Global Information Assurance Certification Paper

Practical Approach to Detecting and Preventing Web Application Attacks over HTTP/2

GIAC (GCIA) Gold Certification

Author: Russel Van Tuyl, Russel.VanTuyl@gmail.com

Advisor: Stephen Northcutt

Accepted: April 6, 2016

Abstract

HTTP/2 is a newly ratified protocol that builds upon current web communications to increase efficiency and overcome shortfalls of the HTTP/1 protocol. This new protocol is intended to be used only over TLS connections and as such is the only method supported by the top web browser applications. Because this protocol is relatively new, there is a lack of tools capable of inspecting the protocol to detect or prevent attacks against web applications. The protocol's use of Perfect Forward Secrecy TLS cipher suites further complicates matters by preventing inspecting technologies from capturing the keying material required to decrypt traffic for inspection. This paper provides a little background on the HTTP/2 protocol and TLS connections in conjunction with an evaluation of web browser support. Several architectures will be evaluated as a method to detect and prevent web application attacks over HTTP/2 using currently available tools.

1. Introduction

The Hypertext Transfer Protocol (HTTP) was first defined in 1991 by the World Wide Web initiative as method to retrieve hypertext markup language (HTML) messages (Berners-Lee). This protocol has become a staple in modern computing, used as a primary communication method on the Internet to both support business operations and personal endeavors. The HTTP/1 protocol as originally designed is starting to show its age and faces troubles keeping up with modern performance requirements and operational uses. Some of the problems with HTTP/1 include inadequate use of transmission control protocol (TCP) connections, latency, and instances where one packet holds up the transmission of other packets known as head-of-line blocking (Stenberg, 2015). A new protocol is needed to overcome these obstacles.

HTTP/2 is a newly ratified protocol documented under request for comments (RFC) 7540 that aims to solve some of the problems with HTTP/1.x and provide functionality to support current web application operations (Belsche, Peon, & Thomson, 2015). This new protocol utilizes bi-directional connections in which both the server and client are free to communicate with each other at will. Connections such as this, along with other new features of the HTTP/2 protocol, will significantly enhance web application communications by increasing communication efficiency.

With the birth of any new protocol comes both new applications that capitalize on its features as well as new avenues to perform malicious activities. This paper explores practical approaches to detecting and preventing web application attacks over the new HTTP/2 protocol. One of the bigger stumbling blocks associated with a new protocol is understanding how it works to include its capabilities and limitations.

1.1. HTTP/2 Protocol Summary

HTTP/2 communications are multiplexed, bi-direction connections that do not end after one request and response (Belsche, Peon, & Thomson, 2015). A session can stay open and established for as long as the client or server allow. The HTTP/2 specification allows the protocol to operate over both clear-text and encrypted communication channels. The clear-text version of HTTP/2 is identified by ‘h2c’ and encrypted communications as ‘h2’. Additionally, HTTP/2 connections can be established using an

existing HTTP/1.1 connection or during the setup of a transport layer security (TLS) connection.

1.1.1. HTTP/2 Clear-Text h2c

The HTTP/2 clear-text h2c identifier breaks down to h for HTTP, 2 for version 2 of HTTP, and c for clear-text. According to the specification, the h2c protocol is negotiated by the client over HTTP/1.1 using the "Upgrade" header in a request to the server as shown below in Figure 1 highlighted in red. The request "MUST include exactly one HTTP2-Settings header field" according to the RFC (Belsche, Peon, & Thomson, 2015, p. 7). If the server supports h2c, it will respond with an HTTP 101 status code as shown in Figure 1 highlighted in blue. To conclude the negotiation, the client will reply with the connection preface (also known as the magic string) 'PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n' to ensure the server really supports HTTP/2 as shown in Figure 1 highlighted in red (Belsche, Peon, & Thomson, 2015, p. 10). At this point, both the client and server will now communicate in clear-text over HTTP/2.

```
GET / HTTP/1.1
Host: 127.0.0.1
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: AAMAAABkAAQAAP__
Accept: */*
User-Agent: nghttp2/1.3.4

HTTP/1.1 101 Switching Protocols
Upgrade: h2c
Connection: Upgrade

..............d............PRI * HTTP/2.0

SM
```

Figure 1: HTTP/1.1 h2c Negotiation

1.1.2. HTTP/2 Encrypted h2

An encrypted HTTP/2 communication channel, identified as h2, is established within the TLS protocol prior to HTTP communications. This type of HTTP/2 communication does not use the HTTP "Upgrade" header. The Application-Layer Protocol Negotiation (ALPN) TLS extension, documented in RFC 7301, is used to negotiate the use of the h2 protocol (Belsche, Peon, & Thomson, 2015). ALPN is used to

announce support for multiple application protocols on a given port and to negotiate the use of one of them such as h2, h2c, or http/1.1 (Friedl, Popov, Langley, & Stephan, E 2014). Support for TLS extensions can only be found in TLS 1.2, documented in RFC 5246. According to the OpenSSL Software Foundation, support for the ALPN extension was added to OpenSSL, a popular TLS library, in version 1.0.2 on January 22, 2015. If the client supports HTTP/2, it will send the string 'h2' in the ALPN extension field of its client hello message during a TLS session setup (see Figure 2).

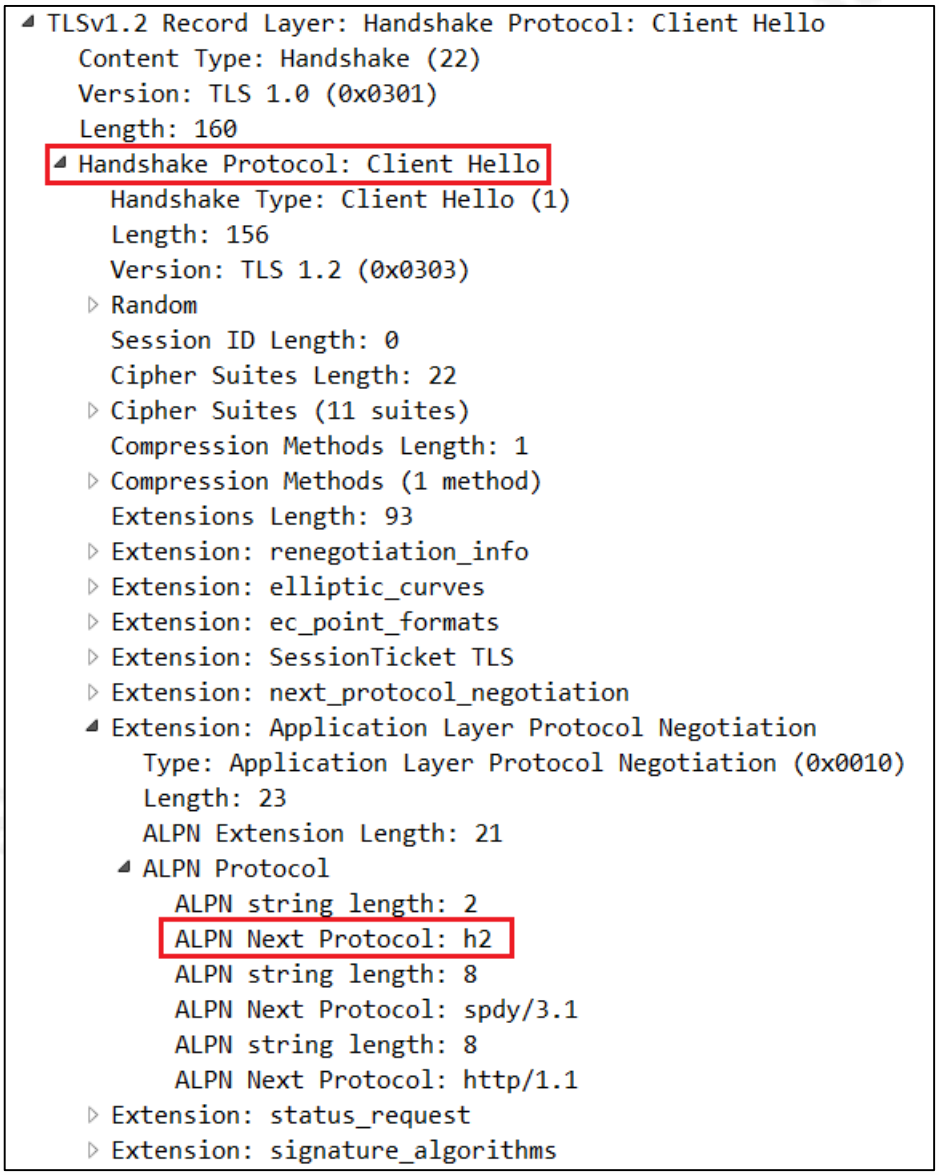

Figure 2: TLS Client Hello ALPN

If supported, the server will select the h2 protocol as the communication method and notify the client with the string ‘h2’ in the ALPN extension field of its server hello message (see Figure 3).

