Hijacking browser TLS traffic through Client Domain Hooking

Piotr Duszyński - piotr@duszynski.eu

08/05/19

Abstract

This paper describes a new variation of a man-in-the-middle (MITM) technique which, under certain circumstances, allows to permanently hijack browsers encrypted HTTP communication channel flow and compromise its confidentiality and integrity. The technique has been named as "Client Domain Hooking", since it relies on a particular way of achieving client-side communication endpoint persistency by forcing an application to communicate only through a chosen attacker-controlled domain through a single intercepted HTTP request and without breaking applications functionality. This technique, currently applies to browser-based applications only (both desktop and mobile) with a significant impact on embeded browsers, where full transparency of an attack can be achieved.

1 Introduction

In the following sections of this paper, we will take a closer look at the details of the Client Domain Hooking technique, then we will review the current security posture of some of the popular browser-based applications, along with top Internet web applications configuration in relation to this form of an attack. This paper will be supported by a practical implementation of the described technique in form of a diagnostic tool that will help to evaulate and improve security of affected applications and related services.

First practical implementation of this technique was released in January 2019 as part of the Modlishka publication, where Client Domain Hooking was

used to highlight currently used 2FA ("two factor authentication") weakness in relation to modern, automated, phishing attacks. The technique allowed to create a universal proxy with a generic support for most of the currently used 2FA schemes, including SMS one-time password (OTP) [1], Time-based One-Time Password (TOTP)[2], HMAC-based One-Time Password (HOTP)[3], Push Based login approval[4], etc.

2 Problem description

2.1 State of the Art

MITM is a well-known type of an attack, where an attacker intercepts traffic between communication endpoints and transparently relays them between each other. Depending on the target protocol, specific modifications are being made on the fly to the relayed traffic flow in order achieve a particular goal. In the most typical case scenario, the goal is to simply compromise integrity and confidentiality of the victim's communication channel. The technique described in this paper aims at interception of encrypted HTTP traffic flow, which is located at the Application layer of the OSI model [5].

Previous research made around this topic, such as the famous SSLStrip project [6] created by Moxie Marlinspike, is using a technique where all of the URLs in an intercepted HTTP responses are stripped off from the SSL layer, by rewriting their URL schemes to HTTP. As a result, this approach achieves great transparency to the end user at the clear-text HTTP traffic level, but it also relies on an active network layer attack.

Other tools like Bettercap[7], which is very accurately called as the "Swiss Army knife" for MITM, comes with a variety of different attack techniques that rely on a network layer MITM. Among many different functionalities, this tool implements TCP, HTTP and HTTPS proxies (including the SSLStrip attack implementation), that can be used to hijack the HTTP traffic flow.

The new thing about the "Client Domain Hooking" MITM technique, is that it achieves permanent HTTP traffic flow interception without the requirement of an active and ongoing network layer traffic redirection. Interception of the client HTTP flow is done only once, through a wide range of available techniques. Furthermore, the TLS traffic is by default trusted by the client, since the proxy is using a real, CA signed, certificate.

2.2 Client Domain Hooking

The Client Domain Hooking technique is based on a MITM attack and relies on a reverse proxy mechanism at its core. That said, the way the proxied traffic is handled in this approach is new and, as a result, enables permanent interception of multi origin encrypted traffic over a single endpoint with a single, CA signed, TLS certificate. Furthermore, all of this can be achieved without actually breaking applications, often complex, functionality and in an entirely transparent manner in some cases. The simplicity of the technique, allows to follow and intercept applications browsing traffic flow in an arbitrary path of encrypted website referrals.

In order for this type of MITM to work in above described way, the following requirements have to be achieved:

- All client generated TLS traffic has to always go through a single, attacker controlled, domain.

  The general principle in this approach is to "hook" the client application in to an attacker-controlled domain. This step has to be done only once. As soon this is achieved, the internal state of clients browser will be adjusted accordingly and all of the subsequent requests will be automatically sent to an attacker-controlled domain. We can say then that the client domains have been "hooked".

  In the simplest, browser attack scenario, in order to "hook" applications domain, an attacker can first intercept and redirect initial clear-text traffic to an attacker-controlled domain (for example, clear-text HTTP traffic is by default sent by the browser, when no URL scheme has been specified in the address bar by the user). This can be done either through a typical MITM network-based attack or by relying on users' direct interaction (ex. malicious links based on open redirect). There are however also other strategies, that can be used to achieve this initial step.

