Creating A Secure Channel In The Web Application Layer

CS5321: Network Security – Project Report

Akshay Viswanathan, Khairul Rizqi B Mohd Shariff, Lee Ming Xuan, Thng Kai Yuan
School of Computing, National University of Singapore

***Abstract*—With increasing number of content owners/organizations utilizing the services of Content Delivery Networks (CDNs) to serve and consume critical, personal as well as potentially confidential data, security of the data served is surprisingly not as widely discussed as expected. Should a CDN or its corresponding edge server be compromised, malicious or unintended content could potentially be served to clients and confidential data could be intercepted and misused despite the presence of a transport-layer security. In our project, we explore and discuss weaknesses in a typical setup using CDN before proposing a solution that aims to ensure the end-to-end integrity, confidentiality and authenticity of data being served to and from services leveraging the functionality of CDNs.**

Keywords—secure channel; content delivery network; confidentiality; integrity; authenticity

I. Introduction

A. Content Delivery Network

Content Delivery Network (CDN) is an umbrella term used to describe a wide variety of types of services. Also commonly known as Content Distribution Network, CDN service provides high availability and performance by distributing content to servers deployed in multiple data centers around the world [1]. Content owners leverage on their distributed network to efficiently serve their contents as well as leverage other security related features offered by CDNs. CDNs offer a large fraction of the Internet content today including static web objects, files and applications.

B. Advantages of Using a Content Delivery Network

So why are CDNs so popular? First and foremost, they offer speed [2]. CDNs behave as a facade between the client and the origin server, intercepting and decrypting requests and either serving the responses themselves, or forwarding the requests to the origin servers. They utilize extensive caching, load-balancing and optimized routing to deliver content as quickly as possible to clients.

In addition to speed, CDNs also offer protection [2]. Most CDNs have sufficient resources to absorb traffic originating from denial-of-service attacks and today, many CDNs also provide security services such as web application firewalls, email address obfuscation and hotlink protection for free. These added securities often provide a further incentive for companies to host their content with the providers.

C. Problem Description

With increasing number of content owners/organizations utilizing the services of CDNs to serve and consume critical, personal as well as potentially confidential data, security of the data transmitted is surprisingly not as widely discussed as expected. In our investigation, we realised that in a typical CDN setup, the HTTPS layer that we assume to be safe is no longer so because of men-in-the-middle between the origin server and its end users. These men-in-the-middle are the reverse proxy nodes of the CDN which relays web content between the origin server and its end users [3]. In a typical mode of operation, information arriving at the reverse proxy node is decrypted and re-encrypted before retransmission to either the end user or the origin server. As a result, a rogue CDN node could potentially compromise the security properties of the HTTPS layer such as confidentiality, integrity and authenticity.

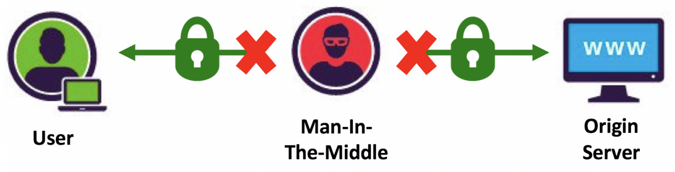


Fig. 1. A CDN edge node as the man-in-the-middle

As an example, the recent Cloudbleed incident [4] underscored how data confidentiality can be breached by a CDN. In that incident, a vulnerability in the parsing engine on Cloudflare's edge servers was discovered that could potentially have been exploited to leak confidential information (e.g. cookies, personally identifiable information) residing in the memory of the servers.

Aside from confidentiality, it is also possible for the men-in-the-middle to tamper with client requests and/or server responses. This could lead to false information being transmitted as the origin entity and result in dire consequences. An example would be malicious content being served in place of legitimate static content. For instance, jQuery (which is widely consumed through CDNs) could be modified to perform unexpected actions on the client's browser.

With the nature of the internet being such that clients and consumers of CDNs utilize their service, there is no compulsion that they should completely trust the services provided by the CDN, and considering the scenarios described above, should utilize other methods to protect both their data and their clients from malicious actions, should the integrity of a CDN provider be in doubt. Therefore, our team proposes the creation of an application-layer security within HTTPS itself named Secure Channel.

II. Our Solution

A. Overview

Secure Channel is an application-layer protocol designed to ensure the end-to-end security of communications between origin servers and clients going through reverse proxy nodes of a CDN. The relevance of such a secure channel, as emphasised before, is underscored by the recent Cloudbleed [4] incident in which sensitive data (e.g. cookies, personally identifiable information) was compromised due to vulnerable CDN nodes. Secure Channel's application-layer encryption would have prevented an attacker from exploiting any leaked data. Secure Channel can also ensure the authenticity and integrity of data transmitted between an origin server and its client.