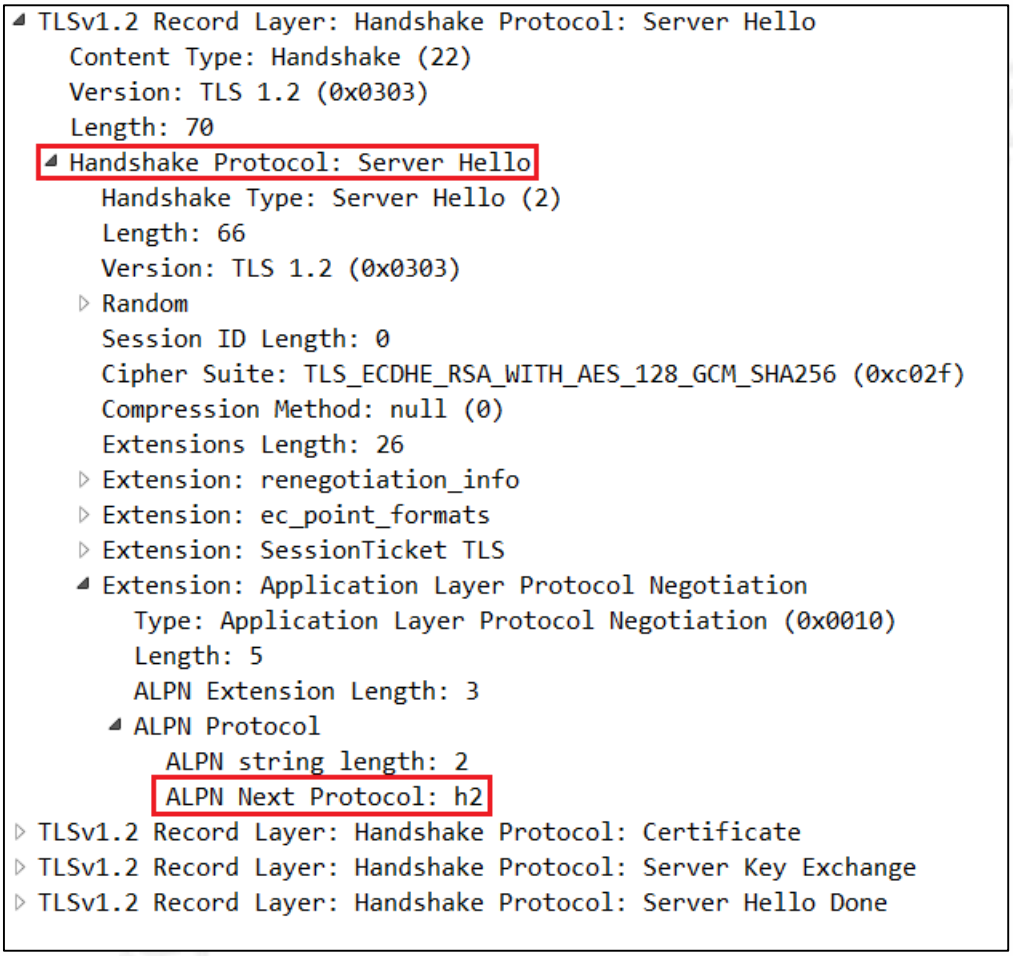

Figure 3: TLS Server Hello ALPN

Once the TLS handshake has completed, the client and server will setup an HTTP/2 connection using a number of different HTTP/2 frames. The client will again start by sending the connection preface (also known as the magic string) ‘PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n’ (see Figure 4). Traffic between the client and server is protected against eavesdropping and inspection once the h2 encrypted TLS communication channel is setup.

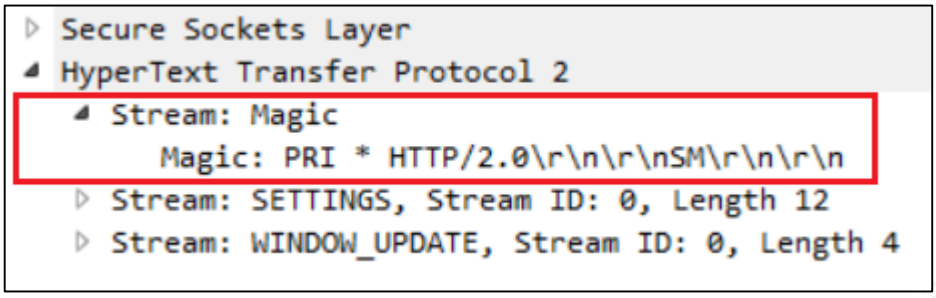

Figure 4: TLS Magic String

1.2. TLS Encryption

In order to establish a TLS connection, the client and server must agree on a cipher suite to secure communication channel. The server will select a TLS cipher suite that is supported by both the client and the server. The client provides a list of cipher suites it supports in its Client Hello message to the server.

The Elliptic Curve Diffie-Helman Exchange (ECDHE) and ephemeral Diffie-Helman Exchange (DHE) key exchange methods benefit from Perfect Forward Secrecy (PFS) (Bernat, 2011). PFS ensures that the successful compromise of one session will not allow all other connections between a client and server to be compromised. The HTTP/2 protocol recommends that all non-PFS-enable cipher suites are black listed, documented in Appendix A of RFC 7540 (Belsche, Peon, & Thomson, 2015). The RFC reads as follows: "An HTTP/2 implementation MAY treat the negotiation of any of the following cipher suites with TLS 1.2 as a connection error of type INADEQUATE_SECURITY" (Belsche, Peon, & Thomson, 2015, p. 66). A total of 276 cipher suites are listed in Appendix A of RFC 7540. If the protocol has been implemented properly, both the client and server will only utilize PFS enabled TLS cipher suites to secure communications.

1.2.1. Current Web Client Landscape

The vast majority of HTTP communications take place between a web browser (the client) and a web application (the server). The major web browser vendors (Google, Mozilla, Microsoft, etc.) will not support h2c communications (Stenberg, 2015). Therefore, HTTP/2 communications taking place between a web browser and a web application will be encrypted using the h2 protocol. Therefore, the majority of HTTP/2 communications will occur over a TLS1.2 encrypted channel. It is important to know which TLS cipher suites are supported by each of the popular web browsers.

According to w3schools.com, the most common web browsers are Chrome, IE, Firefox, Safari, and Opera. Microsoft's Edge browser is a recent addition to its newest operating system. Figure 5 documents supported TLS cipher suites for each of these web browsers. The rows highlighted with green or yellow use the ECDHE or DHE key exchange method. The rows highlighted in red employ the RSA key exchange technique. This information is essential as it dictates what type of encryption will be used to secure HTTP/2 communications.

Cipher Suites	Opera	Chrome	Firefox	IE	Edge	Safari
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)				x	x	x
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)				x	x	x
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)	x	x	x	x	x	x
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)				x	x	x
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)				x	x	x
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)				x	x	
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)		x		x	x	
TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)				x	x	x
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc14)	x	x				
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc13)	x	x				
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)	x	x	x	x	x	x
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)	x	x	x	x	x	x
TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)		x	x			
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)	x	x	x	x	x	x
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)	x	x	x	x	x	x
TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)						x
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)						x
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc008)						x
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012)						x
TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)		x	x			
TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)	x	x		x	x	x
TLS_RSA_WITH_AES_256_CBC_SHA256 (0x003d)				x	x	x
TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)				x	x	x
TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)	x	x	x	x	x	x
TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)	x	x	x	x	x	x
TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)	x	x	x	x	x	x
TLS_RSA_WITH_RC4_128_MD5 (0x0004)						x
TLS_RSA_WITH_RC4_128_SHA (0x0005)						x
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256 (0x006a)				x	x	
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256 (0x0040)				x	x	
TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)				x	x	
TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)				x	x	
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x0013)				x	x	

Figure 5: Browser Supported TLS 1.2 Cipher Suites

Figure 6 documents the remaining cipher suites that are not black listed in the RFC but are supported by the major web browsers. If implemented according to the RFC, traffic between a web browser and web server will be utilizing these cipher suites. It is important to note that the RFC uses the word MAY when suggesting what TLS cipher

suites to support. Therefore, browser vendors can still support a black listed cipher suite if they choose. Opera, Chrome, and Firefox will not allow a connection to a web application when a web server is configured not to support ECDHE or DHE cipher suites. Internet Explorer, Edge, and Safari will allow a connection using the TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d) cipher suite. This cipher suite does not benefit from PFS as it does not utilize the ECDHE or DHE key exchange method.