- TLS certificate, used by the endpoint, has to be trusted by the client for all of its target domains.

  In order to ensure that applications functionality stays intact, it is crucial that the offered TLS certificate is trusted by the client for all of the target domains. Furthermore, this has to be achieved without

actually modifying anything on the client itself. The solution to this point is described in detail, in the following section.

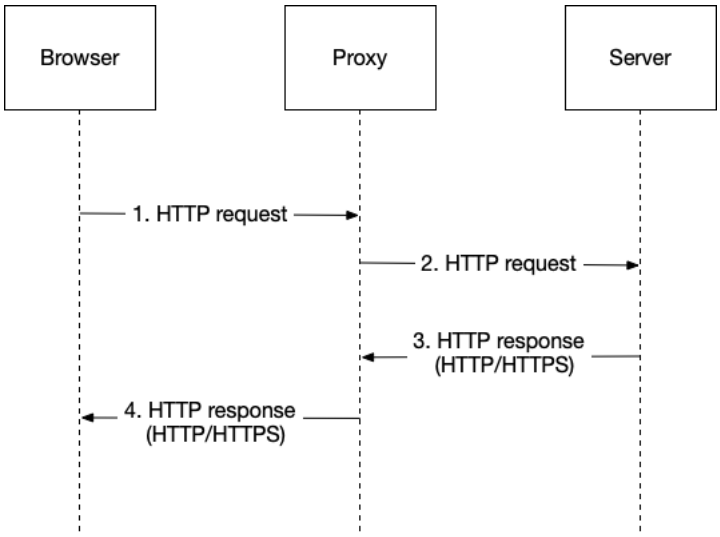
- HTTP traffic has to be handled and modified in such way that all of application functionalities are working properly.

  This includes all security mechanisms, such as SOP[8], CSP[9], etc. Otherwise applications functionality would be broken due to security related errors.

2.3 Browser Hooking Example

In order to better understand how all of the above-mentioned requirements can be achieved in practice the following diagram will be used on a simple browser based attack example. Of course, the same example can be applied to embeded browsers or mobile application WebView.

In the example, we assume that an initial non-encrypted HTTP request (port 80) was initally redirected to the proxy through a network based MITM attack:



1. Browser sends HTTP request which is intercepted by an attacker an redirected to the proxy. There are two cases that can be handled by the proxy during this step:

- initial Client Domain Hooking through a single intercepted clear-text HTTP response.
- handling clients both encrypted and clear-text application traffic that was previously domain hooked.

2. Proxy inspects the incoming request (including context data) and forwards it to the relevant server on victims' behalf.

3. Server responds with a resource that contains multiple URLS with a number of different domains and URL schemes (E.g. HTTP and HTTPS).

4. Proxy inspects the incoming response and replaces all occurrences of FQDNs (fully qualified domain names) with a single, attacker controlled, domain.

   In order to maintain the context for future client requests, information about the original domain is encoded into subdomain of attackers-controlled domain. Without this information, it would not be possible to determine the original destination of the received request and proxying multiple origins over a single domain, would not be feasible. Furthermore, passing context information over a subdomain allows us to use a single wildcard certificate that will be always trusted by the client. One could use another approach, such as passing context information through additional URL parameters. Nevertheless, subdomains appeared to be more suitable, generic, location. Another, possible approach, is to use injected by the proxy JavaScript code and translate all of the relevant FQDNs from within the victim's browser.

   Example handling of FQDNs, for 'evil.tld' Client Domain Hooking in case of stateless proxying:

Original FQDN	Translated FQDN
http://www.dom.dev	http://EC(www.dom.dev).evil.tld
https://account.dom.dev	https://EC(account.dom.dev).evil.tld
https://anotherdom.dev	https://EC(anotherdom.dev).evil.tld

   Where, 'EC' means: "encode and compress", which is a function used to optimize the final FQDN generation. For this example, base32 with

a chosen alphabet and adequate small string compression algorithm was used in order to meet the domain naming standard (RFC 1035).

It should be noted that in case of stateful proxying, the subdomain could be simply replaced with a short identifier that would be mapped to the original domain in the applications memory. Of course, this would mean that the distributed proxy instances would have to additionally rely on a common translation database.

5. Client receives the response and from now it is hooked to an attackers-controlled domain. All further, both encrypted and clear-text communication that originate from the client, will be sent through an attackers' proxy.