The Secure Channel setup comprises three main components:

1. A highly available, potentially a replicated cluster, public key store
2. Client browsers and extension supporting the Secure Channel protocol
3. A middleware installed on the origin server to produce responses complying with the Secure Channel protocol

Below is a target state diagram of the setup described above:

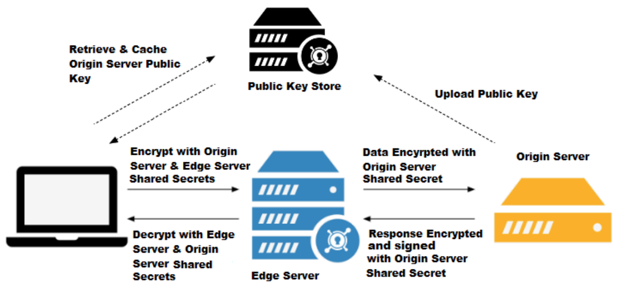

Retrieve & Cache Origin Server Public Key
Public Key Store
Upload Public Key
Encrypt with Origin Server & Edge Server Shared Secrets
Data Encyrpted with Origin Server Shared Secret
Origin Server
Decrypt with Edge Server & Origin Server Shared Secrets
Edge Server
Response Encrypted and signed with Origin Server Shared Secret

Fig. 2. Our target state diagram

Prior to the use of the Secure Channel protocol, an owner or administrator of an origin server must first enrol their domain into this programme by uploading their public key to the public key store. Due diligence will be performed to determine the authenticity and validity of such requests. From then on, all data transmitted between that domain and its clients adopting our solution will be protected by our Secure Channel protocol.

B. A Typical Request-Response Lifecycle

When a request is made from a secured client browser, the browser would first check whether the requested domain is enrolled in our programme. This check is first done against the browser cache. If the browser cache contains no information about the enrolment of the requested domain, a query will then be made against our public key store to check the enrolment status of the requested domain and if applicable, retrieve the public key of the requested domain.

If the domain is enrolled in our programme, the browser would then cache the domain's public key and continue to use it in the Secure Channel protocol. In the first step of the protocol, a random symmetric key is generated by the client browser. This symmetric key is then used to encrypt the payload which contains the original request and an unkeyed hash of the original request. The symmetric key is then encrypted using the origin server's public key and together with the encrypted payload, sent to the origin server via the CDN's edge server.

We reason that the CDN's edge server would not be able to read the original request as it is encrypted using a random symmetric key known only to the client and the origin server. It would also not be able to tamper with the request without risking detection because that would cause the integrity check to fail at the origin server. We note that the authenticity of the client can be verified at the application layer, for instance through the use of cookies. Therefore, in performing the steps mentioned above, the confidentiality, integrity and authenticity of the request is ensured.

Once the request reaches the origin server, the Secure Channel middleware installed on the server would retrieve the client-generated symmetric key using its own private key. The symmetric key is then used to decrypt the payload and the integrity of the payload is checked before the original request is passed to the web application. After a response is formulated, the response is encrypted and signed (keyed-hash) using the client-generated symmetric key before it is sent back to the client via the CDN's edge server.

Again, the CDN's edge server would not be able to read the original response and it is encrypted using the random symmetric key known only to the client and the origin server. It would also not be able to tamper with the response without risking detection because that would cause the integrity check to fail in the client's browser. The edge server would also not be able to forge a response because such a forgery would then fail the keyed-hash verification in the client's browser, assuming that the symmetric key is still unknown to the edge server. Hence, the confidentiality, integrity and authenticity of the response is also ensured.

Finally, when the response is received by the client's browser, the keyed-hash is checked to verify the authenticity and integrity of the response before the original response is decrypted and loaded by the browser. The results of the authenticity and integrity check is conveyed to the Secure Channel browser extension through a header field which then decides whether to block the resource. If any of the checks fail, the resource would be blocked. Otherwise, the resource would be rendered by the browser and the user can be sure of the end-to-end confidentiality, integrity and authenticity of his communication with the website.

C. Implementation Details

To demonstrate our protocol, we have had to implement many components ourselves. These components include:

- A modified Chromium browser
- A Chromium browser extension
- A public key server

- A middleware that implements the Secure Channel protocol on the demo web server
- An origin server with a demo website developed using Django

In the following paragraphs, we will use the context of a POST request to our demo website https://untampered.info to illustrate the interactions between our different components and the cryptographic primitives that goes into our Secure Channel implementation.

1) *Sending the request (browser)*

a) Enrolment check

When a request is made, the browser first checks whether the requested domain (untampered.info) is enrolled in our programme. This check is first done against the browser cache. If the browser cache does not contain any information on the enrolment of the requested domain, the browser would then make a HTTPS GET request to our public key store to obtain enrolment information of the requested domain. Eventually, if the requested domain is enrolled in our programme (as is the case for our domain untampered.info), the browser would have obtained a copy of the domain's public key either from the browser cache or from the public key store.

b) Generating a cryptographically random symmetric key

Since a secured request is to be made, the browser then proceeds to generate a cryptographically random 256-bits symmetric key (*K*) and binds it to the transaction.

c) Encrypting and encoding the request header

The generated symmetric key is then used to encrypt the plaintext `1 || SHA256(request_header) || request_header` with AES-256-CBC where `||` is the concatenation operator. The ciphertext, along with its IV (`Enc_Hdr = IV || AES_256_CBC(1 || SHA256(request_header) || request_header, K)`) is then encoded in Base64 resulting in the final form `base64_encode(Enc_Hdr)`.