Cipher Suites	Opera	Chrome	Firefox	IE	Edge	Safari
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)				x	x	x
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)				x	x	x
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)	x	x	x	x	x	x
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)	x	x	x	x	x	x
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)				x	x	
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)		x		x	x	
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc14)	x	x				
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc13)	x	x				

Figure 6: HTTP/2 Supported (Non-Black List) TLS Cipher Suites

1.2.2. Web Browser Tools

There is not an easy way for an end user to know when they're using HTTP/2 instead of HTTP/1.1. This is partly because the protocol was specifically designed to use current Uniform Resource Identifiers (URI) (Belsche, Peon, & Thomson, 2015). However, a web browser's built-in Web developer tools can be leveraged to determine the type of HTTP connection. For Opera, Chrome, and Firefox the Web developer console can be enabled by using the keyboard shortcut Ctrl + Shift + I. Click on the Network tab and then reload the web page. Right click on the Status header and select Protocol and you can now determine if HTTP/1.1 or HTTP/2 was used as shown below in Figure 7. For Internet Explorer and Edge, just press the F12 key and reload the page.

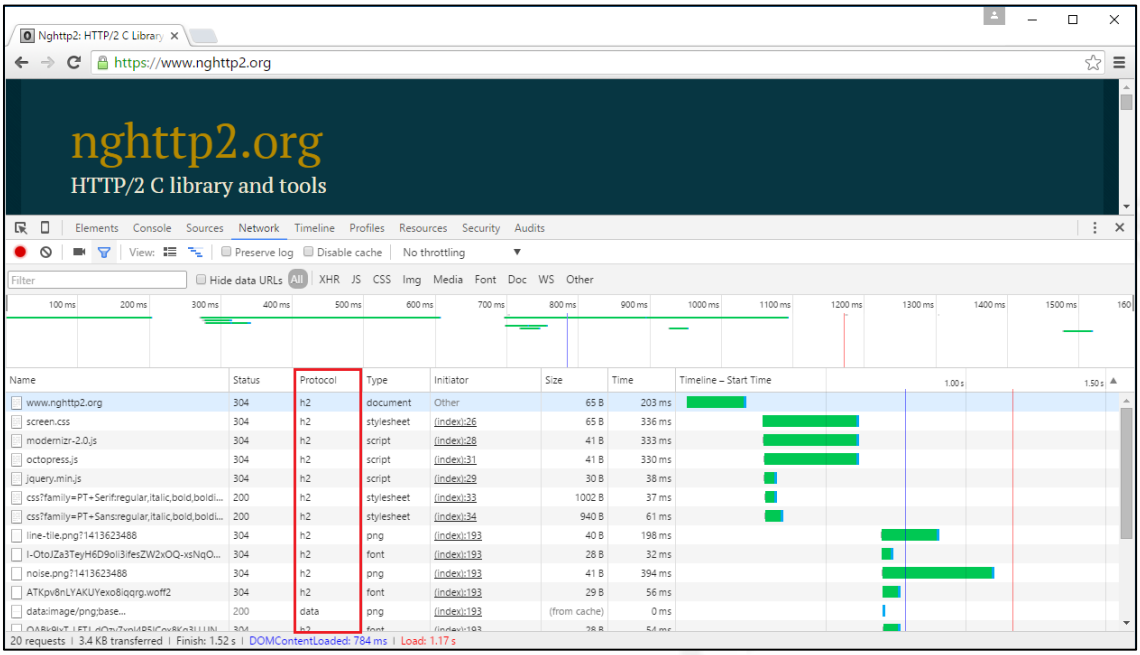

Figure 7: Web Console Protocol

Alternatively, both Chrome and Firefox can be enabled with a plugin that provides a status indicator to identify when a page was loaded over HTTP/2 as shown below in Figure 8. The Firefox plugin can be found at https://addons.mozilla.org/en-us/firefox/addon/spdy-indicator/ and the Chrome plugin at https://chrome.google.com/webstore/detail/http2-and-spdy-indicator/mpbpobfflnpcgagjijhmgnchggcjblin.

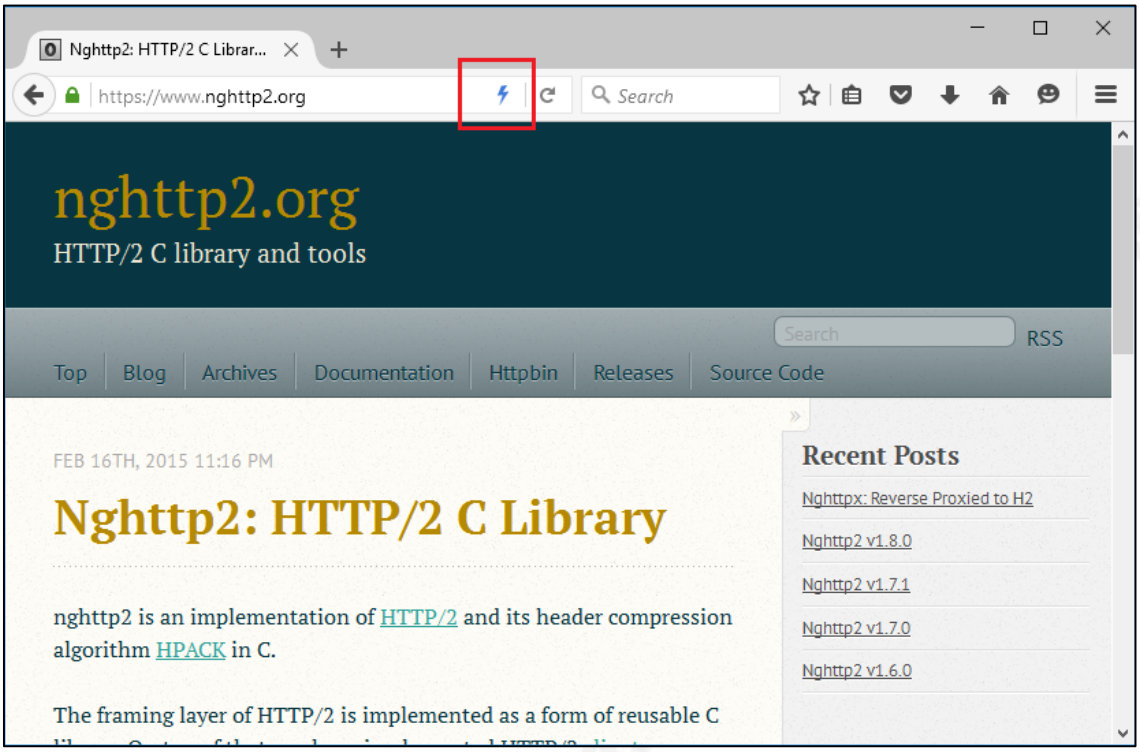

Figure 8: HTTP/2 Status Indicator

2. Inspecting HTTP/2 Traffic

With any new technology comes the potential for it to be abused or used as a communication method to perform malicious attacks. Protection technologies are often installed to defend valuable business assets. These technologies inspect traffic looking for nefarious actions and either alert administrators or block the activity. The most common of these technologies include an Intrusion Detection System (IDS), Intrusion Protection System (IPS), or Web Application Firewall (WAF). Unfortunately, these technologies are faced with challenges when it comes to handling the HTTP/2 protocol.

2.1. Packet Inspection

An IDS, IPS, or WAF can be configured to inspect traffic as it passes through the appliance on its way to target application (see Figure 9). In this configuration, the evaluating device must be able to decrypt packets so their contents can be inspected.

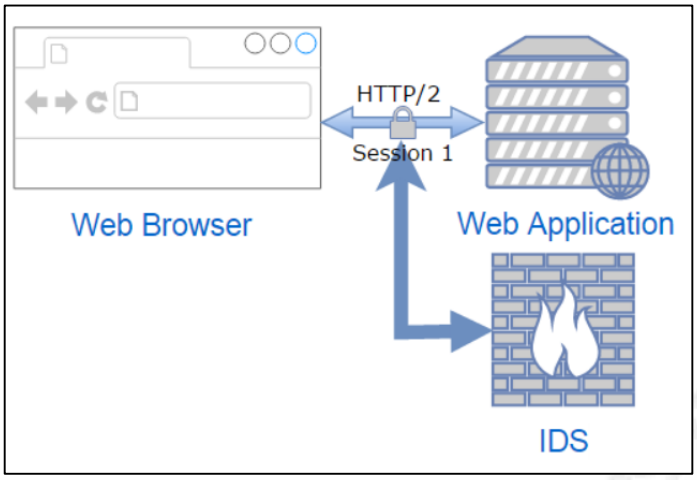

Figure 9: IDS Inspection Architecture

Current inspection technologies are provided with a target application's private key so that they can decrypt the traffic for evaluation. This method of packet inspection can only be utilized when an RSA key exchange method cipher suite (i.e. TLS_RSA_WITH_AES_256_GCM_SHA384) is used to secure communications between a client and a server.

