When Tolerance Causes Weakness: The Case of Injection-Friendly Browsers

Yossi Gilad
Bar-Ilan University, Israel
mail@yossigilad.com

Amir Herzberg
Bar-Ilan University, Israel
amir.herzberg@gmail.com

ABSTRACT

We present a practical off-path TCP-injection attack for connections between current, non-buggy browsers and web-servers. The attack allows *web-cache poisoning* with malicious objects; these objects can be cached for long time period, exposing any user of that cache to *XSS, CSRF* and *phishing* attacks.

In contrast to previous TCP-injection attacks, we assume neither vulnerabilities such as client-malware nor predictable choice of client port or IP-ID. We only exploit subtle details of HTTP and TCP specifications, and features of legitimate (and common) browser implementations. An empirical evaluation of our techniques with current versions of browsers shows that connections with popular websites are vulnerable. Our attack is modular, and its modules may improve other off-path attacks on TCP communication.

We present practical patches against the attack; however, the best defense is surely adoption of TLS, that ensures security even against the stronger Man-in-the-Middle attacker.

Categories and Subject Descriptors

C.2.2 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Protocols*

Keywords

Web and Network Security; Off-Path Attacks; Browser Security

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. However, it seems that MitM and eavesdropping attacks are relatively rare in practice, since they require the attacker to control routers or links *along the path* between the victims. Instead, many practical attacks involve malicious hosts, *without MitM capabilities*, i.e., the attackers are *off-path.*

There is a widespread belief that TCP communication is reasonably immune to off-path attackers; i.e., that such adversaries cannot *inject* traffic into a TCP connection. The reasoning is that TCP specifications and implementations were enhanced to provide security against such adversaries, who are incapable of eavesdropping to communication: modern TCP implementations randomize not only the 32-bit sequence number [14], but also the 16-bit client port [21]; in order to successfully inject data to the TCP stream, the adversary must provide valid values to both fields.

This belief is even stated in RFCs and standards, e.g., in RFC 4953, discussing on TCP spoofing attacks (see Section 2.2 of [33]). Indeed, since its early days, most Internet traffic is carried over TCP - and is not cryptographically protected, in spite of warnings, e.g., by Morris [23] and Bellovin [6, 7].

We present an attack that allows an off-path adversary to learn the (randomized) client port and sequence numbers, and thereby inject traffic to the TCP connection. The technique exploits subtle properties of the TCP specification, as well as common - and legitimate - behavior of browsers, which was introduced in the early versions of browsers and still exists in the modern browsers. Our TCP injection technique is independent of the victim's operating system, and allows the attacker to bypass the browser's same origin policy (SOP) defense [5, 28, 36]. In particular, this allows injection of web-pages and scripts in the context of a third-party web-server, and can be exploited for cross-site scripting (XSS), cross-site request forgery (CSRF) and phishing attacks without relying on a vulnerability in the web-server.

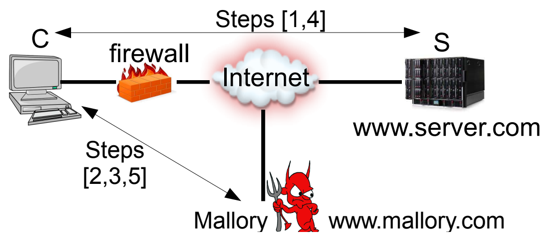
1.1 Network Settings and Attack Outline

Figure 1 illustrates our network model and outlines our attack. 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 [4, 11, 17], IP spoofing is less commonly available than before, but it is still possible with many ISPs[1], see [1, 8, 10]. Mallory can use an ISP that allows IP-spoofing; hence, the spoofing attacker model is (still) realistic.

Our attack requires that the user enters 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* [5, 28] and can only communicate via the browser, i.e., request (and receive) HTTP objects (no access to TCP/IP packet headers). Following [3], we refer to such attacker-controlled scripts as *puppets*. Puppets are usu-

[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 [1, 8].

WWW 2013, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.



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.

Figure 1: Network Model and Attack Outline.

ally 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).

The attack has five steps, which are illustrated in Figure 1. In order to circumvent the browser's same origin policy, the off-path attacker sends (in the final step of the attack) a forged response for a request that the puppet sends to S. This response may contain a malicious object, typically a script, which we refer to as *persistent XSS*, since it may be *cached*, for a long time (theoretically forever). Persistent XSS can circumvent *Content Security Policy (CSP)* [30] and other defenses (e.g., [15]), and perform *cross-site scripting (XSS)*, *cross-site request forgery (CSRF)* [32], *phishing, defacement* and more.

Organization. In the rest of the Introduction, we discuss related works and summarize our contributions. Section 2 presents a modular overview of our attack, and compares to related attacks. Section 3 presents our client-port derandomization technique. Section 4 shows how Mallory can learn the server sequence number. Section 5 discusses exploits, Section 6 presents defenses, and Section 7 concludes.

