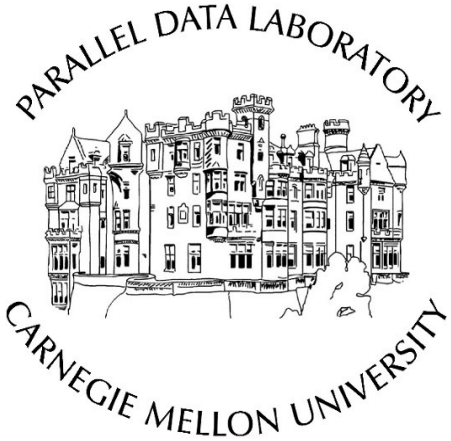


PETAL: Preset Encoding Table Information Leakage

Jiaqi Tan, Jayvardhan Nahata

Contact: tanjiaqi@cmu.edu

CMU-PDL-13-106
April 2013

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

SPDY is an application-layer protocol which multiplexes multiple HTTP requests and compresses HTTP headers over a single TCP connection protected by SSL/TLS encryption. Web applications are ubiquitous, and HTTP headers carry HTTP cookies which often contain sensitive information which can result in loss of privacy if leaked. We perform a security analysis on the proposed compression scheme for the next revision of the SPDY protocol, particularly with respect to the previously disclosed CRIME attack which uses compression-based information leaks. We have identified a new information leakage in the compression scheme of the proposed and previous versions of the SPDY protocol, which we call PETAL[1], which exploits the use of a fixed Huffman encoding table and the lack of byte-alignment of encoded characters, and we have identified a way to recover cookies using this information leakage by exploiting the way that multiple HTTP cookies with the same name but different `Path` *attributes are handled by current web browsers. We perform a detailed analysis of the impact of this information leakage, and find that after considering practical issues such as the byte-padded nature of network communications, our hypothesized attack only leaks less than 2-bits of information for 30-character uppercase alphanumeric strings, and does not allow a network attacker to recover meaningful amounts of information despite our discovered information leakage.*

Acknowledgements: We would like to thank Collin Jackson and Lin-Shung Huang for helpful discussions and suggestions for this paper. We would also like to thank Roberto Peon and Adam Langley for feedback and helpful discussions on this paper.

Keywords: SPDY, Information Leak, HTTPS, SSL, Cookies

1 Introduction

SPDY is an application-layer protocol proposed by Google [12] for transporting web content. The goal of SPDY is to reduce the latency of loading web pages by reducing the number of TCP connections used to avoid the overheads of TCP Slow Start, and by compressing HTTP headers. SPDY multiplexes HTTP requests over a single TCP connection, keeping a persistent TCP connection between the web browser and the web server. By default, SPDY also requires the transport layer to use Secure Sockets Layer/Transport Layer Security (SSL/TLS) for encryption.

SPDY presents a number of opportunities for malicious attackers. First, SPDY maintains a single long-lived (relative to setting up one connection per resource loaded) connection and hence a single context for the connection, creating an opportunity for attackers to try to learn about this context and exploit information learned. Second, SPDY mandates the use of SSL/TLS encrypted transport, presenting opportunities for interaction between SPDY and SSL/TLS which may give rise to emergent weaknesses due to interactions between the two protocols. In particular, as SPDY supports the compression of HTTP requests and responses transported over SPDY, and SPDY mandates the use of encrypted SSL/TLS transport, this creates the opportunity for information leakage via the length of the transmitted data due to the data being compressed prior to encryption. Third, a number of SPDY features for consolidation, such as IP pooling (shared connection for different domains with the same IP address), which were designed with efficiency as the main goal, may also give rise to emergent behaviors that attackers may exploit.

Currently, the latest version of SPDY deployed in web browsers and servers is SPDY3, while some older versions of web browsers and servers continue to support SPDY2. SPDY4 is the latest draft of the SPDY protocol and is under current active development.

In this paper, we perform a detailed analysis of the security of SPDY, and evaluate whether the design of SPDY gives rise to vulnerabilities in HTTP and in SSL/TLS that an attacker can exploit to compromise a web connection to learn secrets about a user's connection, or impersonate or spoof other principals in the connection. Specifically, we analyzed the interactions between compression in SPDY and encryption in SSL to identify potential information leakage due to interactions between the two.

In particular, we identified an information leakage vulnerability in the proposed SPDY4 compression scheme due to its use of a fixed Huffman coding table, and we call this vulnerability PETAL (Preset Encoding Table Information-Leaks). When combined with a method we identified for injecting multiple cookies with the same name, it is possible for a network attacker to inject guesses for cookie values in a SPDY4 connection to mount a chosen plaintext attack. However, after considering practicalities such as the byte-padding of networked communications, we find that this attack leaks less than 2-bits of information in a 30-character string, rendering the attack impractical for network attackers.

1.1 Contributions

We summarize our contributions as follows:

- We systematize the known aspects of the CRIME information leakage and partial chosen plaintext attack against compression used in TLS [11], and show how this applies to the compression used in SPDY2 and SPDY3. (§2.3.3)
- We describe the new SPDY4 compression scheme, and how it successfully mitigates against CRIME-like attacks using partial chosen plaintext attacks. (§2.4)
- We demonstrate how to exploit the behavior of cookie handling when there are multiple cookies with the same name, but different values for the `Path` attribute, to inject guesses to mount a chosen plaintext attack against SPDY4. This enables the recovery of the plaintext contents of a cookie. (§3.1.2)

- We describe a new information leakage vulnerability caused by the use of a fixed Huffman coding table in SPDY4, and we quantify the amount of information that can be leaked due to this vulnerability. We also present a hypothesized attack making use of this vulnerability. (§3.2)
- We found that with byte-padding, the information leaked due to the use of a fixed Huffman coding table in SPDY4 is negligible. Less than 2-bits of information is leaked for a 30-character string, rendering our attack impractical for network attackers. (§4.3)

1.2 Organization

First, we describe our assumed threat model. Second, we give a detailed description of the features and architecture of SPDY. Second, we discuss vulnerabilities arising from the interaction of compression and encryption, and we describe in detail the CRIME attack against the DEFLATE compression used in TLS, which used the length of encrypted packets to infer encrypted secrets. Third, we describe a framework for attacking SPDY by leveraging its use of compression over an encrypted transport channel. Fourth, we describe the newly proposed mitigation measures in SPDY4 and residual vulnerabilities. Then, we discuss other features of SPDY and potential weaknesses. Finally, we conclude and discuss future work.