2.4 Client Domain Hooking Strategies

In regard to potential techniques that can be used to perform the initial "hook" of the application domain, the following examples can be considered:

- ARP cache poisoning and traffic redirection (classical network based MITM) [10]

  A classical MITM, where a malicious host intercepts all users traffic at a network layer. In this case, all clear-text application traffic can be redirected to a malicious proxy, in an attempt to hook client's applications domains.

- DNS entry hijacking/cache poisoning [11]

  Depending on the client implementation and used security mechanisms, hijacking a domain entry can have none to severe consequences. In case of a standard browser, a single clear-text HTTP request that occured because of an replaced DNS entry will be sufficient to hook the client's domain (given that the HSTS mechanism is not currently active). Whereas, for mobile application WebView, the traffic can be either intercepted entirely transparently (in a similar way as typical browsers) or can be immune to this form of attack due to, for example, application level enforcement of TLS traffic only with additional certificate pinning.

- XSS injection [12]

  "Cross site Scripting" attacks can be used to hijack users traffic flow, by dynamically redirecting client browser to a malicious domain - through the injected, into browsers context, JavaScript code.

- Malicious URLS

  For example, standard mistyped domains along with open redirect vulnerabilities that will try to convince the user that the target website is a legitimate one.

- HTTP mixed-content

  This issue can be abused in case of mobile WebViews in an attempt to inject JavaScript code that will redirect client to an attacker-controlled domain. In regard to the standard browsers this type of traffic is blocked by default since: Firefox 23, IE 9 and Chrome (14.0.785.0). Therefore, only legacy browsers would be vulnerable.

2.5 Consequences

Previous MITM attack techniques, aimed at HTTP communication channel hijacking, were either focused on direct interception of plain clear-text traffic or stripping off TLS layer from subsequent client requests by rewriting URL schemes in a previously intercepted clear-text response. They also often required active network traffic redirection throughout the whole attack. In order to address this issue, best security practices have been developed and new security mechanisms implemented to prevent clear-text transmission and rely on TLS based traffic only to pass sensitive data between a client and related backend. However, because of an assumption that practical multi domain TLS traffic flow interception is not easily achievable and TLS communication channel already provides required confidentiality and integrity, these mechanisms are still not strongly enforced by majority of applications. In reality, in regard to browser-based traffic, the context for confidentiality and integrity of transmitted data begins with its first, often clear-text, HTTP request. Furthermore, in case of embedded browsers and misconfigured mobile application WebViews, the whole process of hijacking the TLS flow through Client Domain Hooking can be entirely transparent to the end user, due to lack of clear indication of the target's domains to the end user or lack of domain validity verification mechanism.

For instance, in an embeded browser-based attack scenario, the following can be achieved:

- Clients target domains are hooked into one particular, attackers, controlled domain, often with a single intercepted HTP request.
- All clear-text and encrypted client's traffic is transparently intercepted and modified by the proxy, without any further network level interference.
- Client will by default trust the endpoint wildcard, CA signed, certificate for all of its target domains.
- Context information is being passed through subdomains, which allows an attacker to deploy stateless and distributed proxying set up.
- Application functionality stays intact.

2.6 Mitigation

In order to prevent this type of attack, client and backend application would have to ensure that its browsing session cannot be easily domain hooked. This can be achieved, though with certain limitations, through the following currently available security mechanisms:

- Enforce TLS traffic

  Enforcing encrypted traffic for domains that support TLS would greatly reduce the attack surface. This can be easily achieved on mobile applications, since the exact protocol scheme can be hardcoded in the code by application developer along with certificate pinning mechanisms and clear-text traffic prevention. On the other hand, standard desktop and embeded browsers are a bit more difficult to manage. Currently most of them tend to default to clear-text protocol when typed in domain doesn't contain any URL scheme and HSTS [13] was not enabled. One potential solution could be based on sending a preflight request to both encrypted and clear-text service endpoints (some of the browsers do this), however that type of check could be easily intercepted and blocked, resulting in a clear-text request again. The only, currently feasible, solution seems to be based on HSTS 'preload' database that is

later hardcoded into the browser build (Chrome, Firefox, Opera, Safari, IE 11 and Edge).

- Enforce HSTS for all domains that handle sensitive data

  Unfortunately, this isn't a silver bullet that would solve the issue entirely but it is currently the first line of defence for browser-based applications.