1.2 Related Works

1.2.1 Off-Path TCP Injection Attacks

TCP injections are easy for implementations that use predictable initial sequence numbers (ISNs). This was observed already by Morris at 1985 [23] and abused by Mitnick [29]. Later, at 2001, Zalewski found that most implementations still used predictable ISNs [35]. However, by now, most or all major implementations ensure sufficiently-unpredictable ISNs, e.g., following [14].

Since the adoption of randomized initial sequence numbers and until recently, TCP was widely believed to be immune to off-path attacks. One exception was the off-path attacks on TCP of [34], which disconnected BGP connections that use constant client ports. However, this attack was considered as reflecting a specific vulnerability of BGP availability. In particular, with the notable exception of Windows, most current operating systems adopted algorithms to make it harder to predict the client port. This and other countermeasures make this attack inapplicable today [9].

The first 'proof of concept' showing that off-path attackers may still be able to inject data to the TCP stream, even with randomly-chosen client ports and initial sequence numbers, was in [20]. This was recently improved to efficient off-path TCP injection attacks [13, 26, 27]. However, in this work, we significantly improve upon these works, as we now describe.

The attacks in [26, 27] require malware running on the client machine (albeit with limited privileges). These attacks use the malware to identify a 'victim TCP connection' by probing the client's system variables (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 [26, 27], 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 display false content to the user and trick him or her to believing it is genuine (without complex TCP injections); this is a common attack vector.

In contrast to [26, 27], the attack that we presented in [13] requires only a puppet running on the victim machine (similar to this work). However, [13] as well as [20] exploit a specific operating-system implementation of the TCP/IP stack. Specifically, the use of globally-incrementing allocation mechanisms for both ports and IP-IDs, as exists in the Windows operating systems. Although Windows is very popular, avoiding these requirements from the operating-system still significantly expands the base of vulnerable clients; furthermore, these specific weaknesses may be removed in future versions of the operating system (as suggested in our correspondence with Microsoft's security team).

1.2.2 Address-Based Authentication and SOP

TCP injection attacks were key to some of the most well known exploits, specifically, attacks against address-based *client* authentication, e.g., see [7]. However, as a result, address-based client authentication has become essentially obsolete, and mostly replaced with cryptographic alternatives such as SSH and SSL/TLS.

However, web security still relies, to large extent, on the *Same Origin Policy (SOP)* [5, 28], 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 middleboxes; see [36]. Especially related to our attack is the HTTP response splitting attack [18], which exploits the loose separation between HTTP responses.

Notice that using TCP injections to attack address-based server authentication is more challenging than using it to attack address-based client authentication. In attacks on address-based *client* authentication, the off-path attacker sends the initial SYN to open a new connection; hence, she knows the source and destination IP addresses and ports;

she 'only' needs to predict the server's sequence number. In contrast, to attack address-based *server* authentication, the off-path attacker must also identify the client's port.

1.3 Contributions

The basic contribution of this work is in showing that TCP injections can be very practical, in terms of both efficiency and of requirements: no dependency on malware or non-recommended implementations of TCP/IP, as in previous works. This has significant implications. In the short term, patches should be deployed to prevent our techniques; we present such defenses in this paper. Most significantly, we hope that this work will help promote the use of cryptographic defenses, providing strong security assuming MitM attackers, rather than assuming that attackers only have off-path capabilities.

We identify the main challenges for off-path TCP injections, and build our attack modularly, with independent modules handling different phases and tasks. This allows some of our modules to be used independently of others. As one important example, we present a technique for client port de-randomization. Specifically, we show how to predict the client port when the client's operating system uses the *Simple Hash-Based Port Selection (SHPS) Algorithm* recommended in [21]. Since SHPS is embedded into Linux, it is extensively used, e.g., by Android and NAT devices, which are often based on embedded Linux.

Our attacks, while efficient and practical, are non-trivial and based on in-depth understanding of the operation of TCP and HTTP. In particular, we exploit the fact that (current) browsers process invalid HTTP responses, by handling them as payload with a default response header. This behavior may have helped in debugging of early HTTP 1.1 implementations, but currently seems unnecessary and dangerous; browsers should be patched to avoid it.

Lastly, our cache-poisoning exploit significantly extends compared to known exploits for TCP injections.

2. A MODULAR ATTACK SCHEME

In this section we present a modular scheme for a TCP injection attack, breaking the attack into three separate tasks:
◇ Learn Connection 4-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. A non-trivial task is to find how to successfully exploit a TCP injection ability; this task may depend on the properties of the attack, e.g., required length of connection.

A modular scheme was not presented in previous off-path injection attacks [13, 20, 26, 27], however, these attacks follow our scheme. By explicitly stating the scheme, it is easier to understand new attacks and identify cases where a new module, improving the solution to one task, can improve an earlier attack; and, on the other hand, protocols and systems should be designed to make *each* step (task) infeasible.

The following subsections present the three tasks that compose the scheme; for each task, we compare our implementation of a building-block achieving the task, to implementations in previous attacks. Table 1 summarizes our discussions below.

2.1 Learn Connection 4-Tuple

The first task is to identify a TCP connection to attack, i.e., a 'victim-connection'. In [20], the adversary actively scans the client machine for an existing connection with a particular server. As indicated in [20], this technique is typically detected and blocked by firewalls. In [26, 27], 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`.