2 Background

2.1 Threat Model

The SPDY protocol manages requests below the level of abstraction of the web browser by multiplexing multiple HTTP requests over a single TCP (SSL/TLS) connection. Hence, we consider web attackers as being too weak to affect the behavior of SPDY. Instead, we consider the active network attacker, who is able to intercept, inspect, modify, and spoof packets on the network. However, we consider the cryptographic state of the underlying SSL/TLS connection to be secure: we assume that the attacker does not have access to the encryption keys nor the certificate of the victim server, and that the attacker is unable to read or modify encrypted content. However, due to the format of SSL/TLS data frames, the length of each frame is sent in the clear in the header of the TLS data frame, and the attacker is able to inspect the lengths of encrypted packets. In addition, we do not need to assume that the user will ignore SSL/TLS certificate warnings, and both the existing CRIME attack against SSL/TLS compression and our hypothesized PETAL attack do not generate SSL/TLS certificate warnings.

2.2 SPDY

SPDY [12] is a web transport protocol proposed by Google, and has been implemented in a number of major web browsers and web servers. SPDY has four technical goals: (i) Allow multiple concurrent HTTP requests to use a single TCP connection. (ii) Reduce HTTP bandwidth usage through header compression. (iii) Make SSL the underlying transport protocol for end-to-end security. (iv) Enable web servers to initiate communications with the client to allow data push.

SPDY has four key features [12]: (i) Multiplexed requests: An unlimited number of requests can be issued concurrently over a single SPDY connection. (ii) Prioritized requests: Clients can specify a priority for requests so that higher priority requests are delivered first. (iii) Compression of HTTP headers. (iv) Server-pushed streams: Server Push allows servers to push content to clients without a request. SPDY operates at the Session layer in the OSI model, and is transparent to HTTP, SSL/TLS, and TCP.

When a SPDY client makes a request to a given web page, the client first checks its pool of SPDY sessions to look for an existing connection to the web server, and it creates a session if it doesn't already

exist. The client then sends its request over the SPDY session (either an existing one from a previous request, or the newly created one), and receives its response. For subsequent requests to the same server and their responses, the web client continues to use this SPDY session, saving the latency overhead incurred due to TCP Slow Start.

2.2.1 Sessions and Streams

SPDY sessions are initiated by web clients. In the default configuration, all SPDY sessions must be tunneled over an TLS/SSL connection. The web client first creates an SSL/TLS session with the web server, before opening a SPDY session. Then, a web client opens one SPDY session for a given web server, and each HTTP request and response is sent over a SPDY stream. A SPDY session refers to the persistent connection between a web client and web server for the duration of a logical transaction during which the client may request multiple resources for a given web page, or even for subsequent page requests during the course of the client's web access. A SPDY stream refers to a single request/response. A SPDY session supports the sending and receiving of multiple SPDY streams between the client and server.

2.2.2 SPDY Stream Lifecycle

Each SPDY session (connection) multiplexes multiple SPDY streams, and one SPDY stream is created for each request and its corresponding response. A stream is created by sending a SYN_STREAM control frame. stream IDs must increase monotonically as new streams are created. If the client initiates the connection, the stream ID must be odd, and if the server initiates, the stream ID must be even. 0 is not a valid stream ID. On receiving a request on a stream, the recipient responds on the same stream. The receiving end point of the connection must then send a SYN_REPLY in response to the SYN_STREAM to acknowledge the creation of the stream before data can be sent and received. Sending two SYN_STREAMs with the same steam ID is a protocol error, and the recipient must send a stream error using RST_STREAM.

2.3 SPDY2/3 Compression: Design and Information Leakage Vulnerabilities

SPDY2 and SPDY3 both implemented compression for the name/value header block sent in the SPDY SYN_STREAM and HEADER control frames. This header block stores the HTTP headers used in the HTTP requests and responses transported over SPDY's streams. These headers include high-sensitivity content, such as HTTP cookies, which can contain user authentication tokens stored by web applications which can be used to impersonate the user in web application sessions the user is authenticated to.

The original SPDY2 and SPDY3 specifications called for the name/value header blocks to be compressed. Prior to the disclosure of the CRIME attack, the specification called for the header block to be compressed using ZLIB DEFLATE [7, 6]. The DEFLATE compression uses both Huffman encoding and the LZ77 compression algorithm, although the use of the Huffman encoding in the DEFLATE implementation in SPDY2/3 does not rely on a fixed encoding table.

While web clients and servers which have responded to the CRIME attack have already disabled their use of TLS-DEFLATE compression, SPDY2 and SPDY3 will still be vulnerable to the CRIME attack even without TLS-DEFLATE, because SPDY carries out its own compression of the name/value header block, and all name/value headers are compressed together in the same compression context.

2.3.1 ZLIB DEFLATE Compression

ZLIB [7] is a specification for a compressed data format, while DEFLATE [6] in particular specifies the use of the Huffman encoding [4] and LZ77 compression algorithm [13].

The Huffman encoding is a prefix-coding whose encoding of characters is given by the nodes of a binary tree. The tree has interior nodes labelled '0' or '1', and leaf nodes labelled with the characters to be encoded. Then, the encoding for a given character is given by the binary string obtained from a root-to-leaf traversal ending in that character. The construction of a Huffman encoding tree is designed to place characters with the highest occurrence nearer to the root of the tree so that they have shorter encodings, thus saving space. Huffman trees can be constructed on-demand based on the actual data bytes being compressed, or they can be constructed based on an empirical or expected frequency table when all the data may not be known in advance.

LZ77 is a lossless data compression algorithm which uses a sliding window during compression. It compresses data by replacing repeated occurrences of strings seen in the sliding window with a single copy. Figure 1 illustrates how LZ77's sliding window compression works. In the string "Google is so googled", the substring "oogle" is repeated and can be replaced by a simple encoding of an instruction to look back a certain number of characters, and the number of characters from the previous position to consume.

```
12345678901234567890
Google is so googled
Google is so g(-13,5)d
```

Figure 1: Example of LZ77 compression (Numbers above are to indicate character positions).

2.3.2 Information Leakage via Compression

The length of the output of compression algorithms, i.e. the compressed length of a given string of plaintext, reveals information about the plaintext. This is because the length of the compressed text depends on the contents of the plaintext itself. In general, for a given length of plaintext, shorter lengths of compressed text indicate greater redundancy within the plaintext, i.e. higher compression ratios (greater compression) indicate greater redundancy in the plaintext.

Specifically, this general principle of information leakage about the plaintext via compression can be turned into an attack on encryption if two additional requirements are met: (i) The encryption algorithm preserves the length of its cipher text, i.e. there is a 1:1 correspondence between the length of the plaintext and the length of the encrypted cipher text, and (ii) Partial chosen plaintext injection is feasible.

We assume that the goal of the attacker is to extract a given string which exists in a particular cipher text. Kelsey describes this as a "String Extraction" attack [8]. We also assume that data is compressed prior to encryption.