d) Encrypting and encoding the symmetric key

To ensure the authenticity of the recipient, the symmetric key (*K*) is then encrypted using the origin server's public key (`Enc_Key = RSA_PKCS1(K, public_key)`) and later encoded in Base64.

e) Generating a new request header

To protect the confidentiality of even the request path itself, a new request header preserving only the original request method (e.g. POST) and HTTP version (e.g. 1.1) is generated with an empty path. For example, an original request header with the status line `POST /login.php HTTP/1.1` would have its path stripped and replaced with a status line reading only `POST / HTTP/1.1`. The original `host` header is then added to the new request header and an `x-secure-header` header is added to the new request header with the value `k=base64_encode(Enc_Key); c=base64_encode(Enc_Hdr)`. Eventually, the final request header should look like the following:

```
POST / HTTP/1.1
host: untampered.info
content-length: xxx
content-type: application/x-www-form-urlencoded
x-secure-header: k=base64_encode(Enc_Key);
c=base64_encode(Enc_Hdr)
```

f) Encrypting and encoding the request body

Like the request header, the same symmetric key *K* is used encrypt `2 || SHA256(request_body) || request_body` with AES-256-CBC. This ciphertext, along with its IV, (`Enc_Body = IV || AES_256_CBC(2 || SHA256(request_body) || request_body, K)`) is then encoded in URL-safe Base64. The final request body would then look like `c=url_safe_base_64_encode(Enc_Body)`.

g) The entire (protected) request

Combining the protected request header and body, we would end up with the following complete request:

```
POST / HTTP/1.1
host: untampered.info
content-length: xxx
content-type: application/x-www-form-urlencoded
x-secure-header: k=base64_encode(Enc_Key);
c=base64_encode(Enc_Hdr)

c=url_safe_base_64_encode(Enc_Body)
```

Notice that the *confidentiality* of all sensitive data (e.g. path, cookies, body) are now protected by strong encryption. The symmetric key that is encrypted with the origin server's public key ensures the *authenticity* of the recipient by making sure that only one with the private key (e.g. the origin server and not a CDN node) will be able to decrypt the contents of the request. The cryptographic hash digests included in the ciphertexts also help to ensure the *integrity* of the received request as we will see later.

2) *Receiving the request (middleware)*

a) Check if the request is made using the Secure Channel protocol

On receiving a request, the middleware checks if the request header `x-secure-header` exists. If it does not, the middleware treats this request as unprotected and simply passes on the request. Otherwise, it continues to parse the protected request.

b) Decoding and decrypting the symmetric key

Once the middleware determines that the request is made using the Secure Channel protocol, it extracts and decodes the encoded value `base64_encode(Enc_Key)` from the header to obtain `Enc_Key`. `Enc_Key` is then decrypted using the server's private key to obtain *K*, the session's symmetric key.

c) Decoding and decrypting the request header

Likewise, the encoded value `base64_encode(Enc_Hdr)` is extracted from the header and decoded to obtain `Enc_Hdr = IV || AES_256_CBC(1 || SHA256(request_header) || request_header, K)`. Using the IV and the symmetric key *K*, we then decrypt the ciphertext to obtain the plaintext `1 || SHA256(request_header) || request_header`. The first character of the plaintext is verified to be `1` to ensure that the request header is not maliciously swapped with the request body. The hash of the received `request_header` is then computed and matched against the received hash digest to ensure the *integrity* of the received request header. Then, by checking the `content-length` of the request header, we know if there is content in the request body. This prevents an attacker from subtly removing the request body.

d) Decoding and decrypting the request body

The encoded value `url_safe_base_64_encode(Enc_Body)` is extracted from the request body and decoded to obtain `Enc_Body = IV || AES_256_CBC(2 || SHA256(request_body) || request_body, K)`. Using the IV and the symmetric key *K*, we then decrypt the ciphertext to obtain the plaintext `2 || SHA256(request_body) || request_body`. Again, the first character of the plaintext is verified to be `2` to ensure that the request body is not maliciously swapped with the request header. The hash of the received `request_body` is then computed and matched against the received hash digest to ensure the *integrity* of the received request body.

e) Passing on the decoded and decrypted request

By this point of time, we have verified the *integrity* of the request. The symmetric key *K* is then bound to the request and the original request is passed on to the web application.

3) Sending the response (middleware)

a) Check if the request was made using the Secure Channel protocol

To check if the request was made using the Secure Channel protocol, the middleware simply checks for the presence of a symmetric key *K* bound to the original request. If such a key does not exist, then the response is returned as it is.

b) Initialize a new response and an AEAD cipher

Otherwise, the middleware proceeds to creates a new response with the same status code. It also initializes an Authenticated Encryption with Associated Data (AEAD) cipher (implemented using AES-128-CBC and HMAC-SHA256) with the symmetric key *K*. The first 128 bits of the key *K* (*K1*) is used as the HMAC_SHA256 key and the last 128 bits of the key *K* (*K2*) is used as the AES_128_CBC key.

c) Encrypting and encoding the response header

The entire response header is then encrypted using the AEAD cipher to produce `Enc_Hdr = IV || AES_128_CBC(response_header, K2) || HMAC_SHA256(status_code || AES_128_CBC(response_header, K2) || length(status_code), K1)`. `Enc_Hdr` is then encoded in Base64 and added to the new response header with a key-value of `x-secure-header: base64_encode(Enc_Hdr)`.