Communication channels using the RSA key exchange method utilize the server's public key to encrypt a pre-master secret and then send it over the network to the server (Helme, 2014). The pre-master secret is used to derive session keys that are used to encrypt the communication channel. The server's private key can be applied by an analyst or an attacker to decrypt the previously encrypted pre-master secret and subsequently derive the session keys used to encrypt the communications.

When PFS is utilized, the pre-master secret is not encrypted with the server's public key and is not transmitted over the network (Helme, 2014). Additionally, ephemeral PFS cipher suites generate new session keys for every connection. This ensures that the successful compromise of one session won't allow all other connections between a client and server to be compromised.

To decrypt traffic that was secured using a PFS enable cipher suite, the inspecting application must have the pre-master secret to generate the session key so the traffic can be decrypted (Shaver, 2015). Web browsers that utilize the NSS library (Firefox & Chrome) can be configured to save each connection's session key to a file. Tools such as Wireshark can decrypt traffic secured with a PFS enabled cipher suite only when

provided with the file containing the session keys. This is only practical when an organization or analyst controls the client's host and can enforce the use of Firefox or Chrome. Because it is not practical to retrieve a connection's session keys from every client, tools such as SSLDump require weakened encryption (non-PFS cipher suites) to decrypt traffic successfully (Winkel, 2015).

HTTP/2 communications are expected to take place over TLS encrypted channels using PFS enabled cipher suites as discussed in section 1.2.1. Therefore, inspecting HTTP/2 traffic for web application attacks proves difficult. Another obstacle is that currently available WAF/IDS/IPS solutions are incapable of understanding the HTTP/2 protocol even if they were able to decrypt traffic for inspection. While current options for inspecting HTTP/2 traffic for web attacks are limited or non-existent, a couple of possibilities still exist.

2.2. Proxy

A WAF can also be used as a type of reverse proxy to inspect traffic for attack patterns. In this configuration, the WAF becomes a termination point of the connection from the client. A second connection is made from the WAF to the web application, proxying the connection from the client as shown below in Figure 10. This model is useful because the proxy is a TLS endpoint and therefore, can decrypt PFS enabled cipher suites because the connection was established between the client and the WAF.

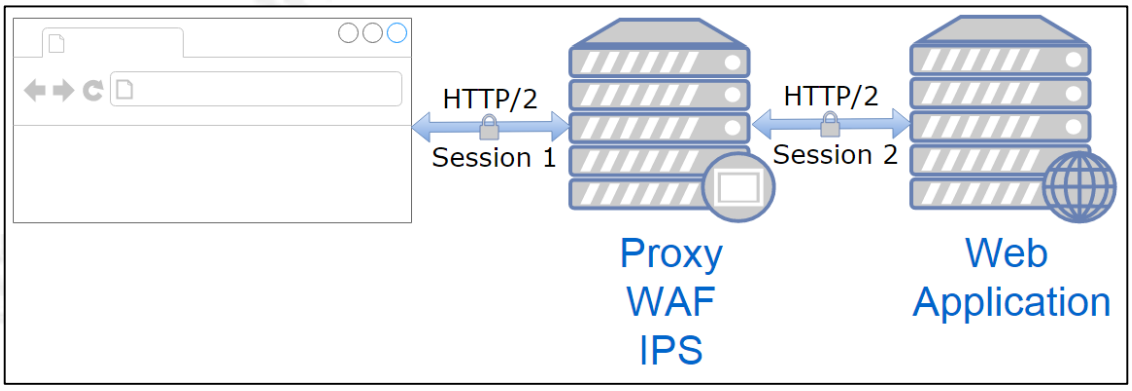

Figure 10: Proxy Architecture

Most currently available WAFs are not capable of establishing HTTP/2 connections with a client or a backend web application. Furthermore, they are not able to understand the new protocol in a way that facilitates malicious behavior detection. This

precludes their use as an effective means of protecting against attack web application attacks.

2.3. Protocol Downgrade

One potential method for inspecting traffic for web application attacks is to downgrade the protocol from HTTP/2 back to HTTP/1. This will allow current technologies that understand HTTP/1.1, but not HTTP/2, to be utilized like an IDS. A proxy that is capable of handling HTTP/2 connections, such as nghttpx, can be leveraged in this architecture (Tsujikawa, n.d.). The proxy can terminate the HTTP/2 connection from the client and then proxy the traffic to a WAF, IPS, or IDS over HTTP/1. Traffic can be sent with or without TLS encryption. Figure 11 illustrates this architecture model. This method is not ideal because it circumvents the value of using the HTTP/2 protocol. The web application is not able to leverage the benefits of HTTP/2 when communicating with the client.

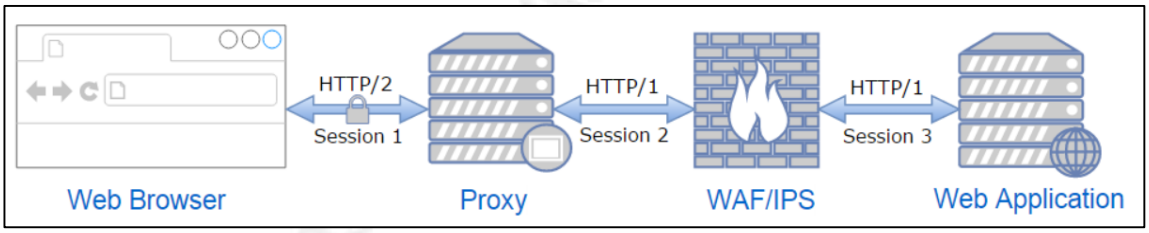

Figure 11: Proxy Protocol Downgrade Architecture

2.3. Encryption Downgrade

In the event that a WAF, IPS, or IDS is capable of inspecting and understanding HTTP/2 communications, they are still faced with the problem of decrypting traffic utilizing PFS cipher suites when they aren't the TLS endpoint. For this method, an HTTP/2 capable proxy can be used as TLS termination point. In this configuration, the proxy can then relay HTTP/2 traffic through an inspecting appliance using a non-PFS enable cipher suite as shown in Figure 12. For instance, a client and proxy can communicate over HTTP/2 using the TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 cipher suite and the proxy can communicate through a WAF/IPS/IDS using the TLS_RSA_WITH_AES_256_GCM_SHA384 cipher suite. However, both the proxy and

web application will have to allow overriding the use of cipher suites blacklisted in RFC 7540.

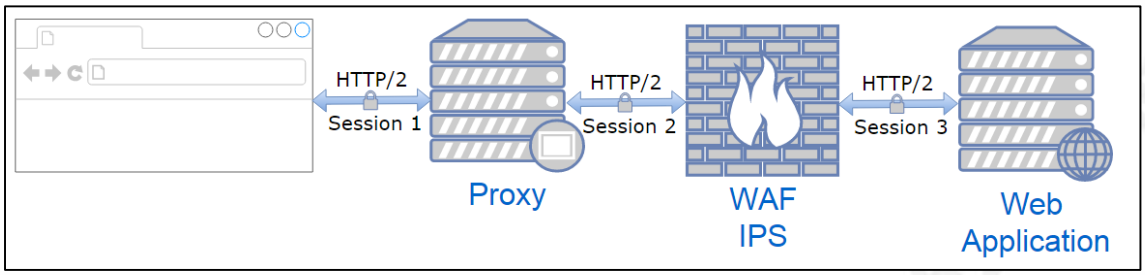

Figure 12: Encryption Downgrade Architecture

2.4. HTTP/2 Clear Text

Many of the previous methods result in circumventing the value of HTTP/2 communications or add processing overhead by decrypting traffic. A different method is to use a proxy to terminate the TLS encrypted HTTP/2 communication channel and then relay the traffic to a web server using the h2c protocol. Figure 13 illustrates this type of architecture. This allows the web application harness the power of HTTP/2 and removes the overhead of decrypting packets for inspection. While the major browser vendors have decided not to support h2c, many web server applications, such as Apache HTTP Server, will support it according to their mod_http2 documentation page.