The first requirement will allow the attacker to learn the length of the compressed text from the length of the cipher text. The second requirement allows the attacker to make use of the partial plaintext injected as a guess for the presence of the injected partial plaintext in the cipher text.

Suppose there is a given plaintext P, which contains secret S such that $S \in P$, and that the attacker is able to inject a guess, G, to obtain some string $P+G$ where '+' is string concatenation. Then, suppose the attacker is able to obtain $strlen(encrypt(compress(P+G)))$ for any number of guesses G and fixed plaintext P containing secret S. Then, if the guess G is such that $G = S$, this redundancy will be reflected in the length of the encrypted cipher text, $strlen(encrypt(compress(P+G)))$, and of all the guesses, for the correct guess G^*, we have:

$$\underset{G^*}{argmin}\{strlen(encrypt(compress(P+G^*)))\} \Rightarrow G^* = S$$

i.e. the guess which has the best compression ratio is most likely to be the secret string in the plaintext. However, this requires that the longest match for guess G is the secret S rather than another string in the

target plaintext.

Hence, the general scheme of the attack is as follows:

1. For each guess, concatenate guess with plaintext, and inject guess into plaintext to obtain $P+G$.
2. Obtain the length of the encrypted, compressed cipher text, $strlen(encrypt(compress(P+G)))$
3. Repeat the above two steps for various guesses. If the guesses cover the entire space of all possible characters in the secret, then the guess which yields the minimum encrypted-compressed-length (i.e. has the highest compression ratio) is the secret in the plaintext P.

However, an additional feature is required to make this attack feasible. The partial chosen plaintext attack described above provides feedback not just for entire guesses of the secret, but also for partial guesses of the secret. Hence, for a secret of known length, the above algorithm can be iteratively executed as follows:

1. Carry out the above algorithm for to guess the first character, G_1.
2. For the character determined to be the correct guess, g_1^*, prepend the character to the guess in guessing the next character, i.e. in the second round, the guesses are constructed as $G_2 = g_1^* + g_2$ where g_2 is the new guess.
3. Repeat the above two steps for each subsequent character in the secret that is being guessed, iteratively concatenating the correctly guessed characters to the current guess.

As the partial chosen plaintext injection attack described above provides partial feedback for correct guesses of substrings of the secret, this greatly reduces the search space for the secret from an exponential order of magnitude of the character-set of possible characters, to a polynomial (linear) order of magnitude. Effectively, for $|\mathcal{L}| = n$ and $|S| = s$, if the attacker gains feedback only if the entire secret is correctly guessed, then the number of guesses needed is $O(n^s)$, but as the attack described above provides partial feedback, the number of guesses needed is reduced to $O(ns)$.

2.3.3 CRIME Attack Against TLS

The CRIME attack against TLS [11] exploits information leakage due to the DEFLATE compression used in TLS 1.0 to guess and recover plaintext in an encrypted SSL/TLS stream by observing the length of encrypted TLS data frames which are sent by attacker-crafted requests which contain guesses of the plaintext. The CRIME attack requires a network attacker for the following two steps:

1. Inject an IFrame to construct HTTP requests which contain guesses of the secret, and
2. Observe the length of the TLS data frames corresponding to the guess-containing requests.

The attacker does not require access to the encryption keys of the SSL/TLS connection, and there is no SSL/TLS session termination or any other action which would render an HTTPS warning in the victim's web browser.

At their Ekoparty 2012 presentation, Rizzo and Duong described how to extract a cookie string using the CRIME attack. First, they assume a network attacker who is able to inject JavaScript into the response stream from the server using a Man-in-the-Middle attack. This code can be injected into the response for any regular HTTP site, and the malicious code then makes requests to the targeted web server running on SSL/TLS.

They injected guesses by directly placing them in the HTTP request. They did this by adding the guess string in the query string of a URL sent to the victim website. For instance, for a victim site `http:`

//www.good.com/, the attacker places the guess in the query string to obtain a request URL: http://www.good.com/?guess_goes_here. Other possible ways of injecting guesses include requesting other resources such as images, and placing the guess in the query string of the requested URL so that the guess gets injected into the stream of bytes in the HTTP request. Then, the full HTTP request sent by the browser to the web server will contain the requested URL, as well as the secret in the cookie string, which will get compressed together by TLS DEFLATE prior to encryption. Then, as the length of TLS frames are prepended to the header of each frame in the clear, the network attacker can directly read this length off the wire to obtain feedback for his guess.

2.3.4 Main Vulnerabilities

Two main conditions enable the partial chosen plaintext attack against the TLS DEFLATE compression to be practical. First, TLS DEFLATE allows parts of the data compressed, such as the requested URL in the HTTP request, to be controlled by the attacker. This enables the attacker to insert guesses which are compressed together with the secret, allowing the attacker to gain feedback about his guesses by observing the length of the resulting cipher text. The key vulnerability here is caused by a shared *compression context* between attacker-controlled bytes and the secret data to be kept confidential, which gives the attacker an opportunity for choosing a partial plaintext (the guess of the secret) to inject. The *compression context* here refers to the bytes which are compressed together using the same compression sliding window in the LZ77 compression algorithm.

Second, the main feature of the information leakage which makes the compression attacks feasible is that the attacker is able to get feedback for correct guesses of substrings of the plaintext, i.e. the attacker gets partial feedback for guesses. This helps reduce the number of guesses from being exponential in the size of the character-set of the secret to being polynomial in the size of the character-set.

2.4 Mitigation: SPDY4 Compression

SPDY4 is the next proposed version of the SPDY protocol. At the time of writing of this paper, SPDY4 is currently still in draft status and has not been implemented. Nonetheless, in response to the vulnerability of header compression in SPDY2 and SPDY3 to the CRIME attack, a new compression scheme has been proposed [10]. Sample code demonstrating this new compression scheme has also been made publicly available[1].

The main idea in the design of the header block compression in SPDY4 is to isolate the compression contexts for each name/value pair in the block. This is to prevent attacker controlled bytes from being injected into the compression context of sensitive headers.

In particular, the header compression in SPDY4 has eliminated the use of the LZ77 compression. Instead, only the Huffman encoding is used. Similarly to SPDY2 and SPDY3, a fixed Huffman code based on the empirical frequency table in the SPDY protocol specification is used.

In place of the LZ77 compression, the SPDY4 compression introduces a new 'delta' compressor [3] based on a fixed dictionary of header names (or keys) and commonly occurring values, and an internal compression state consisting of an Least-Recently-Used-evicted (LRU) cache of previously seen key/value pairs. The delta compressor provides a language which contains 8 operators for operating on the LRU key/value store. Then, the output of the compressor is a stream of operations on the LRU key/value store to describe to the recipient how to decompress the stream.