d) Encrypting and encoding the response body

Also, the entire response body is encrypted using the AEAD cipher to produce `Enc_Body = IV || AES_128_CBC(response_body, K2) || HMAC_SHA256("body" || AES_128_CBC(response_body, K2) || length("body"), K1)`. `Enc_Body` is then encoded in Base64 and added to the new response header with a key-value of `x-secure-body: base64_encode(Enc_Body)`.

e) The entire (protected) response

Finally, the protected response would look like the following:

```
HTTP/1.1 200 OK
x-secure-header: base64_encode(Enc_Hdr)
x-secure-body: base64_encode(Enc_Body)
```

Using the strong encryption within AEAD, we have ensured the *confidentiality* of data in the response. The *integrity* of the response is also protected by the HMAC within the ciphertexts. Lastly, the *authenticity* of the response sender is ensured since only one with possession of the private key will be able to obtain the correct symmetric key (aside from the request originator) to produce the correct HMAC. Also, by using the symmetric key *K* received in the request to protect the response, we ensure that only the request originator (i.e. the client) will be able to decrypt the response. This ensures the *authenticity* of the response receiver.

4) Receiving the response (browser)

a) Check if the response should be processed using the Secure Channel protocol

If a "secured request" was made by the browser, the transaction will be expecting a "secured response" from the server. This is checked using the presence of a transaction key (*K*) in the transaction. Also, every "secured response" will eventually contain the header `x-authentication-results` set by the browser. If this header was set in the original response, it will be overwritten to prevent spoofing of authentication results.

b) Generating the sub-keys

Based on the previously generated symmetric key *K*, the two sub-keys *K1* and *K2* are then generated where *K1* is the first 128 bits of *K* and *K2* is the last 128 bits of *K*.

c) Processing the response header

The header value of `x-secure-header` is first extracted to obtain `base64_encode(Enc_Hdr)`. This encoded value is then decoded to obtain `Enc_Hdr = IV || AES_128_CBC(response_header, K2) || HMAC_SHA256(status_code || AES_128_CBC(response_header, K2) || length(status_code), K1)`. A HMAC of the received status code, ciphertext and length of received status code is computed and matched against the received digest to ensure the *integrity* and *authenticity* of the received response header. Verification of the associated data (status code) is necessary to ensure that the response header was not swapped with the response body. Using *K2*, the ciphertext is then decrypted to obtain the original response header. This decrypted and authenticated response header is then used to overwrite the original "secured" response header (the authentication results header is preserved). If all verifications steps had been successful, the authentication results for the header will be a "pass". Otherwise, it will be a "fail".

d) Processing the response body

Similarly, the header value of `x-secure-body` is extracted to obtain `base64_encode(Enc_Body)`. The encoded value is then decoded to obtain `Enc_Body = IV || AES_128_CBC(response_body, K2) || HMAC_SHA256("body" || AES_128_CBC(response_body, K2) || length("body"), K1)`. A HMAC of the string "body", the received ciphertext and the length of the associated data is computed and matched against the received digest to ensure the *integrity* and *authenticity* of the received response body. Verification of the associated data ("body") is necessary to ensure that the response body was not swapped with the response header. Using K2, the ciphertext is then decrypted to obtain the original response body. This decrypted and authenticated response body is then used to overwrite the original "secured" response body (if any). If all verification steps had been successful, the authentication results for the body will be a "pass". Otherwise, it will be a "fail".

e) Writing the authentication results

By this point of time, we would have verified the *integrity* and *authenticity* of the entire response. These authentication results are then written in the `x-authentication-results` response header and later parsed by the browser extension.

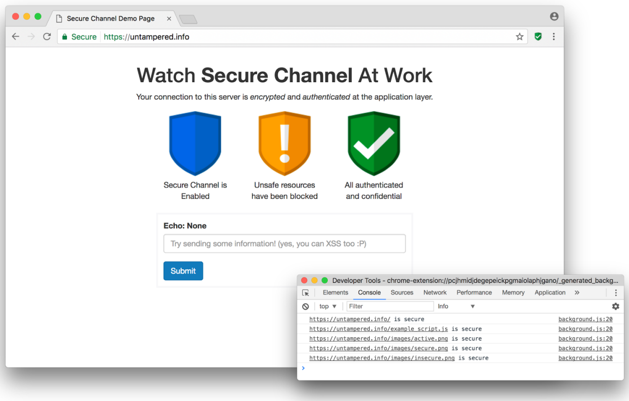
5) Processing and acting on the authentication results (browser extension)

a) Processing received headers

On receiving headers, the Secure Channel browser extension checks for the presence of the `x-authentication-results` header set by the browser. If this header is non-existent, then the response is returned as normal. Otherwise, we check both the results of the response header and body authentication. If either of the results are not "pass" (e.g. none/fail), then the resource is deemed "insecure" and blocked. The extension's icon would also turn amber if any of the resources on the page have been blocked (indicating the lack of message integrity/authenticity). On the other hand, if both checks are "pass", then the resource is deemed "secure" and allowed to be rendered. If all resources on the page are "secure", then a green shield will be presented as the extension's icon as an indication that all 3 security properties (confidentiality, integrity and authenticity) had been met.

