existing objects in cache for all four web-clients; however, the four implementations allow different maximal caching time-periods: Chrome and Firefox allowed us to cache a spoofed page for five years; Internet Explorer has a higher limit, and we were able to cache a page for ten years. In contrast to these very long caching periods, the Squid proxy allows to cache a page for no longer than a week (default configuration).

8.2. Web Spoofing and Phishing

Attackers can use TCP injections together with cache poisoning to perform *web spoofing*, which is a key to *phishing* attacks. Launching a web-spoofing attack is similar to the XSS exploit described in Section 7.1: the puppet requests a web-page (in step 4), which Mallory then injects (in step 5). In order to succeed in a web-spoofing attack, Mallory should provide the spoofed page as a response to a request made by the user (since then the page appears authentic to the user). Hence we use the persistent vari-

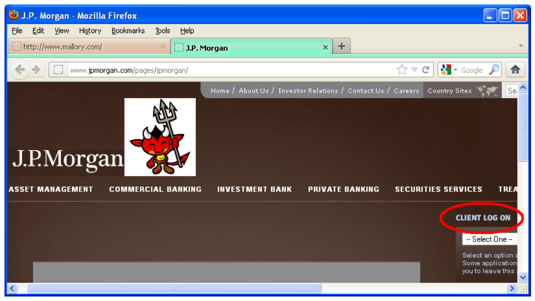

J.P. Morgan - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.mallory.com/
J.P. Morgan
www.jpmorgan.com/pages/jpmorgan/
Google
Home / About Us / Investor Relations / Contact Us / Careers Country Sites
J.P.Morgan
ASSET MANAGEMENT COMMERCIAL BANKING INVESTMENT BANK PRIVATE BANKING SECURITIES SERVICES TREA
CLIENT LOG ON
-- Select One --

Fig. 14. Web Spoofing Attack on Banking Website. Mallory added a devil image to the spoofed page.

ant of this exploit: Mallory caches the spoofed page which users then automatically receive when they request (and therefore expect) the genuine web-site.

Similarly to the XSS exploit, a requirement of this attack is that the initial web-page that the user receives, and which Mallory spoofs, is not protected by SSL. This assumption holds for most sites, which do not use SSL/TLS at all; in particular, according to our evaluation, all of the 1024 most popular websites have a HTTP homepage (which is not protected with SSL/TLS), see line 1 in Figure 10. An important type of vulnerable websites are those that invoke SSL/TLS, but only via a link, e.g, to the login page: namely, the home page http://bank.com contains a link to https://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 considered mandatory (e.g., for login). Another type of sites that can be spoofed with this attack are HTTP sites that automatically redirect the client to HTTPS; see line 4 in Figure 10.

Web-spoofing allows the attacker to circumvent the use of encrypted connections (SSL/TLS) by changing the links (and HTTP redirections) on the original site to point to a phony page on the attacker's site; modification of the original page may even be automated using tools such as SSL-strip [Marlinspike 2009]. Experiments with users show that they will often not identify the web spoofing attack, see [Herzberg and Jbara 2008]: many will not notice that the domain name had changed from the original website domain to that of the attacker's (note that the attacker's site is protected by SSL, i.e., the browser displays the padlock indicator). This exploit can expose users credentials (by spoofing login pages) and can also trick users to download malware (by spoofing download pages).

The reminder of this section presents the implementation of the web-spoofing attack on the J.P. Morgan bank homepage. We have also confirmed this attack on the following banks' websites: Goldman Sachs, Morgan Stanly and The Royal Bank of Scotland, all of these banks use a HTTP homepage (and persistent connections) and only switch to HTTPS when the client clicks the login button. Furthermore, we have confirmed the attack on the following banks web-pages: Deutsche Bank and Bank of America, which automatically redirect their HTTP users to HTTPS (for these sites we spoofed the redirection, sent in plain-text HTTP, to redirect the user to https://mallory.com).