This work and [13] rely on a weaker assumption: that the user's browser runs a puppet, i.e., a malicious script, downloaded and executed automatically from the attacker's website, e.g., `www.mallory.com`, to which the user innocently entered. This puppet *establishes* the victim connection (step 1 in Figure 1). Therefore, Mallory (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).

In [13] the attack additionally assumes sequential port allocation; this allows the attacker to guess the correct client-port of the connection that the puppet establishes. However, many operating systems try to avoid predictable port allocation, as recommended in RFC 6056 [21]. In this paper, we successfully attack the *Simple Hash-Based Port Selection* (SHPS) algorithm, recommended in [21] and implemented in Linux. SHPS applies to many clients, e.g., running Android or connect to the Internet through NATs, which often run Linux (which uses SHPS). Our technique, described in Section 3, is based on an observation from the TCP specification, i.e., is independent of the platform (cf. to [13, 20]).

2.2 Learn Sequence Numbers

The next step after identifying the victim-connection is learning one or both connection's sequence numbers (step 3 in Figure 1); knowledge of the server's (client's) sequence number allows her to inject data to the connection, impersonating as the server (client). Observing the sequence numbers directly from traffic requires an on-path attacker (i.e., eavesdropping capability). Off-path TCP injection techniques use different methods to infer the sequence numbers.

2.2.1 *Operating-System Specific*

In the attacks of [13, 20], the adversary exploits the global counter IP-ID implementation in Windows. The attacker observes the difference in the IP-ID field in packets that she receives from the client to learn the number of packets that the client had sent to other destinations (since each packet increments the IP-ID).

In these attacks, 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 and the attacker learns whether the client responded by observing the IP-ID field (in packets that she receives from the client in a different connection).

After learning the server's sequence number, the techniques in [13, 20] exploit Windows TCP implementation, which filters incoming packets according to their acknowledgment numbers (this mechanism is non-standard). This implementation allows the attacker to learn which acknowledgment number is valid (passes filtering) by again observing the IP-ID side channel; the valid acknowledgment number equals the client's sequence number.

	Learn Connection 4-tuple	Learn Sequence Numbers	Exploit
Lkm [20]	Active probing for connection (Windows client, no firewall)	Exploit global IP-ID counter impl., both seq. # obtained (Windows client)	None
Qian et al.[26, 27]	Monitor connections, e.g., with netstat (Malware)	Read client system counters, server's seq. # obtained (Malware; in [26] also seq. # checking firewall)	XSS, CSRF, phishing (no TLS/SSL)
Gilad and Herzberg [13]	Establish connection, exploit sequential port allocation impl. (Puppet, Windows client)	Exploit global IP-ID counter impl., both seq. # obtained (Puppet, Windows client)	XSS, CSRF, phishing (No TLS/SSL)
This work	Establish connection, client port de-randomization (Puppet, client behind firewall)	Exploit browser behavior, server's seq. # obtained (Puppet, no TLS/SSL)	As above plus web-cache poisoning (No TLS/SSL)

Table 1: Off-Path TCP Injection Attacks: Building Blocks. In brackets: requirements.

2.2.2 *Sequence Number Inference Attacks*

In the sequence number inference attacks [26, 27] the attacker sends spoofed packets to the client machine. Each packet specifies a different sequence number. The observation in [26] 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 [27] 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 ([26]) or responded to it ([27]).

2.2.3 *Inject and Observe*

We use a different approach than the previous attacks; our technique, called '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 [13, 20]). Additionally, since we assume only a puppet running on the client machine, Mallory cannot receive feedback from system files (cf. to [26, 27]).

In the Inject and Observe technique, described in Section 4, Mallory sends to the client data which is spoofed as coming from the server in response to queries that the puppet sends; this phase relies on a very common browser behavior that 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.

2.3 TCP Injection: Exploits

The final building block of the attack is an application of the injection (steps 4, 5 in Figure 1), typically to inject a malicious object into the connection. The malicious object may be cached, and the attacker can easily make sure it stays in cache (theoretically forever); we refer to such a malicious, long-lived object or script as a *persistent XSS*.

Web-cache poisoning with a persistent XSS allows the attacker long-term use of many exploits, including cross-site scripting, cross-site request forgery and phishing (suggested in previous works [13, 26, 27]), bypassing the state of the art defenses such as CSP. See Section 5.

3. CLIENT PORT DE-RANDOMIZATION

The first step in performing a TCP injection is to identify the victim-connection. As described in Section 2.1, Mallory uses the puppet to establish the victim-connection; therefore, she knows the client's address as well as the server's address and port[2]. In this section we describe a new technique that allows Mallory to learn the fourth parameter of the TCP four tuple: the client port.

In Windows, learning of the client port is trivial, since port numbers are assigned consecutively (for all destinations). However, it is widely accepted that this is insecure, and that the client port should be 'unpredictable' to an off-path attacker. RFC 6056 [21] presents five recommended client port selection algorithms to secure against off-path adversaries. We focus on their third suggestion: 'Simple Hash-Based Port Selection' (SHPS).