D. Concept Validation

To test the validity of our solution (and its implementation), we had set up a secured origin server, a public key store, and enrolled the public key of our origin server against our domain untampered.info in the public key store. We then performed man-in-the-middle attacks using mitmproxy to test our setup against data tampering and sniffing attacks. When the channel is secure (untampered, encrypted and authenticated), all resources would load normally and the Secure Channel extension icon would display a green shield as shown below:



Secure Channel Demo Page
Secure https://untampered.info
Secure Channel is Enabled
Unsafe resources have been blocked
All authenticated and confidential
Echo: None
Try sending some information! (yes, you can XSS too :P)
Submit
Developer Tools - chrome-extension://pcjhmidjdegepeickpgmaiolaphjgano/_generated_backg...
Elements
Console
Sources
Network
Performance
Memory
Application
top
Filter
Info

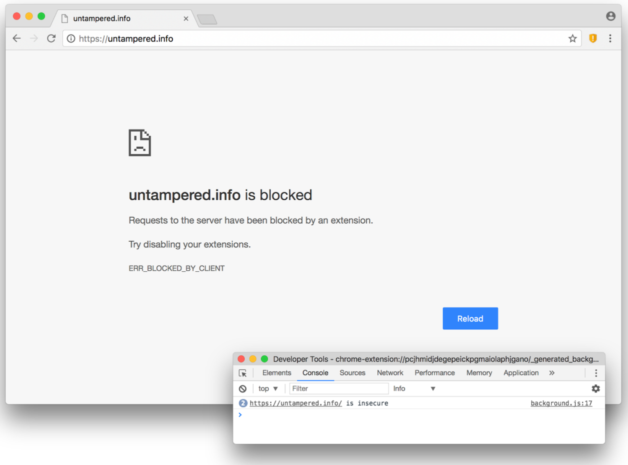
Watch Secure Channel At Work

Your connection to this server is *encrypted* and *authenticated* at the application layer.

```
https://untampered.info/ is secure                          background.js:20
https://untampered.info/example_script.js is secure         background.js:20
https://untampered.info/images/active.png is secure         background.js:20
https://untampered.info/images/secure.png is secure         background.js:20
https://untampered.info/images/insecure.png is secure       background.js:20
```

Fig. 3. What is seen when the channel is secure

Next, when we tampered with any part of a resource's request/response, we would correctly see the resource being blocked by the Secure Channel extension (seen in the figure below) because of it failing the integrity verification step. The same thing would happen if one were to forge a response to the client without knowing the correct symmetric key. Therefore, we are confident that our solution correctly protects the integrity and authenticity of requests and responses.



untampered.info is blocked

Requests to the server have been blocked by an extension.

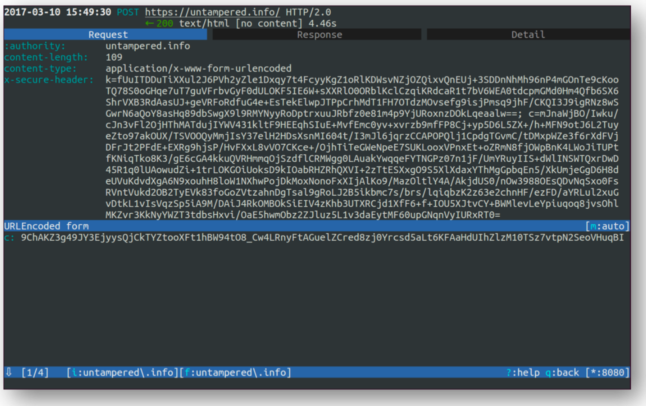
Try disabling your extensions.

ERR_BLOCKED_BY_CLIENT

```
2 https://untampered.info/ is insecure                      background.js:17
```

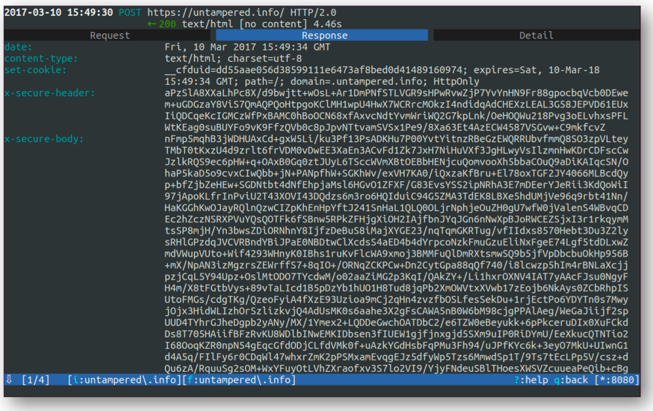
Fig. 4. When a man-in-the-middle tampers with information in the channel