The operations will be explained in the description of the compressor's behavior, and they are: **skvsto** (Stateful Key-Value Store), **ekvsto** (Ephemeral Key-Value Store), **sclone** (Stateful Key Clone), **eclone**

[1] https://github.com/grmocg/SPDY-Specification/tree/gh-pages/example_code

(Ephemeral Key Clone), **stoggl** (Stateful Toggle), **etoggl** (Ephemeral Toggle), **strang** (Stateful Toggle Range), **etrang** (Ephemeral Toggle Range).

Conceptually, the compression can be described as follows. First, headers are operated on in key/value pairs, with keys specifying the name of the header, e.g. "User-agent" or "Accept-language". Second, there are three possible cases for each key/value pair in the header block:

1. The key and value are both previously unseen.
2. The key is either previously seen, or matches a commonly-used key specified in the fixed dictionary, but the value is not previously seen.
3. The key and value are previously seen.

If the key and value are both previously unseen, the compressor can decide to either directly output the key/value pair without storing it to the compression state for keys and values it decides are rare (using the `ekvsto` command), or it can issue a command to insert the key and value into the LRU store and insert it into the stream (using the `skvsto` command).

If the key is previously seen or matches a commonly-used key specified in the dictionary, but the value is not matched, then the compressor can either store the value in the LRU index for the key (using the `sclon` command) while inserting it in the stream, or it can insert it in the stream without storing it in the LRU (using the `eclon` command).

If the key and value are both found in the LRU, then the key/value pair is "switched on", i.e. it is copied into the stream by the decompressor (using the `stoggl` command). The compressor can also disable previously inserted ephemeral clones or stores (using the `etoggl` command).

Then, the output of the SPDY4 compression is a series of commands to be interpreted by the decompressor, and this stream of commands contains data where required, e.g. when new keys and values are specified using the `ekvsto,skvsto` commands, or when new values are specified for existing keys using the `eclon,sclon` commands. Otherwise, for existing key/value pairs in the compressor state, `stoggl` commands are used to indicate that key/value pairs from the internal compression state are to be inserted into the data stream of uncompressed data to be returned to the user.

Hence, the SPDY4 compressor and decompressor state both begin from an initial dictionary as specified in the SPDY4 specification, and this state is then modified by the compressor-issued commands together with new keys/values to either insert directly into the data stream of uncompressed data to be returned to the user, or to insert into the internal LRU cache as well.

Treatment of Cookies In SPDY4, all crumbs in a cookie are treated as values belonging to the key "cookie". The cookie string is then tokenized by the ";" character, and each crumb, i.e. each part of the cookie string separated by ";"s are considered as one distinct value. For instance, if a request consists of a cookie with three crumbs: "`cookie1=cookievalue1`", "`cookie2=cookievalue2`", and "`cookie3=cookievalue3`", then the HTTP request stream will contain ('\' indicates a continuation of the same line):

```
Cookie: cookie1=cookievalue1 ; \
cookie2=cookievalue2 ; \
cookie3=cookievalue3
```

Then, the SPDY4 compressor will treat this string as:

```
cookie = cookie1=cookievalue1
cookie = cookie2=cookievalue2
cookie = cookie3=cookievalue3
```

2.5 Compression Contexts in SPDY4

In SPDY2 and SPDY3, the entire name/value header block is in a single compression context, with the LZ77 compression algorithm applied to the Huffman-encoded version of the entire header block. Hence, any substring which is repeated in the block is replaced by an encoded version. As parts of the headers are controllable by attackers (e.g. the URL to retrieve in the HTTP request), this made the partial chosen plaintext attack feasible.

In SPDY4, the compression context of the header block consists of entire key/value pairs. Each key/value pair has its own compression context, as it is either unmatched and emitted uncompressed, or matched fully and emitted as a delta-encoded version (i.e. the LRU index where the key/value pair was previously written in the compressor's internal state).

Compression, i.e. replacement with a shorter encoding, in the 'delta' compressor occurs only when the entire key and its corresponding value had previously occurred in the request stream. Sensitive data such as cookies are stored in the values of name/value pairs. Hence, to induce a "compression", or replacement of the secret cookie value with a shorter delta-encoded version, the attacker needs to correctly guess the entire cookie before he can obtain feedback.

By the analysis in §2.3.2, this would increase the number of guesses needed from $O(ns)$ in SPDY2 and SPDY3 with compression enabled, to $O(n^s)$ under the new delta compressor for SPDY4.

3 Compression Vulnerabilities

3.1 Compression Leaks in SPDY4

SPDY4 is still susceptible to chosen plaintext attacks through information leakage via compression. However, the attacker will receive feedback only when his entire guess is correct, and the attacker will not receive any partial feedback on guesses on substrings of the targeted secret in the plaintext. As discussed above, the number of guesses is exponential in the size of the character set of the secret.

3.1.1 "Duplicate" cookies using the `Path` attribute

Under the SPDY4 'delta' compressor, the attacker needs to inject a cookie containing his guess of the secret cookie value. This cookie must have the same name as the cookie containing the secret cookie, and its value must match the exact secret value exactly. However, if the victim's browser already contains a cookie with a given name, setting another cookie with the same name will cause the previous cookie to be overwritten. Hence, the attacker needs to be able to induce the web browser to send two cookies with the same name without one overwriting the other.

We have discovered that this can be achieved by making use of the `Path` attribute of a cookie. We tested the following approach in both Firefox 14 and Chrome 26. First, two cookies with the same name are set with different values for the path attributes. Second, when the browser sends subsequent requests to the site, both cookies with the same name will be sent in the request, but the browser will not send the value of the `Path` attribute, which is exactly the behavior we require for the attacker to inject his guesses.

3.1.2 SPDY4 Guess injection using duplicate cookie injection attack

Hence, consider the scenario where the attacker is trying to steal the value of a cookie with name `secretcookie` from `https://www.good.com/`.

First, we assume that the attacker knows the length of the secret contained in the cookie in the plaintext which he wishes to extract. This can be easily achieved by directly accessing `https://www.good.com/` to discover the format of the secret cookie, such as an MD5 or SHA1 hashed string.

$$\sum_i \binom{20}{w_i} \cdot 1^{w_i} \cdot \binom{20-w_i}{x_i} \cdot 5^{x_i} \cdot \binom{20-w_i-x_i}{y_i} \cdot 7^{y_i} \cdot \binom{20-w_i-x_i-y_i}{z_i} \cdot 3^{z_i}.$$

Figure 2: Calculation for number of possible guesses for one solution to equations 1,2 for strings with encoded length L for hexadecimal strings with lowercase letters.