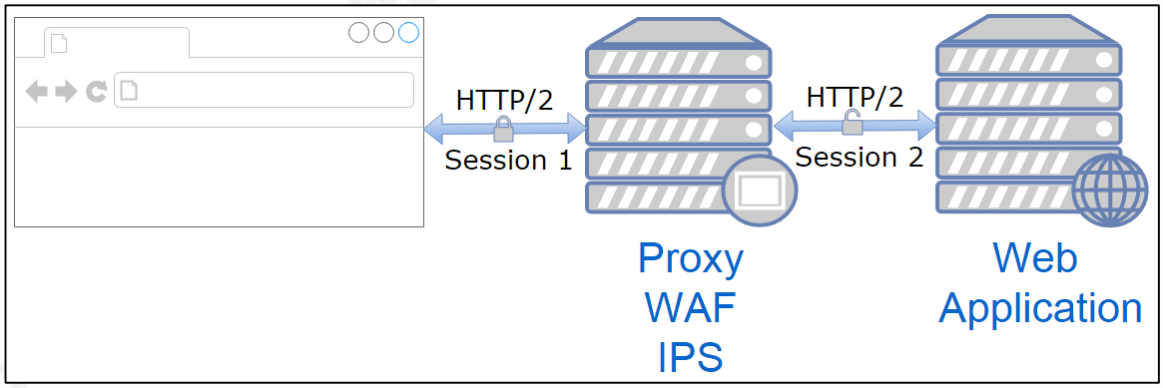

Figure 13: HTTP/2 Protocol Downgrade Architecture

2.5. Layer Seven Inspection

The best and most practical solution is to inspect the traffic at layer seven on the web server. This allows the communication channel between the client and the server to both utilize PFS cipher suites and the HTTP/2 protocol without the hassle of intercepting or decrypting traffic in route. Figure 14 illustrates a client connected to a server employing a PFS enabled cipher suite with HTTP/2. ModSecurity is a web application firewall that

utilizes a web server's application programing interface (API) to inspect traffic at layer 7. When using ModSecurity, traffic is encrypted between the client and the server but is decrypted by the web server as usual for processing. Therefore, ModSecurity never has to worry about the protocol or encryption, making this an ideal solution to detecting web application attacks.

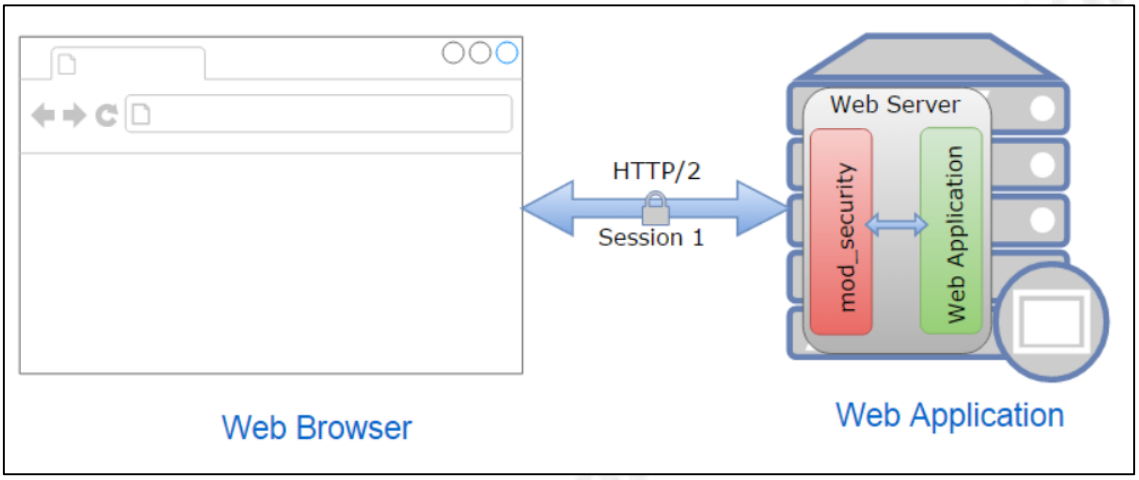

Figure 14: Layer 7 Inspection

This inspection architecture is a valid method for detecting web application attacks taking place over the new HTTP/2 protocol. However, it might be cumbersome or infeasible to install ModSecurity on every web server. This also requires ModSecurity to be supported by the web server. However, this inspection architecture can be used to build a purpose-built WAF that can protect many web applications despite the Web server technology they are using.

3. WAF Setup

The Apache HTTP Server supports HTTP/2 connections through the use of mod_http2 and starting with version 2.4.17. At the time of this writing, the latest version of Ubuntu was 15.10 which utilized Apache version 2.4.12. Additionally, OpenSSL version 1.0.2 is required to use the ALPN extension required for setting up a TLS connection and can be found in Ubuntu version 15.10 and later. This section will walk through the setup of building an HTTP/2 capable WAF using the current Long Term Support (LTS) version of Ubuntu and the latest version of Apache HTTP Server and OpenSSL.

3.1. Building

Download and install the latest LTS version of Ubuntu server (currently 14.04) in your virtual machine application of choice. Do not install Apache during system setup it will need to be built from source. Once Ubuntu Server is install and operational, several applications need to be downloaded and built from as well. The following section contains a header with the application's name, commands to install prerequisite software, titled *Dependencies*, and a section titled *Installation* containing the commands to build and install the application. To ease the installation process, ensure all commands are run from a root terminal prompt. This can be achieved by executing the command `sudo su`.

3.1.1. OpenSSL

As previously identified, support for the TLS ALPN extension wasn't added until version 1.0.2. of OpenSSL. These instructions will download the latest revision of OpenSSL version 1.0.2 from GitHub, configure the application, compile it, and install it into the */opt/openssl-dev* directory.

Dependencies:

```
apt-get update;apt-get install git make gcc -y
```

Installation:

```
cd /opt;git clone -b OpenSSL_1_0_2-stable https://github.com/openssl/openssl.git;cd
openssl;mkdir /opt/openssl-dev;./config --openssldir=/opt/openssl-dev/;make;make
test;make install
```

3.1.2 libevent

Libevent is an event notification library that is utilized within the nghttp2 application. The following instructions will download libevent from GitHub, configure the application, compile it, and install it into the /usr/lib/x86_64-linux-gnu directory.

Dependencies:

```
apt-get install autoconf libtool -y
```

***Installation*:**

```
cd /opt;git clone https://github.com/libevent/libevent.git;cd
libevent;./autogen.sh;mkdir /opt/libevent-dev;./configure --prefix=/usr/lib/x86_64-
linux-gnu;make;make verify;make install
```

3.1.3. nghttp2

Nghttp2 is a HTTP/2 library written in C and is utilized within Apache's mod_http2. Additionally, this application provides a standalone HTTP/2 capable client, server, and proxy. The following instructions will download nghttp2 from GitHub, configure the application, compile it, and install it into the /usr/local/bin directory.

Dependencies:
```
apt-get install g++ binutils libtool pkg-config zlib1g-dev libcunit1-dev libssl-dev
libxml2-dev libev-dev libjansson-dev libjemalloc-dev cython python3-dev python-
setuptools -y
```

Installation:
```
cd /opt;git clone https://github.com/tatsuhiro-t/nghttp2.git;cd nghttp2;autoreconf -
i;automake;autoconf;./configure PKG_CONFIG_PATH=/opt/openssl-dev/lib/pkgconfig LIBS=-
ldl;make check;make install
```

3.1.4. libcurl

The libcurl library is used within ModSecurity and is the library used to create the popular command line program called curl. When installed after nghttp2, curl will support HTTP/2 connections. The following instructions will download libcurl from GitHub, configure the application, compile it, and install it into the /usr/local/bin directory.

Installation:
```
cd /opt;git clone https://github.com/curl/curl.git;cd curl;./buildconf;./configure --
with-ssl=/opt/openssl-dev/bin;make;make test;make install
```

3.1.5. Apache HTTP Server

This is the web server will be configured to act as a WAF for HTTP/2 traffic. The following commands will download the latest version of Apache HTTP Server, configure the application, and install it into the /opt/apache2 directory.

Dependencies:
```
apt-get install libapr1-dev libaprutil1-dev -y
```

Installation:
```
cd /opt;wget http://www.gtlib.gatech.edu/pub/apache//httpd/httpd-2.4.18.tar.bz2;bunzip2
httpd-2.4.18.tar.bz2;tar -xf httpd-2.4.18.tar;cd httpd-2.4.18;mkdir
/opt/apache2;./configure --enable-http2 --enable-proxy --enable-proxy-http --enable-ssl
--prefix=/opt/apache2 --with-ssl=/opt/openssl-dev --enable-ssl-staticlib-deps --enable-
mods-static=ssl;make;make install
```

3.1.6. ModSecurity

ModSecurity is an open source web application firewall that works with Apache. The following commands will download the latest version of ModSecurity, configure the application, compile the mod_security module used with Apache, and install some configuration files onto the host.