SHPS is used by the Linux OS kernel in versions 2.6.15 and above, i.e., from the year 2006; it is 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, we show a method allowing an off-path attacker (Mallory) to predict the next port assignment by SHPS. We begin, in Section 3.1, with *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 Section 3.2, we present a *meet in the middle* optimization method, which applies elimination and testing *concurrently* to multiple ports, improving the efficiency of the prediction technique. We complete this section with Subsection 3.3, which discusses practical challenges and presents an empirical evaluation.

3.1 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 from the client's IP address and port p, 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,

[2]Our initial discussion assumes that the server has one IP address; in practice, large servers often have multiple addresses, we refer to this issue in Subsection 3.3.

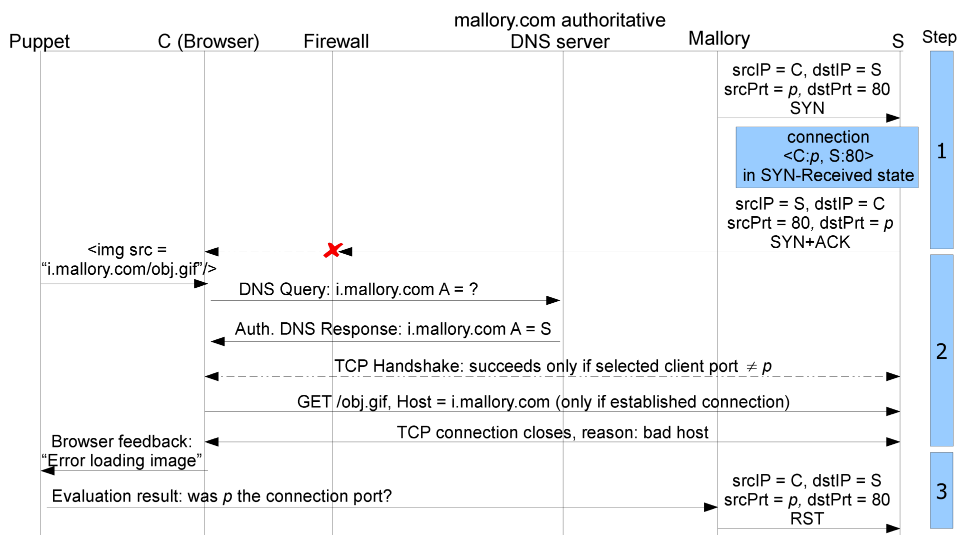

Puppet
C (Browser)
Firewall
mallory.com authoritative DNS server
Mallory
S
Step
srcIP = C, dstIP = S
srcPrt = p, dstPrt = 80
SYN
connection <C:p, S:80> in SYN-Received state
1
srcIP = S, dstIP = C
srcPrt = 80, dstPrt = p
SYN+ACK
<img src = "i.mallory.com/obj.gif"/>
DNS Query: i.mallory.com A = ?
Auth. DNS Response: i.mallory.com A = S
TCP Handshake: succeeds only if selected client port ≠ p
2
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

Figure 2: Port De-Randomization, Elimination and Testing.

the puppet tries to establish a new connection with S; the *response time* gives an indication if the port was eliminated or not. We now provide the details to our technique, illustrated in Figure 2.

In the first step (see Figure 2), 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 `<C:p,S:server-port>` and assigns it the SYN-Received state. S then sends a SYN+ACK response to C; a stateful 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); 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 2), 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 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 2), the puppet evaluates whether p was the connection port and informs Mallory. Evaluation is easy, 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. 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 [25] page 69; the TCP connection will not be established. The client operating system will retry to establish the connection several times and return an answer to the puppet only after several seconds. Often, this answer 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, port $p+1$ will be used. Otherwise, the attacker can repeat the process, until eventually successful. This would work - but not efficiently, requiring approximately 2^{15} iterations until success (since the port field is 16 bits long).

3.2 A Meet-in-Middle Optimization

In this subsection we present a *meet-in-middle optimization*, that reduces dramatically the time and communication involved in the port 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 significantly smaller than 2^{16}, see next subsection). In order to improve de-randomization run-time, Mallory uses the puppet to establish multiple connections to the server and eliminate ports simultaneously.

In the first phase of the de-randomization process, Mallory performs port-elimination (described above) on $\sqrt{\pi}$ ports,

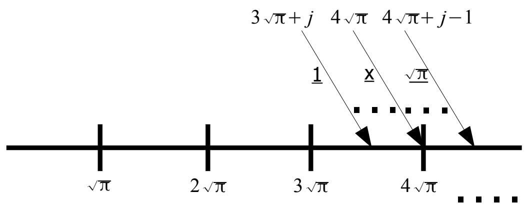


Figure 3: Port De-Randomization, Meet-in-the-Middle Optimization. At the top are ports allocated by the operating system, illustrated by the arrows; numbers with underscore mark the connection number. At the bottom are ports that Mallory eliminates.

specifically, the ports $\{\lceil i\sqrt{\pi}\rceil\}_{i=0}^{\sqrt{\pi}-1}$. In this phase, the puppet establishes $\sqrt{\pi}$ connections to S, which we number by the order of establishment. See illustration in Figure 3.