Finally, we also checked to see if a man-in-the-middle could possibly obtain the original request or response by eavesdropping. As seen in the figure below, nothing sensitive can be discerned from the protected request and response. To begin with, the actual path of the request had been masked so that all requests will appear to be sent to the root path. The original header fields and body content can also no longer be seen in the protected request. Furthermore, if we attempt to decode the protected payloads, we would only end up with a useless chunk of binary (a ciphertext). Consequently, we conclude that the confidentiality of the original requests and responses have also been protected.



```
2017-03-10 15:49:30 POST https://untampered.info/ HTTP/2.0
                         ←200 text/html [no content] 4.46s
        Request                      Response                      Detail
:authority:       untampered.info
content-length:   109
content-type:     application/x-www-form-urlencoded
x-secure-header:  k=fUuITDDuTiXXul2J6PVh2yZle1Dxqy7t4FcyyKgZ1oRlKDWsvNZjOZQixvQnEUj+3SDDnNhMh96nP4mGOnTe9cKoo
                  TQ78S0oGHqe7uT7guVFrbvGyF0dULOKF5IE6W+sXXRlO0ORblKclCzqiKRdcaR1t7bV6WEA0tdcpmGMd0Hm4Qfb6SX6
                  ShrVXB3RdAasUJ+geVRFoRdfuG4e+EsTekElwpJTPpCrhMdT1FH7OTdzMOvsefg9isjPmsq9jhF/CKQI3J9igRNz8wS
                  GwrN6aQoY8asHq89dbSwgX9l9RMYNyyRoDptrxuuJRbfz0e81m4p9YjURoxnzDOkLqeaalw==; c=mJnaWjBO/Iwku/
                  cJn3vFl2OjHThMATdujIYWV431kltF9HEEqhSIuE+MvfEmc0yv+xvrzb9mfFP8Cj+yp5D6L5ZX+/h+MFN9otJ6L2Tuy
                  eZto97akOUX/TSVOOQyMmjIsY37elH2HDsXsnMI604t/I3mJl6jqrzCCAPOPQlj1CpdgTGvmC/tDMxpWZe3f6rXdFVj
                  DFrJt2PFdE+EXRg9hjsP/HvFXxL8vVO7CKce+/OjhTiTeGWeNpeE7SUKLooxVPnxEt+oZRmN8fjOWpBnK4LWoJiTUPt
                  fKNiqTko8K3/gE6cGA4kkuQVRHmmqOjSzdflCRMWgg0LAuakYwqqeFYTNGPz07n1jF/UmYRuyIIS+dWlINSWTQxrDwD
                  45R1q0lUAowudZi+1trLOKGOiUoksD9kIOabRHZRhQXVI+2zTtESXxgO9S5XlXdaxYThMgGpbqEn5/XkUmjeGgD6H8d
                  eUVuKdvdXgA6N9xouhH8loW1NXhwPojDkMoxNonoFxXIjAlKo9/MazOltlY4A/AkjdUS0/nOw3988OEsQDvNqSxo0Fs
                  RVntVukd2OB2TyEVk83foGoZVtzahnDgTsal9gRoLJ2B5ikbmc7s/brs/lqiqbzK2z63e2chnHF/ezFD/aYRLul2xuG
                  vDtkL1vIsVqzSp5iA9M/DAiJ4RkOMBOkSiEIV4zKhb3UTXRCjd1XfF6+f+IOU5XJtvCY+BWMlevLeYpiuqoq8jvsOhl
                  MKZvr3KkNyYWZT3tdbsHxvi/OaE5hwmObz2ZJluz5L1v3daEytMF60upGNqnVyIURxRT0=
URLEncoded form                                                                                    [m:auto]
c: 9ChAKZ3g49JY3EjyysQjCkTYZtooXFt1hBW94tO8_Cw4LRnyFtAGuelZCred8zj0Yrcsd5aLt6KFAaHdUIhZlzM10TSz7vtpN2SeoVHuqBI









⇩ [1/4]   [i:untampered\.info][f:untampered\.info]                                 ?:help q:back [*:8080]
```