Installation:
```
cd /opt;git clone git://github.com/SpiderLabs/ModSecurity.git;cd
ModSecurity;./autogen.sh;./configure --with-apxs=/opt/apache2/bin/apxs;make;make
install;cp /usr/local/modsecurity/lib/mod_security2.so /opt/apache2/modules/;mkdir
```

```
/etc/modsecurity;cp /opt/ModSecurity/modsecurity.conf-recommended
/etc/modsecurity/modsecurity.conf;cp unicode.mapping /etc/modsecurity/
```

3.1.7. OWASP Core Rule Set

The Open Web Application Security Project (OWASP) has prepared a Core Rule Set (CRS) that provides a foundational set of rules that are used with ModSecurity to detect and block malicious traffic. The following commands will download the latest version of OWASP CRS and link them with the current install of Apache.

Installation:

```
mkdir /opt/apache2/conf/crs;cd /opt;git clone https://github.com/SpiderLabs/owasp-
modsecurity-crs.git;cd owasp-modsecurity-crs;cp modsecurity_crs_10_setup.conf.example
/opt/apache2/conf/crs/modsecurity_crs_10_setup.conf;mkdir
/opt/apache2/conf/crs/activated_rules;ln -s
/opt/apache2/conf/crs/modsecurity_crs_10_setup.conf
/opt/apache2/conf/crs/activated_rules/;for r in $(ls /opt/owasp-modsecurity-
crs/base_rules/);do ln -s /opt/owasp-modsecurity-crs/base_rules/$r
/opt/apache2/conf/crs/activated_rules/$r;done; for r in $(ls /opt/owasp-modsecurity-
crs/optional_rules/);do ln -s /opt/owasp-modsecurity-crs/optional_rules/$r
/opt/apache2/conf/crs/activated_rules/$r;done
```

3.1.8. Configuration

To properly utilize the new Apache install and all of the other associated components, they must be configured. Generate an X.509 certificate so that Apache can establish TLS connections. Execute the following commands to create a new certificate:

```
/opt/openssl-dev/bin/openssl req -x509 -nodes -days 365 -newkey rsa:4096 -keyout
/opt/apache2/conf/server.key -out /opt/apache2/conf/server.crt
```

Next, Apache's configuration must be edited to enable some features, disable unused features, and configure used modules. Open the file at */opt/apache2/conf/httpd.conf* and make the following changes:

Uncomment these lines by removing the '#':

```
#Include conf/extra/httpd-ssl.conf
#LoadModule http2_module modules/mod_http2.so
#LoadModule socache_shmcb_module modules/mod_socache_shmcb.so
#LoadModule slotmem_shm_module modules/mod_slotmem_shm.so
#LoadModule unique_id_module modules/mod_unique_id.so
```

Comment out this line by adding a '#':

```
LoadModule lbmethod_heartbeat_module modules/mod_lbmethod_heartbeat.so
```

Add the following information to the end of the configuration file:

```
LoadModule security2_module modules/mod_security2.so
```

```
<IfModule security2_module>
        Include /etc/modsecurity/modsecurity.conf
        Include conf/crs/activated_rules/*.conf
</IfModule>
```

3.1.9. TLS Cipher Suites

In Figure 5 we identified the TLS cipher suites supported by the top web browsers. During the TLS Client Hello message, many of the browsers will attempt to support TLS cipher suites that utilize SHA1 hashing such as TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014). However, the HTTP/2 RFC has black listed these cipher suites. To prevent the use of TLS cipher suites that support SHA1 and cause the connection to fall over back to HTTP/1.1, support for them must be disabled. Additionally, Apache will be configured to only support TLS1.2 because it is the only version that supports HTTP/2. Edit the file at */opt/apache2/conf/extra/httpd-ssl.conf* and make the following adjustments:

Change:

```
SSLProtocol all -SSLv3
SSLCipherSuite HIGH:MEDIUM:!MD5:!RC4
SSLProxyCipherSuite HIGH:MEDIUM:!MD5:!RC4
```

To:

```
SSLProtocol TLSv1.2
SSLCipherSuite HIGH:MEDIUM:!MD5:!RC4:!SHA1
SSLProxyCipherSuite HIGH:MEDIUM:!MD5:!RC4:!SHA1
```

Add the following line to the file just before the </VirtualHost> tag so that the web server will choose HTTP/2 connections over HTTP/1.1 connections: `Protocols h2 http/1.1`.

Edit the file at */etc/modsecurity/modsecurity.conf* and change `SecRuleEngine DetectionOnly` to `SecRuleEngine On`. ModSecurity is now enabled to block attacks instead of just detecting them. Also modify the line `SecAuditLog /var/log/modsec_audit.log` to `SecAuditLog /opt/apache2/logs/modsec_audit.log`.

3.2 Test Web Application

At this point, we have a WAF but no web application to protect. To complete the test environment, we're going to setup a vulnerable web application to protect. For this test environment, we're going to use OWASP Mutillidae II, which is described as "…a

free, open source, deliberately vulnerable web-application providing a target for web-security enthusiest." (Druin, n.d.).

Nginx added support for HTTP/2 in version 1.9.5 (Memon, 2015). To use the TLS ALPN extension with HTTP/2 connections, Nginx must be built from source using OpenSSL version 1.0.2. The vulnerable Mutillidae web application can be accessed over HTTP/2. Download and install the latest LTS version of Ubuntu server in a second virtual machine. During the install, leave the MySQL password blank for this install as the application is intended to be vulnerable and allows for easier configuration. The following instructions will cover installing Nginx with Mutillidae on a separate host.

Dependencies:

```
apt-get update;apt-get install git make gcc mysql-server php5-mysql php5-fpm php-pear
php5-gd php5-mcrypt php5-curl libpcre3-dev zlib1g-dev -y
```

Installation:

```
cd /opt;git clone -b OpenSSL_1_0_2-stable https://github.com/openssl/openssl.git;cd
/opt;wget http://nginx.org/download/nginx-1.9.5.tar.gz;tar -xzvf nginx-1.9.5.tar.gz;cd
nginx-1.9.5;mkdir /opt/nginx;./configure --prefix=/opt/nginx --with-http_ssl_module --
with-openssl=/opt/openssl --with-http_v2_module --user=www-data;make;make
install;mysql_install_db;cd /opt/nginx/html;git clone
git://git.code.sf.net/p/mutillidae/git mutillidae;openssl req -x509 -nodes -days 365 -
newkey rsa:4096 -keyout /opt/nginx/conf/cert.key -out /opt/nginx/conf/cert.pem
```

Configuration:

Edit the file at */opt/nginx/conf/nginx.conf* and add the following to the end of the file before the last right curly brace:

```
# HTTPS server
server {
       listen       443 ssl http2;
       ssl_certificate      cert.pem;
       ssl_certificate_key  cert.key;
       ssl_protocols TLSv1.2;
       ssl_ciphers  HIGH:!aNULL:!MD5:!SHA1;
       ssl_prefer_server_ciphers  on;
       root /opt/nginx/html;
       index index.php index.html index.htm;

       location ~ \.php$ {
              try_files $uri =404;
              fastcgi_split_path_info ^(.+\.php)(/.+)$;
              fastcgi_pass unix:/var/run/php5-fpm.sock;
              fastcgi_index index.php;
              fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
              include fastcgi_params;
       }
}
```

Start the web server by executing `/opt/nginx/sbin/nginx` .To finish the installation, go to the page at *https://<ip address>/mutillidae* and click the "setup/reset the DB" link. Now you have a fully functional vulnerable web application that can be accessed over HTTP/2.

3.3. Internal Proxy

At this point, the WAF is setup and will establish HTTP/2 connections. However, the WAF is not configured to protect a particular web application. The current WAF setup can be utilized as a reverse proxy to protect multiple web applications. Apache has several proxy directives that are used to enable reverse proxy functionality. Apache's proxy module does not yet support HTTP/2 connections. A proxy that does support HTTP/2 connections is in development (Eissing, 2016). Therefore, the reverse connection between the WAF and the web application will take place over HTTP/1.1. However, this prevents current web applications that support HTTP/2 connections from using the new protocol. The previously compiled nghttp2 library comes with a proxy that is capable of handling HTTP/2 connections. We will configure the WAF to use the nghttpx proxy internally that will in-turn proxy the connections to the Mutillidae web application over HTTP/2. Figure 15 illustrates this configuration.