Second, the attacker can inject JavaScript into any HTTP request to insert an IFrame which sends a request to `https://www.good.com/`, and the attacker observes the network to record the length of the request sent.

Third, the attacker needs to set a cookie with the name `secretcookie` with a `Path` attribute that has a different value from that of the legitimate cookie set by `https://www.good.com/`[2] for the `good.com` domain. The attacker can do this by injecting malicious JavaScript if the victim tries to connect to `good.com` using only HTTP without encryption[3], or by first carrying out a DNS poisoning attack, or a TLS forwarding attack, to temporarily redirect `good.com` to point to the attacker's servers so that the attacker can set a cookie with name `secretcookie` and a different `Path` attribute value for the `good.com` domain, and the attacker sets the cookie to contain the value of his guess of the full contents of the cookie string.

Fourth, the attacker can inject JavaScript into any HTTP request to insert an IFrame which then sends a request to `https://www.good.com/` and the attacker observes the length of the request. Suppose the length of the observed request in the second step is S_R, and the length of the secret cookie to be extracted has length S_S, and that the length of the request in the third step is S_T. Then, if $S_T - S_R < S_S$, we know that the attacker-injected guess was compressed by the SPDY4 'delta' compressor replaced the inserted attacker guess with the secret cookie's LRU entry ID, and that the guess was correct.

Although this allows guesses to still be injected into SPDY4 streams, the attacker can only gain feedback for chosen plaintext, but not for partial chosen plaintexts. Hence, the attacker does not gain feedback for partial guesses, and as described in §2.3.2, the number of guesses will be $O(n^s)$ for a secret of length s with n characters in the character-set of the secret, which is impractical for most actual data.

3.2 Leaks Due to Huffman Encoding in SPDY4

SPDY4 is also susceptible to information leakage due to the use of a fixed Huffman coding table. Different characters in SPDY4's fixed Huffman coding table have different encoding lengths. Hence, we can reduce the search space of possible strings encoded using the fixed Huffman coding table when given the following pieces of information:

1. Number of characters in the plaintext string
2. Length of the encoded string

3.2.1 Illustration of Information Leakage

We illustrate how a fixed Huffman coding table can result in information leakage using a simple example. Suppose that there is a fixed Huffman coding table, where character encodings can be 5-, 6-, or 7-bits long. Table 1 illustrates the number of characters with each encoding length in our example coding table.

[2]The attacker can first check the attributes of the targeted `secretcookie` set by `https://www.good.com/` to ensure that he picks a value for the `Path` attribute that is different.

[3]As our attacker does not have access to the server's encryption keys nor certificate, he will not be able to inject any content into an encrypted stream.

Encoding length	Number of characters
5-bits	2
6-bits	3
7-bits	3
Total characters	8

Table 1: Number of characters with each coding length in our illustration.

Then, suppose that we know that a plaintext string contains 3 characters, and we know that the string is 15-bits long when encoded using this coding table. Thus, we can conclude that the encoded string must contain 3 encoded characters which are 5-bits long.

Hence, if we can observe only string lengths, if we observe the length of the unencoded string, the number of possible strings is 8^3. However, if we can observe the length of the encoded string, then for a 15-bit encoded string, the number of possible strings is 2^3, because we know that the string is made up of the 2 characters whose encodings are 5-bits long.

3.2.2 Quantifying Information Leakage via Fixed Huffman Coding

Next, we describe how to quantify the information leaked due to the use of a fixed Huffman coding table. We assume that the attacker knows: (i) the number of characters in the secret contained in the plaintext which the attacker wishes to extract, and (ii) the length of the encoded secret.

We also make use of the possible number of bits in the character encodings in the Huffman table. As an example, we consider the encodings of hexadecimal string in lowercase, i.e. the character set consists of `0-9a-f`. In the fixed Huffman coding given in SPDY4, the lengths of the encoded characters for `0-9a-f` are as shown in Tables 2 and 3.

Character	0	1	2	3	4	5	6	7
Encoding length (bits)	5	5	5	6	7	6	7	6
Character	8	9	a	b	c	d	e	f
Encoding length (bits)	7	6	5	6	5	6	4	6

Table 2: Number of bits in encoding in fixed Huffman encoding table for SPDY4 for each character.

Number of bits in encoding	Number of characters
4	1
5	5
6	7
7	3
Total	16

Table 3: Summary of number of characters with each encoding length for characters used in lowercase hexadecimal string representations.

Then, given the length of the encoded secret, we can compute the number of characters of each encoding length that could have resulted in an encoded secret with the observed length. For instance, for a 20 character (lowercase) hexadecimal string, the encoded length can be between $20 \times 4 = 80$ bits to $20 \times 7 = 140$ bits. Supposing the length of the encoded string is L bits, and denoting the number of characters encoded with 4, 5, 6, and 7-bits respectively as w, x, y, z, then, the number of characters with 4-bits, 5-bits, 6-bits, and

7-bits in their encoding are given by the following system of equations:

$$w+x+y+z = 20 \quad (1)$$
$$4w+5x+6y+7z = L \quad (2)$$

We obtain the possible solutions for this system of equations by a brute-force recursive computation. Then, we denote the number of characters with k-bit encodings by E_k. Thus, suppose the solutions to the system of equations above are given by $(w_1,x_1,y_1,z_1),(w_2,x_2,y_2,z_2),\ldots$, then the number of possible guesses for the string of length L for each of the solutions (w_i,x_i,y_i,z_i) to the above system of equations is given by the summation in Figure 2.

The computation is similar for other character-sets of target guesses, with the number of variables (i.e. w,x,y,z) needed equal to the number of distinct encoding lengths in the fixed Huffman encoding table needed to encode the target character-set of the secret.

In this case, the number of possible strings for a 20-character hexadecimal string is $16^{20} = 2^{80} \approx 10^{24}$. However, in the extreme case, if the length L of the 20-character (lowercase) hexadecimal string is 80-bits, then since there is only one character with a 4-bit encoding, then the search space is immediately narrowed down to a single possible string. In a less extreme case, suppose the length L of the 20-character (lowercase) hexadecimal string is 100-bits, then based on the formula in Figure 2, there are $1.22 \times 10^{19} \approx 2^{63}$ possible strings, and we would have achieved a 17-bit reduction in the search space due to the information leaked from the length of the encoded string.