In order to use the puppet to establish multiple connections to 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 manipulating DNS mapping of attacker controlled domains: the puppet requests objects from $\{i.\texttt{mallory.com}\}_{i=0}^{\sqrt{\pi}-1}$; Mallory controls the DNS records for these domains and maps them to the IP address of S. Browsers use domain-names to identify servers and not IP addresses; hence, this technique, which we verified on Chrome and Firefox, opens $\sqrt{\pi}$ new connections to S.

We assume that the user does not create an independent connection with S during port de-randomization. According to the SHPS algorithm, client port allocation is sequential; therefore, 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 $\sqrt{\pi}-x$ connections after connection x, the current value of the operating system's client-port counter is $k\sqrt{\pi}-x$ for some $0 \leq k < \sqrt{\pi}$. 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 of the remaining $\sqrt{\pi}$ 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 $\lceil \log_2 \sqrt{\pi}\rceil$ iterations to complete.

3.2.1 *Analysis*

The maximal number of *simultaneous* connections that the puppet may open changes according the version of the browser; this value is at least 15 in all modern browsers and typically increases with new releases. Based on this, we estimate the amount of transmitted data and time required to perform port de-randomization.

Mallory sends $\sqrt{\pi} \leq \sqrt{2^{16}} = 256$ 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 at most 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 at most 8 iterations which perform one after the other, i.e., requiring 16 RTTs, i.e., typically about 1.5-3 seconds.

3.3 Real-World Challenges and Evaluation

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

3.3.1 *Challenges*

A. Multiple Server IP Addresses. Large web-sites often map their domains 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 port 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 3.1). If the puppet receives a feedback after a relatively long delay, then Mallory de-randomized the port counter for the correct IP address; otherwise, Mallory performs the de-randomization process again for another IP address.

B. SYN Flooding. Our port de-randomization technique requires sending $\sqrt{\pi}$ 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 [9], i.e., an attempt to clog the server's connections backlog; we now discuss the defenses suggested in [9] that might be triggered and influence our technique.

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 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, 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. 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 specify only one of four options for maximal segment size (MSS), which may degrade service for some clients. Furthermore, SYN-cookies reduce the entropy in the server's sequence number which may allow an attacker to guess its value, see [16].

Finally, the server may reduce its TCP timers; this will release server 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 these defenses have disadvantages which may discourage servers from deployment. A typical solution, suggested in [9] and [22], is a hybrid approach: the server keeps a small state for each connection, e.g., using SYN cache [9], 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 spoofed SYNs; this allowed us to de-randomize the client port with high success rates (see next).

3.3.2 *Evaluation*

Setup. We evaluated our technique on connections with popular web-sites, specifically, the top 1024 sites in the Alexa ranking [2]. 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 search run time.

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.

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 4 shows the failure rates as a function of web-site popularity. Port de-randomization failed for approximately 7% of the 1024 websites that we tested (see Figure 4 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 4 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 it 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 SYN); these 'stateless' servers were approximately 1% of the servers (see Figure 4 line 3). Port de-randomization for other servers (approximately 3%) failed due to other errors; e.g., sometimes we were not able to retreive the server's IP address (probably due to DNS filtering at the network or ISP level).

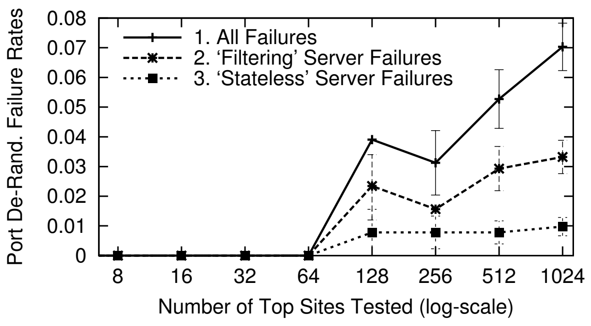


Figure 4: Port de-randomization *failure rates*, as a function of web-site popularity. Rates are the average of two runs: one when puppet runs on Firefox and the other on Chrome. Error-bars mark standard deviations.

4. LEARNING THE SEQUENCE NUMBER

We now proceed to the second building block (and attack phase), as in the design presented in Section 2.2.3. At the end of this phase Mallory learns the 32-bit server's sequence number; this allows her to send data to the client, impersonating as the server. We assume that Mallory has the parameters of the victim-connection, in particular, that she identified the client port; e.g., by executing the technique described in Section 3 (or other methods, see Table 1).