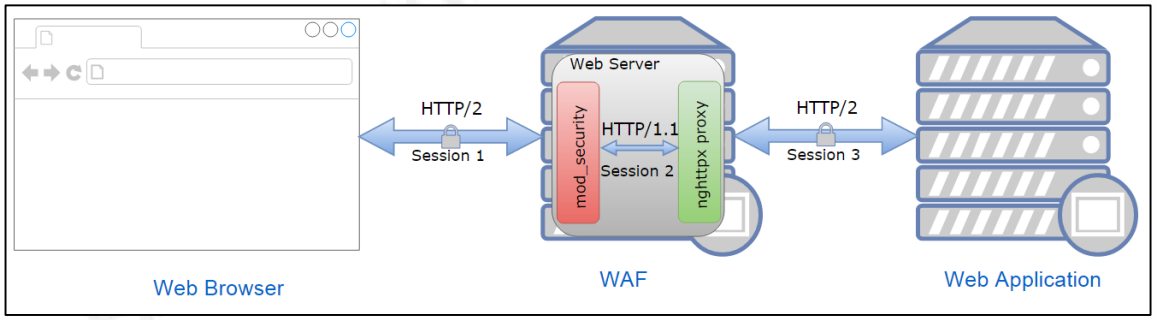

Figure 15: HTTP/2 WAF

This configuration is not as efficient as possible because the connection is dropped down to HTTP/1.1 between the WAF and nghttpx proxy. However, this configuration offers the most flexibility in allowing both a client and web application to support HTTP/2 connections. Additionally, this configuration is the easiest to use at it will require little reconfiguration to support future expansion when built-in HTTP/2 proxy support becomes available.

Configure the WAF to proxy connections internally to the nghttpx proxy that in-turn will proxy to the Mutillidae web application. The connection between the WAF and nghttpx is made by utilizing Apache's proxy directives. These directives can be added to the main configuration file or to a specific virtual host. Open the file at */opt/apache2/confs/extra/httpd-ssl.conf* and add the following directives within the <*VirtualHost _default_:443*> element to enable proxy support:

```
SSLProxyEngine On
SSLProxyCheckPeerName Off
SSLProxyCheckPeerCN Off
ProxyPass / http://127.0.0.1:3000/
ProxyPassReverse / http://127.0.0.1:3000/
```

This configuration will proxy connections internally to the nghttpx proxy listening on port 3000 over HTTP/1.1 without SSL/TLS encryption. Now we need to run nghttpx so that it can accept the connections from the WAF and relay them to back-end Mutillidae web application. Execute the following command, replacing <ip address> with the address of the Mutillidae web application, to enable the nghttpx proxy:

```
nghttpx --frontend=127.0.0.1,3000 --frontend-no-tls --backend='<ip
address>,443;/;proto=h2' --backend-tls -k --errorlog-file=/opt/apache2/logs/nghttpx.log
-L INFO --host-rewrite -D
```

3.4. SQL Injection Attack

To verify the WAF is working correctly, navigate to page *https://<apache_proxy_ip>/mutillidae/index.php?page=login.php* and attempt to login with a username and password of `admin`. The web application will respond with a message that the password is incorrect. Next, try logging in with a username and password of `' or 1=1;--` and notice that web application responds with HTTP 403 Forbidden message. The WAF successfully blocked the SQL injection attempt against the login page (see Figure 16). The attack can be validated by viewing the file at */opt/apache2/logs/modsec_audit.log* as shown below in Figure 17.

Figure 16: Blocked SQL Injection Attack

```
Message: Access denied with code 403 (phase 2). Pattern match "\\W{4,}"
_40_generic_attacks.conf"] [line "37"] [id "960024"] [rev "2"] [msg "Met
Matched Data: ; --  found within ARGS:username: ' or 1=1; -- "] [ver "OW
Apache-Error: [file "apache2_util.c"] [line 271] [level 3] [client %s] M
Action: Intercepted (phase 2)
Apache-Handler: proxy-server
Stopwatch: 1457805061317680 4453 (- - -)
Stopwatch2: 1457805061317680 4453; combined=1346, p1=378, p2=432, p3=0,
Response-Body-Transformed: Dechunked
Producer: ModSecurity for Apache/2.9.1-RC1 (http://www.modsecurity.org/)
Server: Apache/2.4.18 (Unix) OpenSSL/1.0.2h-dev
WebApp-Info: "default" "p7l8lik2i3p5hjq5jtreuuct84" ""
Engine-Mode: "ENABLED"

--61b2d77c-Z--
```

Figure 17: modsec_audit.log

Using the architecture outlined in section 2.5 (Layer 7 Inspection), it was possible to detect and block an SQL injection attack against a web application that took place over the HTTP/2 protocol. This architecture is valuable because current technologies are not able to inspect or decrypt HTTP/2 traffic and the attack would have been successful against a web application communicating over HTTP/2.

4. Evasion

The previous section focused on protecting a web application from attacks that take place over the HTTP/2 protocol. While determining the best method of protecting a web application, it became evident that current IDS and IPS technologies are not capable of decrypting nor analyzing HTTP/2. This set of conditions enables HTTP/2 as an ideal protocol to evade detection by an IDS and potentially an IPS. This type of evasion is likely to occur on traffic leaving a network for such things as exfiltration, command and control, or bot traffic. While not evaluated, a network utilizing a forward proxy may prevent HTTP/2 traffic from egressing the network. It will depend on the applications reaction when the decrypted traffic isn't using the protocol it expected.

The use of PFS enable cipher suites in combination with the lack of HTTP/2 protocol dissectors precludes the ability to identify malicious HTTP/2 egressing a network. However, the information covered in Section 1 can be leveraged to, at a minimum, detect the presence of HTTP/2 traffic. A connection utilizing the h2 variant of HTTP/2 must be setup during the TLS sessions setup through the ALPN extension. While a client sends the server a list of supported protocols, the server makes the final decision on which protocol to communicate over. The selected protocol is announced to the client in the Server Hello message. Therefore, IDS technologies such as Snort, Suricata, or Bro can be configured to inspect Server Hello messages for the existence of the ALPN extension and the selection of the h2 protocol.

Not all HTTP/2 connections must take place over a TLS encrypted communication channel. The h2c variant of HTTP/2 does not require TLS encryption, making its communications clear-text. This variant can be detected a number of ways. One way is to evaluate for the presence of the Upgrade HTTP header with a value of h2c in either a server or client message. Alternatively, all HTTP/2 connections using the h2c variant must send the magic string of `PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n`. A signature can be created in any detection application to identify the magic string.

5. Conclusion

HTTP/2 is a new protocol that enhances web communications by overcoming efficiency problems found in HTTP/1. This new protocol requires the use of TLS version 1.2 and PFS enabled cipher suites when using a secure h2 connection. Because the protocol is new, most applications are not equipped with protocol dissectors that understand HTTP/2. Furthermore, because RFC 7540 requires the use of PFS enabled cipher suites, packets can't be decrypted for inspection while in transit. This combination of features provides a challenge if inspecting traffic to prevent attacks against web applications.

There are many potential architectures available for inspecting traffic and detecting attacks, but most are not feasible. In order to overcome this obstacle, a WAF can be built using open source software. The combination of Apache HTTP Server, ModSecurity, and the nghttp2 library can be leveraged to facilitate HTTP/2 connections

between a client and web application all while providing an inspection point at layer 7 to identify and block malicious traffic.

This HTTP/2 capable WAF is a viable solution to protect web applications that communicate over HTTP/2 from attacks. However, it does not facilitate the detection of HTTP/2 when being used as an evasion technique. The existence of HTTP/2 traffic can be determined by inspecting a Server Hello message for a value of h2 in the ALPN TLS extension. Additionally, the existence of the magic string used to establish an h2c connection can be detected to identify HTTP/2 traffic. While not discussed, Apache HTTP Server, Nginx, or nghttpx could be leveraged as a forward proxy for traffic leaving a network. The HTTP/2 protocol might be new and might leverage hard to inspect encryption, but attacks over it can still be detected and blocked using a combination of publically available software.