Fig. 5. What is seen in a secured request



```
2017-03-10 15:49:30 POST https://untampered.info/ HTTP/2.0
                         ←200 text/html [no content] 4.46s
              Request                     Response                     Detail
date:                       Fri, 10 Mar 2017 15:49:34 GMT
content-type:               text/html; charset=utf-8
set-cookie:                 __cfduid=dd55aae056d38599111e6473af8bed0d41489160974; expires=Sat, 10-Mar-18
                            15:49:34 GMT; path=/; domain=.untampered.info; HttpOnly
x-secure-header:            aPzSlA8XXaLhPc8X/d9bwjtt+wOsL+Ar1DmPNfSTLVGR9sHPwRvwZjP7YvYnHN9Fr88gpocbqVcb0DEwe
                            m+uGDGzaY8ViS7QmAQPQoHtpgoKClMH1wpU4HwX7WCRrcMOkzI4ndidqAdCHEXzLEAL3G58JEPVD61EUx
                            IiQDCqeKcIGMCzWfPxBAMC0hBoOCN68xfAxvcNdtYvmWriWQ2G7kpLnk/OeHOQWu218Pvg3oELvhxsPFL
                            WtKEag0suBUYFo9vK9FfzQVb0c8pJpvNTtvamSVSx1Pe9/8Xa63Et4AzECW4587VSGvw+C9mkfcvZ
x-secure-body:              nFmp5mqhB3jWDHUAxCd+gxW5Li/ku3Pf13PsADKHu7P00YvtYitnzRBeGzEWQRRUbvfmmQ8SO3zpVLtey
                            TMbT0tKxzU4d9zrlt6frVDM0vDwEE3XaEn3ACvFd1Zk7JxH7NiHuVXf3JgHLwyVsIlzmnHwKDrCDFscCw
                            JzlkRQS9ec6pHW+q+OAxB0Gq0ztJUyL6TSccWVmXBtOEBbHENjcuQomvooXh5bbaCOuQ9aDiKAIqcSN/O
                            haP5kaD5o9cvxCIwQbb+jN+PANpfhW+SGKhWv/exVH7KA0/iQxzaKfBru+El78oxTGF2JY4066MLBcdQy
                            p+bfZjbZeHEw+SGDNtbt4dNfEhpjaMsl6HGvO1ZFXF/G83EvsYSS2ipNRhA3E7mDEerYJeRii3KdQoWiI
                            97jApoKLfrInPviU2T43XOVI43DQdzs6m3ro6HQIduiC94G5ZMA3TdEK8LBXe5hdUMjVe96q9rbt41Nn/
                            HaKGGhKwOJayRQlnQzwCIZpKhEnHpYftJ241SnHaL1QLQ0OLjrNphjeOuZH0gU7wfW0jValenS4WBvqCD
                            Ec2hZczNSRXPVuYQsQOTFk6fSBnw5RPkZFHjgXiOH2IAjfbnJYqJGn6nNwXpBJoRWCEZSjxI3r1rkqymM
                            tsSP8mjH/Yn3bwsZDiORNhnY8IjfzDeBuS8iMajXYGE23/nqTqmGKRTug/vfIIdxs8570Hebt3Du3Z2ly
                            sRHlGPzdqJVCVRBndYBiJPaE0NBDtwClXcdsS4aED4b4dYrpcoNzkFmuGzuEliNxFgeE74Lgf5tdDLxwZ
                            mdVWupVUto+Wif4293WHnyK0IBhs1ruKvFlcWA9xmoj3BMMFuQlDmRXtsmwSQ9b5jfVpDbcbuOkHp956B
                            +mX/NpAN3izMgzrsZEWrffS7+8qIO+/ORNqZCKPCw+Dn2CytGpa88qQf740/i8lcwzp5hIm4rBNLaXcjj
                            pzjCqL5Y94Upz+OslMtODO7TYcdwM/o02aaZiMG2p3KqI/QAkZY+/Li1hxrOXNV4IAT7yAAcFJsu0NgyF
                            H4m/X8tFGtbVys+89vTaLIcd1B5pDzYb1hUO1H8Tud8jqPb2XmOWVtxXVwb17zEojb6NkAys0ZCbRhpIS
                            UtoFMGs/cdgTKg/QzeoFyiA4fXzE93Uzioa9mCj2qHn4zvzfbOSLfesSekDu+1rjEctPo6YDYTn0s7Mwy
                            jOjx3HidWLIzhOrSzlizkvjQ4AdUsMK0s6aahe3X2gFsCAWA5nB0W6bM98cjgPPAlAeg/WeGaJiijf2sp
                            UUD4TYhrGJheDgpb2yANy/MX/1Ymex2+LQDDeGwchOATDbC2/e6TZW0eBeyukk+6pPkceruDIx0XuFCkd
                            Ds8T705HAiifBFzRvKU8WDlbINwEMKIDbsen3fIUEW1gjfjnxgjd5SXm9uIP0RiDYmU/EeXkucQTNTio2
                            I68OoqKZR0npN54gEqcGfdODjCLfdVMk0f+uAzkYGdHsbFqPMu3Fh94/uJPfKYc6k+3eyO7MkU+UIwnG1
                            d4A5q/FIlFy6r0CDqWl47whxrZmK2pPSMxamEvqgEJz5dfyWp5Tzs6MmwdSp1T/9Ts7tEcLPp5V/csz+d
                            Qu6zA/RquuSg2sOM+WxYFuyOtLVhZXraofxv3S7lo2VI9/YjyFNdeuSBlTHoesXWSVZcuueaPeQib+cBg
⇩ [1/4]   [i:untampered\.info][f:untampered\.info]                              ?:help q:back [*:8080]
```

Fig. 6. What is seen in a secured response

Hence, we have in this section successfully validated the three security promises – confidentiality, integrity and authenticity – of our protocol.

E. *Drawbacks*

In this section, we will highlight some drawbacks in the use of the Secure Channel protocol and will also propose some simple workarounds to circumvent these drawbacks, where possible.

1) *Consuming Cached Content*

One of the major advantages of using a CDN is caching of static content at different datacenter and retrieval of this cached content by the client. At first glance, it may seem that adoption of the Secure Channel protocol would not allow this, since the expectation is that requests and responses to and from a particular domain would be encrypted using a dynamically generated symmetric key. While this is true, the request is only encrypted (and response decrypted) if the domain is enrolled for Secure Channel. A simple workaround to allow cached content to still be retrieved from the CDN, is to host the static content in a different subdomain (or domain) from the domain which has been enrolled for Secure Channel. For example, if https://private.untampered.info has been enrolled for Secure Channel, content from a different subdomain, https://static.untampered.info can be cached and consumed from the CDN.