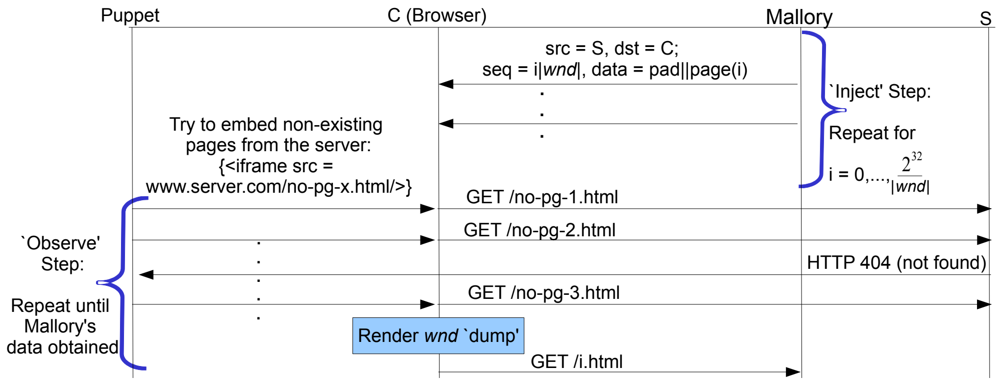
Our attack exploits an *under-specification of HTTP 1.1* [12]. Subsection 4.1 provides required background, explaining how browsers handle HTTP responses that they receive. Subsection 4.2 describes our search technique. Subsection 4.3 presents the requirements of our search technique and presents an empirical evaluation in the real-world.

4.1 HTTP Request/Response Handling

As of HTTP 1.1 [12], 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 receive-buffer (allocated per-connection) when they become available. The browser expects to find the matching response in the beginning of TCP's receive-buffer. Next, the browser parses the response as per [12], 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 [12] does not specify what the browser should do when the receive-buffer 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



Puppet
C (Browser)
Mallory
S
src = S, dst = C;
seq = i|wnd|, data = pad||page(i)
`Inject' Step:
Repeat for
i = 0,...,$\frac{2^{32}}{|wnd|}$
Try to embed non-existing pages from the server: {<iframe src = www.server.com/no-pg-x.html/>}
`Observe' Step:
Repeat until Mallory's data obtained
GET /no-pg-1.html
GET /no-pg-2.html
HTTP 404 (not found)
GET /no-pg-3.html
Render wnd `dump'
GET /i.html

Figure 5: Server Sequence Number Learning Technique.

the receive-buffer 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[3]. The following subsection explains how we exploit this behavior to learn the server's sequence number.

4.2 Inject and Observe

In this subsection we present the server sequence number learning technique which is illustrated in Figure 5. The technique has two steps: *(1) Inject* and *(2) Observe.*