8.2.1. Example: Spoofing J.P. Morgan. The J.P. Morgan bank website is an example of a sensitive site that uses HTTP persistent connections and its homepage is not protected by SSL (but the login page is protected). Hence, this website is vulnerable to the web spoofing attack. Figure 14 shows the result of a successful web spoofing attack: here, the user received the J.P. Morgan homepage that Mallory provided; the devil image (does not exist in the original page) indicates that this page is spoofed. The 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.

9. DEFENSES

We showed how an off-path attacker can learn the client port and sequence numbers of a TCP connection, allowing the exploits in Sections 7 and 8. This section presents client-end and server-end defenses for the attack vectors presented in this paper.

9.1. Client-End Defenses

9.1.1. Client-Port Prediction. Client-side operating systems should stop using the global counter and the popular Simple Hash-Based Port Selection algorithms, attacked in Section 3, and adopt a secure alternative. RFC 6056 [Larsen and Gont 2011] presents four other algorithms, which are therefore good candidates. The security of the client-port selection algorithm should be analyzed considering TCP mechanisms that might leak the state of connections (such as those exploited in this paper).

Furthermore, since many clients connect to the Internet via NAT devices, which modify the client port selection, effective mitigation of our attack requires modification of the port selection algorithm at the NAT level as well.

9.1.2. Sequence Number Prediction. In Section 4, we presented a technique for learning both sequence numbers of a TCP connection, by exploiting a side-channel provided by the global IP-ID counter implementation in Windows. The use of global IP-ID counter is long known as a weak implementation choice (e.g., exploited in the idle-scan [Sanfilippo 1998]) and should be mitigated by using a more secure alternative, such as choosing the value of the IP-ID in a pseudo-random manner.

Furthermore, we believe that the proprietary implementation choice of the Windows TCP stack: to use the acknowledgment number for validating incoming packets (as well as the standard sequence number validation), was introduced in order to increase security. However, this non-standard extension allowed us to learn *both* the server and client sequence numbers. Therefore, the implementation is best modified to conform with the TCP specification.

In Section 5, we presented the Inject and Observe technique for learning the server's sequence number, by exploiting a de facto browser behavior standard, which is not required by the HTTP specification: process and display of corrupt responses. We believe that browsers should modify this behavior and in the exception case that a response does not pass HTTP parsing, browsers should identify a problem in the TCP connection, send a TCP reset to the server and close the connection. This modification conforms with the HTTP standard and protects the user from attacks based on the Inject and Observe technique.

9.2. Server-End Defense

Our exploits, presented in Sections 7 and 8, inject spoofed data to the TCP stream. In order to ensure data integrity, cryptographic defenses should be deployed; i.e., servers should use SSL/TLS instead of relying on randomized initial sequence numbers for authentication.

Many sites already support SSL, and redirect their HTTP users to the HTTPS version of the website. This (unprotected) redirection is typically performed over a persistent HTTP connection (see Figure 10, compare lines 3 and 4) which allows our attacks. We recommend that such 'automatically-redirecting' servers will use a non-persistent HTTP connection for this purpose, this will prevent our attacks and will not effect most clients, since they typically close the HTTP connection when redirected.

10. CONCLUSIONS AND FUTURE WORK

We have shown how an off-path attacker can inject data into a TCP connection and evaluated the practical effect of such an attack on connections with popular servers.

We showed the need to fix two components of Internet communication: (1) the client port selection algorithm and (2) the way that browsers handle invalid HTTP responses; we suggested modifications that conform with the HTTP and TCP specifications, as well as provided further motivation to cease the use of global counters as IP-IDs.

This work continues a line of recent works on TCP injections [klm 2007; Qian and Mao 2012; Qian et al. 2012], showing that the folklore belief that TCP communication is immune to off-path attacks is incorrect. This motivates deployment of cryptographic protocols, such as SSL/TLS, to protect communication. We believe that more servers should adopt these defenses, even if communication is not considered sensitive.