3.2.3 Generalizing Quantification of Information Leakage via Fixed Huffman Coding

Next, we generalize the equations in Figure 2 for quantifying the information leakage due to the use of a fixed Huffman Coding (as compared to an unencoded string with fixed-length characters). First, Figure 3 gives the system of equations which specifies the linear combinations of the number of each of the encoded characters that can make up an encoded string with length L-bits if it has C characters.

$$\sum_{j=1}^{m} x_{i,j} = C$$
$$\sum_{j=1}^{m} |l_j| \cdot x_{i,j} = L$$

Figure 3: System of simultaneous equations which gives linear combinations of the number of (encoded) characters with each encoding length given number of characters in string, C, and length of string, L in bits. Each $\forall j \in [1,m], l_j$ is a length of encoded characters in the Huffman coding, and $|l_j|$ indicates the number of characters whose encodings are l_j long.

Thus, we obtain, from the equations in Figure 3, the n linear combinations $\forall i \in [1,n], x_{i,1}, x_{i,2}, \ldots, x_{i,m}$ of the numbers of each encoded character length which makes up the encoded string of length L with C characters where each $x_{i,1},\ldots,x_{i,m}$ is one such linear combination. Then, Figure 4 gives the number of guesses for a string with encoded length L with C characters.

Hence, from Figures 3,4, we denote by G the number of possible guesses for a string with encoded length L-bits which is known to have C characters. Then, the number of possible strings with C characters is given by M^C. Hence, the amount of information leaked is given by:

$$G = \sum_{i=1}^{n} \prod_{j=1}^{m} \binom{L - (\sum_{k=1}^{j-1} x_{i,k})}{x_{i,j}} \cdot |l_i|^{x_{i,j}}$$

Figure 4: General form of equation in Figure 2 for counting number of guesses, where $x_{i,j}$ denotes the j-th entry in the i-th solution to the system of equations in Figure 3, and l_i denotes an encoding length and $|l_i|$ denotes the number of characters in the character-set with encoding length l_i.

$$\frac{\log(M^C - G)}{\log 2}$$

Figure 5: Information leakage due to fixed Huffman coding table.

3.3 Information Leakage in Byte-padded Strings

The amount of information leaked in strings padded to the nearest byte-length is reduced due to the byte-padding. As the SSL/TLS protocol which SPDY4 is wrapped in specifies that frames are to be padded to the nearest byte-length, only byte-lengths will be observable. Hence, instead of observing the bit-length L of the encoded string, an attacker will only be able to observe the byte-length $\lceil \frac{L}{8} \rceil$ of the encoded string. Thus, for an observed byte-length L', the attacker must consider bit-lengths $(8 \times (L' - 1)) + \{1,2,3,4,5,6,7,8\}$ and the possible strings that can be encoded with the given bit-lengths.

4 Exploiting SPDY4 Leakage

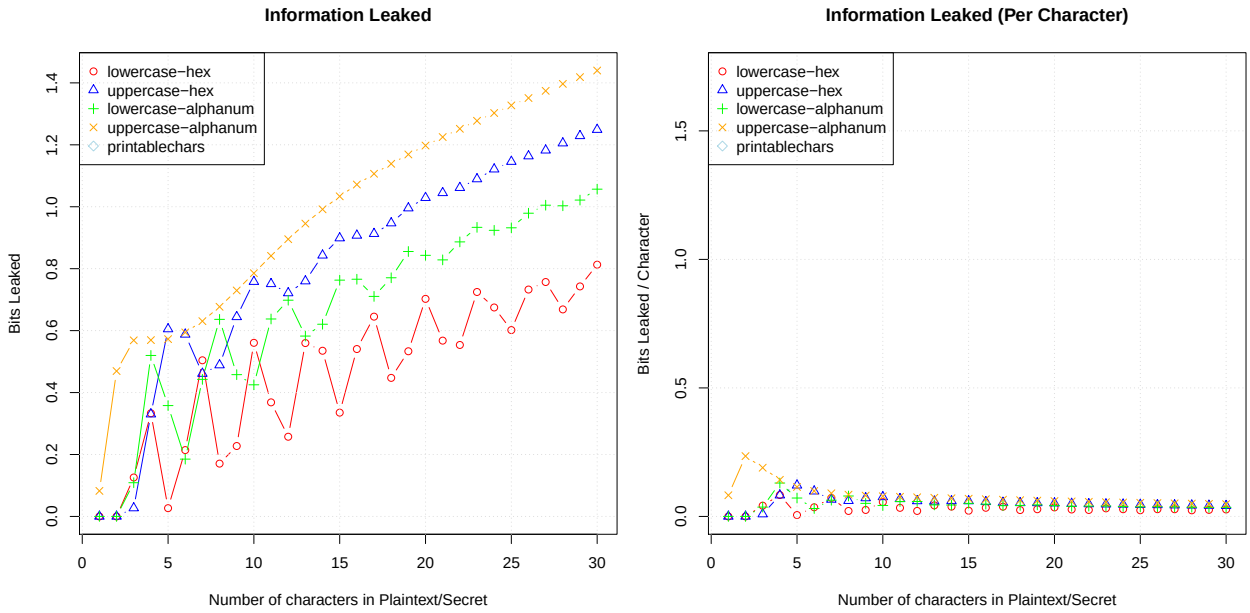
4.1 Information Leakage in SPDY4

Next, we applied the equations in Figures 3 and 4 to various possible character-sets for SPDY4, given the SPDY4 Fixed Huffman Coding table. We used the expression in Figure 5 to compute the number of bits leaked for various character-sets in SPDY4 given the fixed Huffman coding table specified for SPDY4 requests. In addition, we used the procedure described in §3.3 to consider the amount of information leaked when only byte-lengths of encoded strings are observable (rather than bit-lengths).

For a given string with C characters for a given character-set, there are a number of possible encoded string lengths. To summarize our findings, we calculated the expected information leakage, by computing the information leaked for each possible encoded string length L, and weighting it by the proportion of strings in the space of all possible strings with that given string length.

We consider various types of character-sets, as the different character-sets have different distributions of encoding characters of various lengths, which can affect the amount of information leaked for each character-set. Specifically, we quantified the information leaked due to the fixed Huffman coding for the following four character-sets:

- Lower-case hexadecimal strings `0-9a-f`
- Upper-case hexadecimal strings `0-9A-F`
- Lower-case alphanumeric strings `0-9a-z`
- Upper-case alphanumeric strings `0-9A-Z`



(a) Total number of bits of information leaked.

(b) Number of bits of information leaked per character in the secret/plaintext.

Figure 6: Amount of information leaked due to the use of a fixed Huffman coding table for various types of character-sets in SPDY4.

Figure 6 shows the amount of information leaked for each of the character-sets we considered. We computed the amount of information leaked for strings with 1 to 30 characters for 4 possible character-sets (lowercase and uppercase hexadecimal and alphanumeric strings). We did this in the following procedure. First, we computed the number of possible strings of each observable byte-length, and the possible bit-lengths for each observed byte-length. Second, we computed the number of possible plaintext strings with each encoded length. Third, we computed the conditional expectation of the number of guesses for each plaintext string conditioned on each observed length of the encoded string. Fourth, we computed the information leaked as the number of bits needed to represent the difference between the number of possible plaintext strings of each given plaintext length, and the expected number of guesses for plaintext strings of the given plaintext length when given the length of the encoded string.