(1) 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 receive-buffer for the connection and $|wnd|$ denote its size. In order to inject the data, Mallory sends to the browser $\frac{2^{32}}{|wnd|}$ packets, 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 $pad||page(i)$, where pad is an easily-removable 'pad'[4] and $page(i)$ is a simple web-page defined as follows:

```
<HTML><BODY>
<iframe src = "www.mallory.com/i.html" />
</BODY></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 [25] page 72. 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. Hence, exactly one of the packets will contain 'good' sequence and acknowledgment numbers, and its data is saved in the receive-buffer (wnd). Namely, after this step, C's victim-connection wnd is as illustrated in Figure 6.

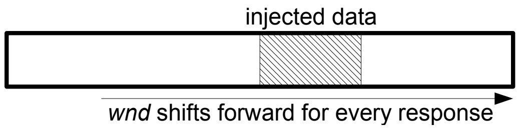


Figure 6: The state of *wnd* after 'inject' step.

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 (these requests were removed for readability from Figure 5). The reason that one request must always be enqueued is that when there are no pending requests, some browsers clear the receive-buffer (those will discard the injected data).

(2) 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 5). Each response that arrives at C shifts wnd forward; once a sufficient number of responses arrived, such that there is no gap of unreceived bytes between the injected data (buffered in wnd) and the last response, then the browser will also read the injected response, expecting it to be the following response; see illustration in Figure 7. In fact, the last response would (usually) overwrite part of the pad at the beginning of the injected data; the pad is at least as long as the server's response[5], hence, some of the pad and all of $page(i)$ would remain and be read and *rendered* by the browser. As explained in the previous subsection, in all browsers that we

[3]This behavior may have been adopted to simplify 'debugging' of servers that implement HTTP pipelining incorrectly.
[4]The length of the pad and its use will become clear when we present the following 'observe' step.

[5]The pad may, for example, be of the form $\{0\}^m||1$, where m is the length of the longest possible response.

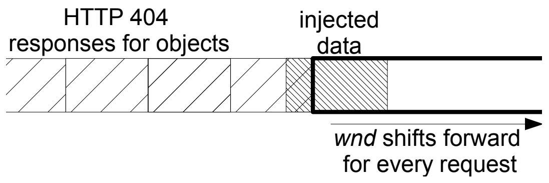


Figure 7: The state of *wnd* during the 'observe' step.

tested, the remaining injected data is handled as a regular response with a default header, and returned to the puppet.

The requests that the puppet sends are for arbitrary webpages that will yield short responses, e.g., HTTP 404 responses (page not found). When the browser renders the injected response, it will try to retrieve the page *i.html* from Mallory's web-site (see Figure 5); providing to Mallory the value of i. This allows Mallory to compute the next server sequence number that C expects.

4.2.1 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 $|wnd|$ of 2^{16}, we find that the Mallory sends 2^{17} packets. The number of additional packets sent during 'observe' step is negligible. Each packet is of maximal response size to the puppet queries, which are usually short; assume that it is 800B; in this case, Mallory sends approximately 100MB of data during Inject and Observe.

4.3 Requirements and Real-World Evaluation

In this subsection we present the requirements of the Inject and Observe technique and evaluate its success rate on connections with popular web-sites.

4.3.1 Requirements

Our Inject and Observe technique has two requirements from the web-server side:

1. Support persistent HTTP connections and request pipelining (default in HTTP 1.1 [12]). This allows the puppet to send several requests over the same connection; if the server does not support these properties, then the connection will close after the first response arrives.
2. 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).

In Figure 8 we evaluate the applicability of the Inject and Observe technique on connections with the top 128 websites in Alexa popularity rank; we observe that 73% of these web-sites are potentially vulnerable (see Figure 8, line 1).

4.3.2 Evaluation

We verified that the browser behavior which we exploit exists in Chrome (v23), Firefox (v16) and Internet Explorer (v9). We empirically evaluated the Inject and Observe technique with Chrome and Firefox (IE does not run on our Linux client machine). We measured the success rate in two scenarios: (1) as a standalone component, where the client-port is obtained by executing netstat on the client machine; (2) as part of a complete injection attack, together with the client-port de-randomization technique presented in Section 3. We identified a successful execution when Mallory receives a GET request for *i.html* from the client (where i is an integer).

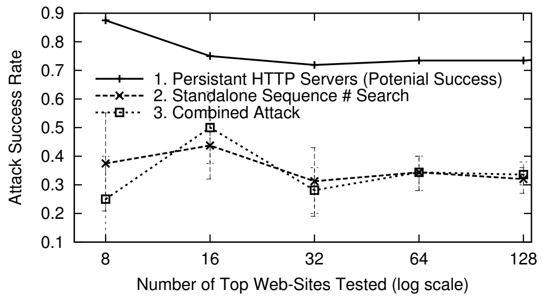


Figure 8: Inject and Observe, Evaluation. Potential and measured success rates as a function of web-site popularity. Error-bars mark standard deviations.

Setup. Our setup is as in Section 3.3.2.

Results. Figure 8 compares the success rates (average of measurements in Chrome and Firefox) for both scenarios. The indicated success rate of our attack as a standalone component is approximately 35% (see Figure 8, line 2). This rate is significant, but lower than what we expected, and is roughly half of the potential success rate (Figure 8, line 1); observing the logs, we found that some servers responded to the puppet's requests (during the 'observe' step) with long HTTP 404 responses. These responses were longer than our padding (800 bytes), which caused an error and failed the search. In practice, inspection of the web-server to identify short objects that the puppet could request will increase the success rate. The combined attack (see Figure 8, line 3) has a similar success rate to that of standalone Inject and Observe; this is since our port de-randomization technique has high probability for success (see evaluation in Section 3.3.2).

The average run-time of a successful Inject and Observe was approximately 146 seconds (standard deviation 21 seconds), the average run-time of the complete attack was approximately 180 seconds (standard deviation 28 seconds).

5. EXPLOITS

In Sections 3 and 4 we showed how Mallory can learn the client port and server's sequence number for a 'victim' connection that the puppet establishes; these parameters allow Mallory to inject data to the connection, impersonating as the server. We, as well as Qian et al., extensively discussed the XSS, CSRF and phishing exploits in [13, 26, 27] which are briefly reviewed in this section. This section focuses on the web-cache poisoning exploit that was not considered in prior works.

5.1 XSS, CSRF and Phishing

Cross-Site Scripting (XSS). The puppet requests an HTML page from the server over the victim-connection, e.g., by including `<iframe src="www.server.com/page.html"/>`. Mallory provides a spoofed HTTP response with an HTML page that contains a JavaScript. This script then executes in context of `www.server.com`; i.e., Mallory circumvents the same origin policy. Since Mallory writes the HTTP header of the response, she can bypass state of the art XSS defenses such as CSP [30]. This attack vector is illustrated in Figure 1.

Cross-Site Request Forgery (CSRF). Once attackers succeed in an XSS attack, i.e., run a malicious script in context of a victim site, they can exploit it in many ways. In particular, an XSS attack allows attackers to send a forged request to the server on the user's behalf, circumventing all known defenses against CSRF attacks for non-secured connections, except for (few) defenses requiring extra user efforts for submission of each sensitive request; see [24].