References

Apache Module mod_http2. (n.d.). Retrieved March 1, 2016, from https://httpd.apache.org/docs/2.4/mod/mod_http2.html

Belsche, M., Peon, R., & Thomson, M. (2015, May). Request for comments 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2). Retrieved October 7, 2015, from https://www.rfc-editor.org/rfc/rfc7540.txt

Bernat, V. (2011, November 28). SSL/TLS & Perfect Forward Secrecy. Retrieved March 15, 2016, from http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html

Berners-Lee, T. (n.d.). The Original HTTP as defined in 1991. Retrieved February 3, 2016, from https://www.w3.org/Protocols/HTTP/AsImplemented.html

Browser Statistics. (n.d.). Retrieved January 13, 2016, from http://www.w3schools.com/browsers/browsers_stats.asp

Dierks, T. and E. Rescorla (2008, August). Request for comments 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. Retrieved October 7, 2015, from https://www.rfc-editor.org/rfc/rfc5246.txt

Druin, J. (n.d.). OWASP Mutillidae II. Retrieved March 1, 2016, from https://sourceforge.net/projects/mutillidae/

Eissing, S. (2016, February 08). Mod_proxy_http2 [Checked in a very experimental mod_proxy_http2]. Retrieved from http://marc.info/?l=apache-httpd-dev&m=145495124513617&w=2

Friedl, S., Popov, A., Langley, A., & Stephan, E. (2014, July). Request for Comments 7301 - Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. Retrieved April 2, 2016 from http://tools.ietf.org/html/rfc7301

Helme, S. (2014, May 10). Perfect Forward Secrecy - An Introduction. Retrieved April 04, 2016, from https://scotthelme.co.uk/perfect-forward-secrecy/

Memon, F. (2015, September 22). HTTP/2 Supported with NGINX Open Source 1.9.5 | NGINX. Retrieved March 15, 2016, from https://www.nginx.com/blog/nginx-1-9-5/

OpenSSL 1.0.2 Series Release Notes. (n.d.). Retrieved October 20, 2015, from https://www.openssl.org/news/openssl-1.0.2-notes.html

Shaver, J. (2015, February 11). Decrypting TLS Browser Traffic With Wireshark – The Easy Way! Retrieved April 04, 2016, from https://jimshaver.net/2015/02/11/decrypting-tls-browser-traffic-with-wireshark-the-easy-way/

Stenberg, D. (2015, May 21). Http2 explained [PDF]. Retrieved December 15, 2015, from https://daniel.haxx.se/http2/http2-v1.12.pdf

Tsujikawa, T. (n.d.). Nghttpx(1). Retrieved January 15, 2016, from https://www.nghttp2.org/documentation/nghttpx.1.html

Winkel, S. (2015, December 27). Network Forensics and HTTP/2. Retrieved March 12, 2016 from https://www.sans.org/reading-room/whitepapers/forensics/network-forensics-http-2-36647

Appendix A – Apache httpd.conf Without Comments

```
ServerRoot "/opt/apache2"
Listen 80
LoadModule authn_file_module modules/mod_authn_file.so
LoadModule authn_core_module modules/mod_authn_core.so
LoadModule authz_host_module modules/mod_authz_host.so
LoadModule authz_groupfile_module modules/mod_authz_groupfile.so
LoadModule authz_user_module modules/mod_authz_user.so
LoadModule authz_core_module modules/mod_authz_core.so
LoadModule access_compat_module modules/mod_access_compat.so
LoadModule auth_basic_module modules/mod_auth_basic.so
LoadModule socache_shmcb_module modules/mod_socache_shmcb.so
LoadModule reqtimeout_module modules/mod_reqtimeout.so
LoadModule filter_module modules/mod_filter.so
LoadModule mime_module modules/mod_mime.so
LoadModule http2_module modules/mod_http2.so
LoadModule log_config_module modules/mod_log_config.so
LoadModule env_module modules/mod_env.so
LoadModule headers_module modules/mod_headers.so
LoadModule unique_id_module modules/mod_unique_id.so
LoadModule setenvif_module modules/mod_setenvif.so
LoadModule version_module modules/mod_version.so
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_connect_module modules/mod_proxy_connect.so
LoadModule proxy_ftp_module modules/mod_proxy_ftp.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule proxy_fcgi_module modules/mod_proxy_fcgi.so
LoadModule proxy_scgi_module modules/mod_proxy_scgi.so
LoadModule proxy_wstunnel_module modules/mod_proxy_wstunnel.so
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_express_module modules/mod_proxy_express.so
LoadModule slotmem_shm_module modules/mod_slotmem_shm.so
LoadModule lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
LoadModule lbmethod_bytraffic_module modules/mod_lbmethod_bytraffic.so
LoadModule lbmethod_bybusyness_module modules/mod_lbmethod_bybusyness.so
LoadModule unixd_module modules/mod_unixd.so
LoadModule status_module modules/mod_status.so
LoadModule autoindex_module modules/mod_autoindex.so
LoadModule dir_module modules/mod_dir.so
LoadModule alias_module modules/mod_alias.so
LoadModule security2_module modules/mod_security2.so
<IfModule unixd_module>
      User daemon
      Group daemon
</IfModule>
ServerAdmin you@example.com
<Directory />
    AllowOverride none
    Require all denied
</Directory>
DocumentRoot "/opt/apache2/htdocs"
<Directory "/opt/apache2/htdocs">
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>
<IfModule dir_module>
    DirectoryIndex index.html
```

```
</IfModule>
<Files ".ht*">
    Require all denied
</Files>
ErrorLog "logs/error_log"
LogLevel warn
<IfModule log_config_module>
    LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined
    LogFormat "%h %l %u %t \"%r\" %>s %b" common
    <IfModule logio_module>
      LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\" %I %O"
       combinedio
    </IfModule>
    CustomLog "logs/access_log" common
</IfModule>
<IfModule alias_module>
    ScriptAlias /cgi-bin/ "/opt/apache2/cgi-bin/"
</IfModule>
<IfModule cgid_module>
</IfModule>
<Directory "/opt/apache2/cgi-bin">
    AllowOverride None
    Options None
    Require all granted
</Directory>

<IfModule mime_module>
    TypesConfig conf/mime.types
    AddType application/x-compress .Z
    AddType application/x-gzip .gz .tgz
</IfModule>
<IfModule proxy_html_module>
       Include conf/extra/proxy-html.conf
</IfModule>
<IfModule ssl_module>
       SSLRandomSeed startup builtin
       SSLRandomSeed connect builtin
</IfModule>
       Include /opt/apache2/conf/extra/httpd-ssl.conf
<IfModule http2_module>
        LogLevel http2:info
</IfModule>
<IfModule security2_module>
       Include /etc/modsecurity/modsecurity.conf
       Include conf/crs/activated_rules/*.conf
</IfModule>
Protocols h2 h2c http/1.1
<IfModule proxy_module>
       LogLevel proxy:debug
</IfModule>
```

Appendix B – Apache httpd-ssl.conf Without Comments

```
Listen 443
SSLCipherSuite HIGH:MEDIUM:!MD5:!RC4:!SHA1
SSLProxyCipherSuite HIGH:MEDIUM:!MD5:!RC4:!SHA1
SSLHonorCipherOrder on
SSLProtocol TLSv1.2
SSLProxyProtocol TLSv1.2
SSLPassPhraseDialog  builtin
SSLSessionCache        "shmcb:/opt/apache2/logs/ssl_scache(512000)"
SSLSessionCacheTimeout  300
<VirtualHost _default_:443>
       DocumentRoot "/opt/apache2/htdocs"
       ServerName www.example.com:443
       ServerAdmin you@example.com
       ErrorLog "/opt/apache2/logs/error_log"
       TransferLog "/opt/apache2/logs/access_log"
       SSLEngine on
       SSLCertificateFile "/opt/apache2/conf/server.crt"
       SSLCertificateKeyFile "/opt/apache2/conf/server.key"
       <FilesMatch "\.(cgi|shtml|phtml|php)$">
           SSLOptions +StdEnvVars
       </FilesMatch>
       <Directory "/opt/apache2/cgi-bin">
           SSLOptions +StdEnvVars
       </Directory>
       BrowserMatch "MSIE [2-5]" \
                nokeepalive ssl-unclean-shutdown \
                downgrade-1.0 force-response-1.0
       CustomLog "/opt/apache2/logs/ssl_request_log" \
                 "%t %h %{SSL_PROTOCOL}x %{SSL_CIPHER}x \"%r\" %b"
       Protocols h2
       SSLProxyEngine On
       SSLProxyCheckPeerName Off
       SSLProxyCheckPeerCN Off
       ProxyPreserveHost Off
       ProxyPass / http://127.0.0.1:3000/
       ProxyPassReverse / http://127.0.0.1:3000/
</VirtualHost>
```

Appendix C – Nginx nginx.conf Without Comments

```
worker_processes  1;
events {
    worker_connections  1024;
}
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    server {
        listen       80;
        server_name  localhost;
        location / {
            root   html;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
       }
       server {
              listen       443 ssl http2;
              ssl_certificate      cert.pem;
              ssl_certificate_key  cert.key;
              ssl_protocols TLSv1.2;
              ssl_ciphers  HIGH:!aNULL:!MD5:!SHA1;
              ssl_prefer_server_ciphers  on;
              root /opt/nginx/html;
              index index.php index.html index.htm;
              location ~ \.php$ {
                     try_files $uri =404;
                     fastcgi_split_path_info ^(.+\.php)(/.+)$;
                     fastcgi_pass unix:/var/run/php5-fpm.sock;
                     fastcgi_index index.php;
                     fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
                     include fastcgi_params;
              }
       }
}
```