Hence, the information leaked for a plaintext string of length M, measured in bits, where the character-set has C characters, i represents the observable lengths of the encoded strings, and G_i represents the number of possible strings with encoded length i (computed from Figure 4), can be computed as follows:

$$\frac{\log\left(M^C - \sum_i \left(\frac{G_i}{M^C} G_i\right)\right)}{\log 2}$$

From Figure 6(a), we can see that the total number of bits leaked increases with increasing length of the plain-text string. The number of bits leaked is the greatest for uppercase alphanumeric characters, followed by lowercase alphanumeric characters, while the fewest bits are leaked for hexadecimal strings. This is because the hexadecimal characters have the smallest variation in the number of bits used to represent the different characters in its character-set, whereas the variation in the number of bits used to represent the larger character-sets of the alphanumeric strings gives away more information through the length of the Huffman-coded strings.

From Figure 6(b), we can see that the number of bits leaked on a per-character basis (i.e. total number

of bits leaked divided by the number of characters in the plaintext) levels off for all the different character-sets. This is because as the number of characters in the plaintext string increases, the number of linear combinations satisfying the equations in Figure 3 increases, increasing the number of possible guesses. The number of bits leaked per character is highest for alphanumeric strings, and lowest for hexadecimal strings. This is consistent with our observation above that larger character-sets leak larger amounts of information.

In addition, from Figure 6(a), we can see that the number of bits leaked varies slightly for differing numbers of characters in the plaintext string. This is because of the interaction between the possible bit-lengths and the observable byte-lengths. Plaintext strings with a particular number of characters may have different possible observable byte-lengths, and strings whose encodings have more possible observable byte-lengths can result in slightly more information leaked.

4.2 Hypothesized Attack

We showed in §4.1 that between 15% to 45% of the number of bits used to encode strings with various character-sets is leaked. An attack can be mounted against SPDY4 to recover plaintext from encrypted data using a chosen plaintext attack. This attack requires: (i) Network attack: Observe traffic, and inject data into unencrypted streams. (ii) No circumvention of encryption: No HTTPS keys needed, no certificate errors will be caused. (iii) Number of characters in plaintext to be recovered. (iv) Character-set of plaintext.

Then, the attack is a variant on the guess-injection described in §3.1.2. The attacker continues to use the technique in §3.1.2 to inject guesses, but the guesses are generated as follows:

- After obtaining the length and character-set of the secret cookie, the attacker computes the linear combinations of the different numbers of each character of each encoding length.
- For each i-th linear combination, the attacker constructs guesses for the linear combination. $\forall j \in [1,m], x_{i,j}$, $x_{i,j}$ specifies the number of characters with encoding lengths l_j. Thus, for each j, the attacker picks $x_{i,j}$ characters from the fixed Huffman-coding table whose encoded lengths are l_j.
- From the previous step, the attacker picks the selected characters, and constructs guesses by permutating the selected characters.

4.3 Implications

Quantifying Information Leaked The number of bits leaked increases rapidly for short plaintext strings, but appears to level off as the plaintext string gets longer. This is likely because as the number of characters in the plaintext string increases, the number of linear combinations satisfying the equations in Figure 3 increases, increasing the number of possible guesses. The number of bits leaked, as illustrated by Figure 6(a), levels off around 1.4 bits of information leaked for 30-character plaintext strings consisting of uppercase alphanumeric characters.

Quantifying Request Volume Required Next, we determined the volume of requests required to generate sufficient guesses for each plaintext length to be recovered. For this exercise, we focus on strings whose character-set is made up of uppercase alphanumeric characters. The results for hexadecimal and lowercase alphanumeric strings is similar, and are a linear factor of the results for uppercase alphanumeric characters.

Figure 7 shows the volume of requests required to be sent to inject each possible guess for both the brute-force attack and our proposed fixed Huffman-coding attack. We conservatively assume that each request is 200 bytes in size, and that each guess requires a request to be sent twice, according to our cookie-overwriting attack described in §3.1.2 and §4.2. We can see that the volume of requests needed is similar for both brute-force guessing, and when guessing using our information leak. This is because the amount of information leaked remains small at under 2-bits even for 30-character plaintext strings.

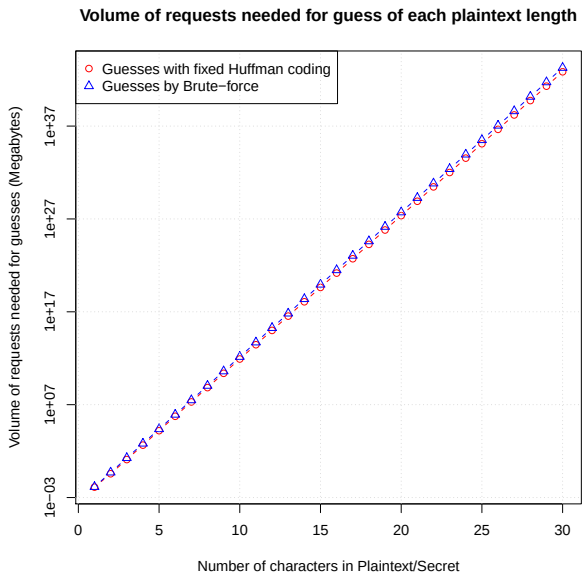


Figure 7: Volume of data needed to generate requests for all guesses for both brute-force guessing, and for guessing assisted by information from the observed Huffman coding length for strings of printable characters.

Quantifying Time Required Next, we quantify the time required to transmit all guesses. We assume a typical broadband connection, and we optimistically assume a bandwidth of 5.0 megabits per second (or 0.625 megabytes per second). We also assume that the attacker does not take care to reduce his network footprint by limiting the rate of transmission of guesses. Figure 8 shows the time taken to transmit requests for all guesses for each plaintext length being guessed. Again, we can see that the time taken for a brute-force guess is similar to the time taken when guesses are informed by the observed length of the Huffman-coded string.