Phishing. Mallory opens the victim-connection to some website and learns the connection's parameters. Mallory waits until the user enters to the same website, and since browsers re-use connections, the browser will send the user's requests over the victim-connection, allowing Mallory to inject a spoofed web-page in response. In particular, Mallory can learn user credentials by spoofing the homepage of websites, which often do not invoke SSL until the user presses a log-on link; this is done by injecting a spoofed homepage where the log-on link points to Mallory's site.

5.2 Web-Cache Poisoning and Persistent XSS

The exploits above are limited: they can only run at the present moment and in the current victim-connection between C and S. This motivates a long-lasting *web-cache poisoning* attack [19, 31]. Mallory can cache spoofed responses (for requests made by the puppet) at the browser, as well as possible intermediate network proxies that will provide the spoofed page to other users. For example, the following HTTP headers cache the spoofed response for a day, and impede the browsers from refreshing the page:

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

Mallory can poison the web-cache with spoofed pages that users will receive when they access the (poisoned) websites. In particular, this allows Mallory to cache spoofed login pages, i.e., a *persistent phishing* exploit. Furthermore, Mallory can cache a malicious script in context of some target website, and run it automatically (without further TCP injections) when a user in the network enters the target website. Similarly to the XSS exploit shown above, the cached script executes in context of victim website; i.e., a *persistent XSS* exploit.

6. DEFENSES

In Sections 3 and 4 we showed how an off-path attacker can learn the client port and server sequence number of a TCP connection, allowing the exploits in Section 5. This section presents client and server end defenses for the attack vectors considered in this paper.

6.1 Client-End Defenses

6.1.1 Client-Port De-Randomization

Client-side operating systems should stop using the popular Simple Hash-Based Port Selection (SHPS) port selection algorithm, attacked in Section 3, and adopt a secure alternative. RFC 6056 [21] presents SHPS, together with four other algorithms, which are therefore good candidates. The security of the port selection algorithm should be analyzed considering TCP mechanisms that might leak the state of connections.

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.

6.1.2 Exposure of Server Sequence Number

The Inject and Observe technique that we presented for exposing the server's sequence number exploits a de facto browser behavior standard, which is not required by the HTTP specification: process and display 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.

6.2 Server-End Defense

The Inject and Observe technique that we introduced for learning the server's sequence number injects data to the TCP stream; injected data is observed by the puppet who provides a corresponding feedback to the attacker. 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.

7. CONCLUSIONS AND FUTURE WORK

We presented a new technique to perform off-path TCP injections and evaluated its effect 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.

This work continues a line of recent works on TCP injections [13, 20, 26, 27], 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 [21] is required to identify the best alternative for the extensively deployed SHPS algorithm. Second, it may also be possible to learn the client sequence number, e.g., as in [13]; this will allow data injection to the server-side, which may allow new exploits.

8. ACKNOWLEDGMENTS

We thank the anonymous referees for their comments and suggestions. This research was supported by the Ministry of Science and Technology, Israel; by grant 1354/11 from the Israeli Science Foundation (ISF); and by the Check Point Institute for Information Security.

9. REFERENCES

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

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

[3] S. Antonatos, P. Akritidis, V. T. Lam, and K. G. Anagnostakis. Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure. *ACM Transactions on Information and System Security*, 12(2):12:1–12:15, Dec. 2008.

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

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

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

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

[8] R. Beverly, A. Berger, Y. Hyun, and K. C. Claffy. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In A. Feldmann and L. Mathy, editors, *Internet Measurement Conference*, pages 356–369. ACM, 2009.

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

[10] T. Ehrenkranz and J. Li. On the State of IP Spoofing Defense. *ACM Transactions on Internet Technology (TOIT)*, 9(2):6:1–6:29, 2009.

[11] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which Employ IP Source Address Spoofing. RFC 2827, May 2000.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[13] Y. Gilad and A. Herzberg. Off-Path Attacking the Web. In *USENIX Workshop on Offensive Technologies*, pages 41 – 52, 2012.

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

[15] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web*, pages 601–610. ACM, 2007.

[16] D. Kaminsky. Black Ops of TCP/IP. In *Black Hat conference*, Aug. 2011. `http://dankaminsky.com/2011/08/05/bo2k11`.

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

[18] A. Klein. Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. *White Paper*, 2004.

[19] A. Klein. Web Cache Poisoning Attacks. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 1373–1373. 2011.

[20] klm. Remote Blind TCP/IP Spoofing. Phrack magazine, 2007.

[21] M. Larsen and F. Gont. Recommendations for Transport-Protocol Port Randomization. RFC 6056 (Best Current Practice), Jan. 2011.

[22] J. Lemon. Resisting SYN Flood DoS Attacks with a SYN Cache. In S. J. Leffler, editor, *BSDCon*, pages 89–97. USENIX, 2002.

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

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

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

[26] Z. Qian and Z. M. Mao. Off-Path TCP Sequence Number Inference Attack. In *IEEE Symposium on Security and Privacy*, pages 347–361, 2012.

[27] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative TCP Sequence Number Inference Attack: How to Crack Sequence Number Under a Second. In *Proceedings of ACM Conference on Computer and Communications Security*, CCS '12, pages 593–604, New York, NY, USA, 2012. ACM.

[28] J. Ruderman. Same Origin Policy for JavaScript. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`, 2001.

[29] T. Shimomura and J. Markoff. *Takedown: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaw - by the Man Who Did It.* Hyperion Press, 1st edition, 1995.

[30] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web*, pages 921–930. ACM, 2010.

[31] The Open Web Application Security Project. Cache Poisoning. `www.owasp.org/index.php/Cache_Poisoning`, 2009.

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

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

[34] P. Watson. Slipping in the Window: TCP Reset Attacks. Presented at CanSecWest, 2004.

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

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