This paper leaves directions for future work. First, a security analysis of the remaining four port selection algorithms suggested in [Larsen and Gont 2011] is required to identify the best alternative for the extensively deployed SHPS algorithm. Second, learning the client's sequence number, as shown in Section 4, allows data injection to the server-side, which may allow new exploits.

REFERENCES

ADVANCED NETWORK ARCHITECTURE GROUP. 2013. Spoofer Project. http://spoofer.csail.mit.edu/summary.php.

ALEXA WEB INFORMATION COMPANY. 2013. Top Sites. http://www.alexa.com/topsites.

ANTONATOS, S., AKRITIDIS, P., LAM, V. T., AND ANAGNOSTAKIS, K. G. 2008. Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure. *ACM Transactions on Information and System Security 12,* 2, 12:1–12:15.

BAKER, F. AND SAVOLA, P. 2004. Ingress Filtering for Multihomed Networks. RFC 3704 (Best Current Practice).

BARTH, A. 2011. The Web Origin Concept. RFC 6454 (Proposed Standard).

BARTH, A., JACKSON, C., AND MITCHELL, J. C. 2008. Robust Defenses for Cross-Site Request Forgery. In *ACM Conference on Computer and Communications Security*, P. Ning, P. F. Syverson, and S. Jha, Eds. ACM, 75–88.

BELLOVIN, S. M. 1989. Security Problems in the TCP/IP Protocol Suite. *Computer Communication Review 19,* 2, 32–48.

BELLOVIN, S. M. 2004. A Look Back at "Security Problems in the TCP/IP Protocol Suite". In *ACSAC*. IEEE Computer Society, 229–249.

BERNSTEIN, D. J. 1996. SYN Cookies. http://cr.yp.to/syncookies.html.

BEVERLY, R., BERGER, A., HYUN, Y., AND CLAFFY, K. C. 2009. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In *Internet Measurement Conference*, A. Feldmann and L. Mathy, Eds. ACM, 356–369.

BROWSERSCOPE. 2012. Browser Comparison. http://www.browserscope.org.

EDDY, W. 2007. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational).

EHRENKRANZ, T. AND LI, J. 2009. On the State of IP Spoofing Defense. *ACM Transactions on Internet Technology 9,* 2, 6:1–6:29.

FERGUSON, P. AND SENIE, D. 2000. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 (Best Current Practice). Updated by RFC 3704.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard).

GILAD, Y. AND HERZBERG, A. 2012a. Off-Path Attacking the Web. In *USENIX Workshop on Offensive Technologies*. 41 – 52.

GILAD, Y. AND HERZBERG, A. 2012b. Off-path attacking the web. *CoRR abs/1204.6623*.

GILAD, Y. AND HERZBERG, A. 2013a. Puppet Code (Java Script). http://u.cs.biu.ac.il/~herzbea/security/code/puppet-example.js.

GILAD, Y. AND HERZBERG, A. 2013b. When Tolerance Becomes Weakness: The Case of Injection-Friendly Browsers. In *Proceedings of the International World Wide Web Conference*.

GONT, F. AND BELLOVIN, S. 2012. Defending against Sequence Number Attacks. RFC 6528 (Proposed Standard).

HERZBERG, A. AND JBARA, A. 2008. Security and Identification Indicators for Browsers Against Spoofing and Phishing Attacks. *ACM Trans. Internet Techn. 8,* 4, 16:1–16:36.

HERZBERG, A. AND SHULMAN, H. 2012. Security of Patched DNS. In *European Symposium on Research in Computer Security*, S. Foresti, M. Yung, and F. Martinelli, Eds. LNCS Series, vol. 7459. Springer, 271–288.

JIM, T., SWAMY, N., AND HICKS, M. 2007. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the International Conference on World Wide Web*, C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, Eds. ACM, 601–610.