5 Related Work

Information Leaks in SSL/TLS Two prominent attacks against the SSL/TLS encryption protocol for transporting encrypted HTTP traffic were the BEAST (Browser Exploit Against SSL/TLS) and CRIME (Compression Ratio Information Leakage Made Easy) attacks [5] which enabled the recovery of HTTP session cookies using chosen plaintext attacks. These attacks enabled a network attacker with the ability to sniff network traffic to recover HTTP session cookies from an encrypted SSL/TLS session. The key approach was to inject guesses of the value of the session cookie from the victim's host to the target server, and observe the encrypted traffic for feedback about whether the guess was correct. In BEAST, the feedback was in the form of the XORed encrypted text matching the guessed value due to the initialization vectors (IVs) used in the Cipher Block Chaining (CBC) mode being taken from the last plaintext block, which allowed the attacker to match his guess against a previous block containing the secret. In CRIME, the feedback was in the form of shorter encrypted text length due to better compression from the injected guess matching the secret HTTP session cookie value.

Our hypothesized PETAL information-leak attack against the proposed compression scheme in SPDY4 is analogous to the BEAST and CRIME attacks, in that the attacker injects guesses of the HTTP session cookie string, and receives feedback in some form. In the case of PETAL, the feedback is similar to CRIME:

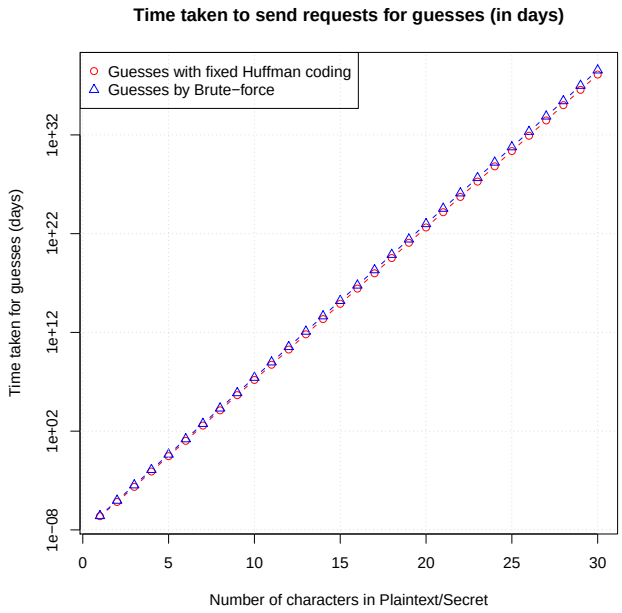


Figure 8: Time needed to generate requests for all guesses for both brute-force guessing, and for guessing assisted by information from the observed Huffman coding length for strings of printable characters.

the attacker can determine if his guess is correct by observing if the length of the encrypted text falls as compared to other guesses. However, unlike BEAST, PETAL does not rely on the type or mode of the encryption used in the underlying SSL/TLS layer (Cipher Block Chaining mode). In CRIME, the attacker is able to receive feedback for correct partial guesses, leading to a highly practical attack which can recover long strings (e.g. greater than 20 characters long for actual session cookies of current websites), whereas in PETAL, due to the effective mitigations against partial chosen plaintext attacks in the proposed compression in SPDY4, the attacker in PETAL is able to receive feedback only for a complete correct guess. However, the amount of information leaked when the encoded strings are byte-padded is very small, and under 2-bits even for a 30-character string.

Other SSL/TLS Vulnerabilities A number of other vulnerabilities in SSL/TLS have been discovered. First, Paterson et al. identified ways an attacker can recover plaintext from a TLS connection when RC4 encryption is used [9]. These attacks are due to statistical flaws in the keystream generated by RC4 which become apparent in TLS ciphertext when the same plaintext is repeatedly encrypted at fixed locations across many TLS sessions. We speculate that using SPDY on top of SSL/TLS is likely to reduce the vulnerability to such attacks due to SPDY being designed to persist sessions, although attackers can force TLS sessions to be terminated to cause multiple sessions to be created. The US National Vulnerability Database has assigned a vulnerability entry to this flaw [1]. Paterson et al. have also identified the Lucky-13 attack [2] which enables the attacker to guess the contents of ciphertext by sending malformed messages to the server, and observing the time taken for decryption to fail. The TLS MAC calculation includes 13 bytes of header information, so the attacker can replace the last few blocks of ciphertext with chosen blocks and observe the time taken for the server to respond. TLS messages with the correct padding will require less time on the server to process. Hence, attackers could guess the contents of ciphertext by timing the server's response time. These attacks are orthogonal to our PETAL information-leak on SPDY4, as SPDY4 is layered on top of the SSL/TLS protocols.

6 Future Work

Our attack is a hypothesized attack against the proposed compression scheme in the next version of the SPDY protocol. Hence, the attack needs to be verified against a working implementation of the next version of the protocol.

In addition, we believe that our information-leak attack against fixed encoding tables with differing lengths of encoded characters can be generalized beyond the SPDY protocol.

7 Conclusion

We have describe the CRIME vulnerability [11] and how it applies to SPDY2/3. We have also described the mitigations against CRIME-like vulnerabilities in the new compression scheme in SPDY4, and we have shown that this mitigation is effective against partial chosen plaintext attacks.

In addition, we have described a source of information leakage in SPDY4 due to its use of a fixed Huffman coding table, which enables a chosen plaintext attack by a network attacker when combined with our method for sending duplicate cookies with the same name by using a different `Path` attribute. We have quantified the implications of the amount of information leaked, and shown that an attack which makes use of this information leakage is not practical, as it leaks only less than 2-bits of information for 30-character uppercase alphanumeric strings.

Acknowledgements

We would like to thank Collin Jackson and Lin-Shung Huang for helpful discussions and suggestions for this paper. We would also like to thank Roberto Peon and Adam Langley for feedback and helpful discussions on this paper.

References

[1] Cve-2013-2566, Mar 2013. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2566.

[2] N. AlFardan and K. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2013.

[3] HTTPbis Working Group . Header Delta-Compression for HTTP/2.0. http://datatracker.ietf.org/doc/draft-rpeon-httpbis-header-compression/.

[4] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[5] IETF 85 SAAG. BEAST & CRIME: How TLS was attacked, Nov 2012. http://www.ietf.org/proceedings/85/slides/slides-85-saag-1.pdf.

[6] IETF Network Working Group. DEFLATE Compressed Data Format Specification, May 1996. http://tools.ietf.org/html/rfc1951.

[7] IETF Network Working Group. ZLIB Compressed Data Format Specification version 3.3, May 1996. http://tools.ietf.org/html/rfc1950.

[8] John Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2002.

[9] K. Paterson. On the security of rc4 in tls, Mar 2013. http://www.isg.rhul.ac.uk/tls/index.html.

[10] R. Peon. SPDY/4 Compression Update. https://groups.google.com/forum/#!msg/spdy-dev/3W8YlAIgnls/ALI2C10dX5wJ.

[11] J. Rizzo and T. Duong. The CRIME Attack, Sep 2012. https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU.

[12] The Chromium Project. SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[13] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.