- Enforce domain and TLS chain verification

  This mostly applies to mobile application WebViews, since all of the modern browsers automatically verify target domain TLS certificate - one exception to this rule are "url spoofing" bugs that further complicate the issue and can result in false sense of security for the end user.

3 Current browser specific landscape

As this has been described in the previous section. In order to hook an application to a particular domain, at least one point of entry has to be provided. In the most typical case scenario, a simple redirection of a single unencrypted HTTP request to an attacker-controlled domain will be sufficient.

First line of defense for browsers, is the HTTP Strict Transport Security (HSTS) mechanism. The following table summarizes the current status of implementation for the most popular browsers:

Browser (version)	Supports HSTS
Opera Mini (All)	false
UC Browser for Android (11.8)	false
Samsung Internet (9.2)	false
IE Mobile (11)	false
IE (11)	true
Edge (18)	true
Firefox (4-68)	true
Chrome (4-76)	true
Safari (7-12)	true
Opera (12.1-68)	true
iOS Safari (7-12.2)	true
Android Browser (4.4-67)	true
Blackberry Browser (7-10)	true
Opera Mobile (46)	true
Chrome for Android (73)	true
Firefox for Android (66)	true

This above table was based on the following source: 'https://caniuse.com'

The conclusion is that the HSTS mechanism is used by most of the browsers. Nevertheless, in order to fully use the benefits of this security mechanism the backend services also have to support it correctly.

The following table presents results of an HSTS support analysis for Alexa rated TOP 1000 domains (test sample: 1000 domain names)

Test	Results (%)
Lack of HSTS support	808 (80%)
HSTS enabled without a 'preload'	114 (59%)*
HSTS enabled with 'max-age' $<$ 60 days	23 (11%)*
HTTP 302 temporary redirect from HTTP to HTTPS	100 (10%)
Number of permanently vulnerable domains	244 (24 %)

*of all HSTS enabled hosts

The number of permanently vulnerable domains has been calculated by adding up all of the applications that do provide a valid HSTS header or use HTTP 301 (permanent redirect) redirection to the HTTPS based service. The sum (756) was deducted from the sample size (1000) and resulted in the

final number of permanently vulnerable domains (244). In this approach we are assuming that both HSTS and HTTP 301, will prevent a browser from sending a clear-text HTTP request, on its second visit to the target domain.

According to the 'RFC2616', unless appropriate 'Cache-Control' or 'Expire' response headers are provided, the HTTP 302 temporary redirect will not be cached by the browser. Unfortunately, in practice, RFC2616 (sec. 10.3.3) is not strictly followed by all of the browsers. It appeared that some of them, such as Firefox and Safari, are caching the temporary responses for a certain period of time. However, for the sake of simplicity, the worst possible scenario has been assumed where user will use an RFC2616 compliant browser.

It should be also noted that an assumption has been made to exclude subdomains from the summary and focus only on the main domains. This should resemble typical users' behavior when typing in the domain name in the browsers address bar.

4 Mobile Application Webviews

Mobile application based on WebView components are susceptible to transparent Client Domain Hooking. This of course depends on a particular application and backend set up, but in majority of current implementations this type of attack is feasible.

4.1 Android

Android WebView is based on the Chromium [14] component and allows local mobile application to load and display remote web pages. Up to Android 7.0 Nougat the WebView was a separate component called 'Android System WebView' that could be installed and updated through the Android play store. In the latest Android version, WebView is handled by the Chrome application directly, which means that the HSTS mechanism is supported and HSTS 'preload' entries are respected. Furthermore, starting with Android 9.0, clear-text application traffic is disabled by default, which is a great improvement in terms of default security practice.

The standard method used in every WebView to load a remote page is declared as following (since API 1.0):

```
public void loadUrl (String url)
```

The following conclusions have been verified on Android 9.0 (PIE) emulator with enabled clear-text traffic in the AndroidManifest.xml ('usesCleartextTraffic' set to true) and default "LOAD_DEFAULT" cache settings:

- Initial clear-text HTTP request can be used to domain hook the application.
- The URL scheme has to be defined in the 'loadUrl' method argument otherwise the remote page will not be loaded.
- HTTP Permanent 301 redirects are being cached for the previously requested resources between the application restarts and system reboots.

4.2 iOS

iOS WKWebView is based on the WebKit [15] framework and allows local mobile application to load and display remote web pages. Up to iOS 8.0 the default webview was based on UIWebView, which suffered from a number performance and security issues out of the box. Starting from iOS 8.0 the default webview is based on WKWebVIew, which was a significant improvement. Furthermore, starting from iOS 9.0, a new security mechanism has been introduced called App Transport Security ('ATS') that enforces secure connections and is by default enabled for all mobile applications. It should be noted however that the requirement of having the ATS mechanism enabled for all of the 'App Store' distributed applications is not currently enforced by Apple.

The standard method used in every WKWebView to load a remote page is declared as following (since iOS 8.0):

```
func load(_ request: URLRequest) -> WKNavigation?
```

The following conclusions have been verified on iOS 12.1 simulator with ATS disabled in the project Info.plist ('Allow Arbitrary Loads' set to true) and cache policy set to the default '.useProtocolCachePolicy':

- Initial clear-text HTTP request can be used to domain hook the mobile application.

- The URL scheme has to be defined in the 'load' method argument otherwise the remote page will not be loaded.
- HTTP Permanent 301 redirects are being cached between application restarts and system reboots. With one exception, when the WKWebView 'load' metod is called with the same domain as the previously loaded one. In such case, the previously cached redirects are being ignored and a new clear-text HTTP request is being sent again.
- HSTS along with the 'preload' mechanism is working accordingly.

5 Mitigation for WebViews

- Prevent HTTP traffic by default: Enforce ATS mechanism for iOS and disallow any clear-text traffic for Android mobile applications.
- Ensure to define valid HSTS headers for your services. Additionally, consider adding your domain to the HSTS 'preload' 'list'
- Disable JavaScript if it is not required by the application.
- Verify your applications with the 'Modlishka' and 'hijack' plugin. Check out the 'NoGoToFail' project.

6 Conclusions

Client Domain Hooking consequences may vary and depend on a client particular implementation and its backend configuration. In the worst-case scenario, it appeared that it is possible to transparently and permanently hijack TLS connection flow for both desktop and mobile browser-based applications through a single intercepted clear-text HTTP request. The review of the current security posture of these applications, revealed that the state of HSTS implementation is not sufficiently strong enough to defend entirely against this form of attack. Furthermore, the review of chosen Internet web applications revealed that in majority the HSTS is not used at all or the 'preload' statement is missing, leaving an opening to Client Domain Hooking attacks through a simple network based MITM. An overall conclusion is that the current security posture of browser-based applications and related services,

from the described attack perspective, is not sufficiently strong and requires further work. In regard to the mobile applications, the easiest mitigation is to forbid any clear-text traffic at the application level and ensure that the endpoint TLS certificate is always verified. Browsers, should rely on the HSTS 'preload' mechanism, to at least partially mitigate the issue. It should be also noted that the presented approach has a potential to be applied to other type of applications and protocols.

Acknowledgement

I would like to give credit to the following people for their review of the final version of this paper: Michał Trojnara (Twitter: @mtrojnar) and Luca Carettoni (Twitter: @lucacarettoni).

References

[1] Wikipedia: One-time-password `https://en.wikipedia.org/wiki/One-time_password`

[2] Wikipedia: Time-based One-time Password algorithm `https://en.wikipedia.org/wiki/Time-based_One-time_Password_algorithm`

[3] Wikipedia: HMAC-based One-time Password algorithm `https://en.wikipedia.org/wiki/HMAC-based_One-time_Password_algorithm`

[4] Wikipedia: Multi-factor authentication `https://en.wikipedia.org/wiki/Multi-factor_authentication`

[5] Wikipedia: OSI model `https://en.wikipedia.org/wiki/OSI_model`

[6] Moxie Marlinspike: SSLStrip `https://moxie.org/software/sslstrip/`

[7] Bettercap is the Swiss Army knife `https://www.bettercap.org/`

[8] MSDN web docs: Same-origin policy `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`

[9] MSDN web docs: Content Security Policy (CSP) `https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP`

[10] Wikipedia: ARP spoofing https://pl.wikipedia.org/wiki/ARP_spoofing

[11] Wikipedia: DNS hijacking https://en.wikipedia.org/wiki/DNS_hijacking

[12] Wikipedia: Cross-site scripting https://en.wikipedia.org/wiki/Cross-site_scripting

[13] Wikipedia: HTTP Strict Transport Security https://pl.wikipedia.org/wiki/HTTP_Strict_Transport_Security

[14] Chromium https://en.wikipedia.org/wiki/Chromium_(web_browser)

[15] Webkit https://en.wikipedia.org/wiki/WebKit