JONCHERAY, L. 1995. A Simple Active Attack Against TCP. In *Proceedings of the 5th Symposium on UNIX Security*. USENIX Association, Berkeley, CA, USA, 7–20.

KAMINSKY, D. 2011. Black Ops of TCP/IP. In *Black Hat conference*. http://dankaminsky.com/2011/08/05/bo2k11.

KILLALEA, T. 2000. Recommended Internet Service Provider Security Services and Procedures. RFC 3013 (Best Current Practice).

KLEIN, A. 2004. Divide and conquer. *HTTP Response Splitting, Web Cache Poisoning Attacks and Related Topics, Sanctum whitepaper*.

KLEIN, A. 2005. DOM Based Cross Site Scripting or XSS of the Third Kind. Tech. rep., Web Application Security Consortium: Articles. July.

KLEIN, A. 2011. Web Cache Poisoning Attacks. In *Encyclopedia of Cryptography and Security (2nd Ed.)*. 1373–1373.

KLM. 2007. Remote Blind TCP/IP Spoofing. Phrack magazine.

LARSEN, M. AND GONT, F. 2011. Recommendations for Transport-Protocol Port Randomization. RFC 6056 (Best Current Practice).

LEMON, J. 2002. Resisting SYN Flood DoS Attacks with a SYN Cache. In *BSDCon*, S. J. Leffler, Ed. USENIX, 89–97.

MARLINSPIKE, M. 2009. New Tricks for Defeating SSL in Practice. In *BlackHat DC*.

MORRIS, R. T. 1985. A Weakness in the 4.2BSD Unix TCP/IP Software. Tech. rep., AT&T Bell Laboratories. Feb.

PAUL PETEFISH, ERIC SHERIDAN, AND DAVE WICHERS. 2011. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

POSTEL, J. 1981. Transmission Control Protocol. RFC 793 (INTERNET STANDARD). Updated by RFCs 1122, 3168, 6093, 6528.

QIAN, Z. AND MAO, Z. M. 2012. Off-Path TCP Sequence Number Inference Attack. In *IEEE Symposium on Security and Privacy*. 347–361.

QIAN, Z., MAO, Z. M., AND XIE, Y. 2012. Collaborative TCP Sequence Number Inference Attack: How to Crack Sequence Number Under a Second. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 593–604.

RUDERMAN, J. 2001. Same Origin Policy for JavaScript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

SANFILIPPO, S. 1998. A New TCP Scan Method. http://seclists.org/bugtraq/1998/Dec/79.

SHIMOMURA, T. AND MARKOFF, J. 1995. *Takedown: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaws - by the Man Who Did It* 1st Ed. Hyperion Press.

STAMM, S., STERNE, B., AND MARKHAM, G. 2010. Reining in the Web with Content Security Policy. In *Proceedings of the International Conference on World Wide Web*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 921–930.

THE OPEN WEB APPLICATION SECURITY PROJECT. 2009. Cache Poisoning. https://www.owasp.org/index.php/Cache_Poisoning.

THE OPEN WEB APPLICATION SECURITY PROJECT. 2010. Cross-Site Request Forgery. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

THE OPEN WEB APPLICATION SECURITY PROJECT (OWASP). 2013. OWASP Top 10 Security Risks. https://www.owasp.org/index.php/Top_10_2013-Top_10.

TOUCH, J. 2007. Defending TCP Against Spoofing Attacks. RFC 4953 (Informational).

WATSON, P. 2004. Slipping in the Window: TCP Reset Attacks. Presented at CanSecWest, http://bandwidthco.com/whitepapers/netforensics/tcpip/TCP%20Reset%20Attacks.pdf.

ZALEWSKI, M. 2001. Strange Attractors and TCP/IP Sequence Number Analysis. http://lcamtuf.coredump.cx/newtcp/.

ZALEWSKI, M. 2011. *The Tangled Web: A Guide to Securing Modern Web Applications* 1st Ed. No Starch Press, San Francisco, CA, USA.