2) *Web Application Firewalls*

Another feature that CDNs commonly offer is a Web Application Firewall, which is an online security solution that filters out bad HTTP traffic between a client and a web application. With nearly the complete HTTP request made from a client being encrypted, Web Application Firewalls, if enabled, may not be effective since the CDN would not be able to determine the plaintext content of the request to perform filtering, when a request is made through Secure Channel. A workaround for this is to modify the implementation of the protocol to only encrypt a subset of fields of the request, as defined by the customer/website owner, while making the request. The website owner can define the fields when the website is being on-boarded to the public key store and will be retrieved, cached and utilized by the browser when making a request.

3) *Performance Impact due to Cryptographic Operations*

When Secure Channel is being used for communication with a website, additional cryptographic operations are performed on the requests and responses, as described in previous sections. These cryptographic operations performed at client and server side would incur a performance penalty. Since these operations are at the core of the Secure Channel protocol, they cannot be avoided. In the next section, we will determine the actual performance degradation incurred and determine if this penalty is acceptable.

F. *Performance Testing*

With additional cryptographic operations being performed on both requests and responses, we fully expect that there would be slight performance degradation when a website is loaded. In this section, we will investigate the performance penalty incurred and determine whether this penalty is acceptable.

For the scope of the performance test, we have disregarded the time spent retrieving the public key from the public key store for a domain since this is a one-time operation and the information would be available in-memory for subsequent requests.

The tests are conducted through the modified Chromium browser, with none of the network resources (.html, .js, .css files) being cached.

We compare the time taken to load the project website https://untampered.info with and without the Secure Channel protocol being enabled. The time taken for 20 page loads in each case were computed and the averages compared.

Description	**Page**	**Avg. Time**
Without Secure Channel	https://untampered.info	222 ms
With Secure Channel	https://untampered.info	253 ms

From the above table, we observe that there is 13.9% increase in the time taken to load a web page when the Secure Channel protocol is employed. We believe that this overhead, though not insignificant, is acceptable when we take into consideration the security advantages offered by the Secure Channel protocol.

G. *Implementation*

The implementation of the various portions of the project can be found in the following repositories:

i. https://github.com/thngkaiyuan/chromium - The modified Chromium implementation to perform client-side cryptographic operations on the request and response.
ii. https://github.com/thngkaiyuan/secure-channel - Chrome extension to block content and notify the user

if the integrity of a web page (or any resource on a web page) has been compromised.

iii. https://github.com/thngkaiyuan/secure-channelmiddleware - The middleware component which performs cryptographer operations on the requests and responses being sent to and from the origin server

iv. https://github.com/akshayv/integrity-guard-key-api - The public key server implementation to retrieve the stored public key for a domain.

III. Conclusion

All in all, our team had identified a systemic weakness in the use of CDNs that would allow rogue CDN nodes to violate the expected security properties of a HTTPS session. We then followed up with a strong proposal and concrete implementation of our solution which would protect the end-to-end security of requests and responses served even through a CDN. Finally, we also conducted our own experiments to validate the soundness and security of our implementation. After many iterations of refinement and testing, we are confident that have arrived at a polished product that is production-ready and even scalable should there be enough adopters.

While our proposed solution and implementation may present some drawbacks, mostly with regard to the introduction of overheads and delays, we have detailed some methods, which can be put in place to circumvent them. Additional overheads, involving cryptographic operations, which lie at the core of the Secure Channel protocol, cannot be circumvented.

Aside from our finished product, we also took home valuable lessons on open-source development, cryptography, protocols and computer networking. For instance, in the past, we would never have imagined ourselves compiling our own browsers but today, we proved our past selves wrong by having overcome this big challenge and we can proudly say that we have compiled our own versions of Chromium and even implemented our own protocol in it. We also familiarized ourselves with the various cryptographic primitives and mastered the Hypertext Transfer Protocol (HTTP) itself. Finally, we also learned to develop Chromium extensions, websites and use network tools such as the Chrome developer tools, Wireshark and mitmproxy to dissect and even modify network traffic.

Overall, we have had an incredible learning journey discovering problems, dissecting and analysing them, formulating solutions, and finally to implement and validate our solutions.

Acknowledgment

We would like to thank Professor Liang Zhenkai for giving us this opportunity to work on this project, not to mention the ample guidance and advice that he had dispensed through his patient mentorship.

References

[1] Beal, V. (n.d.). What is Content Delivery Network (CDN)? Webopedia Definition. Retrieved April 17, 2017, from Webopedia: http://www.webopedia.com/TERM/C/CDN.html

[2] Čandrlić, G. (2013, July 11). Advantages of Content Delivery Network. Retrieved April 17, 2017, from GlobalDots Blog: http://www.globaldots.com/advantages-of-content-delivery-network/

[3] Remaker, P. (2015, November 4). What is the difference between Let's Encrypt and Universal SSL? Retrieved April 17, 2017, from Quora: https://www.quora.com/What-is-the-difference-between-Lets-Encrypt-and-Universal-SSL/answer/Phillip-Remaker

[4] Graham-Cumming, J. (2017, February 23). Cloudflare. Retrieved April 17, 2017, from Incident report on memory leak caused by Cloudflare parser bug: https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug