Poster: Security Analysis of HSTS Implementation in Browsers

Yan Jia[1,2], Yuqing Zhang[2,1]
[1]School of Cyber Engineering, Xidian University
[2]National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences
E-mail: {jiay, zhangyq}@nipc.org.cn

***Abstract*—Currently, HTTP Strict Transport Security, used to harden HTTPS, has gained increasing adoption in browsers and servers. We conduct an in-depth empirical security study of HSTS implementation in browsers, then successfully discover several new flaws in storage implementation and interaction with certificates. These flaws enable cookies theft, DoS, and bypassing problems. Moreover, we point out some other concerns including origin risk, entries missing, preload, and complex interaction.**

Keywords—HTTPS; HSTS; security; browser.

I. INTRODUCTION

HTTPS is wildly used to provide confidentiality and integrity to browsers and Web servers. However, it has suffered many serious security problems for years. E.g. SSL Strip techniques and the "clicking through" problem [1]. The primary countermeasure to HTTPS stripping is strict transport security (HSTS) [2]. With the use of HSTS, a server can notify browsers that it wants to always be accessed by a secure version connection, thus preventing user errors, lapses, or redirection tricks. As of April 4, 2017, 83.79% browsers had supported HSTS [3].

We conduct an in-depth empirical security study of HSTS implementation in browsers. We successfully find a security flaw in Chrome storage management allowing attackers to bypass HSTS, which affects all platforms. Chromium community has labeled the flaw as medium severity. Moreover, we find cookies theft and DoS risks when HSTS interacts with browser certificates management. Finally, other issues are summarized. It is highlighted that implementing a mechanism both securely and efficiently is difficult, and deploying a conceptually simple security mechanism is complicated when it interacts with many other features.

II. PREVIOUS WORK

Most previous studies focus on the server. [4] and [5] analyze HSTS deployment on Alexa and Shodan domains. They find basic pitfalls in HSTS configuration and cookies scoped to non-HSTS domains, which makes stripping and cookie theft possible. In 2014, Jose Selvi presented a tool, Delorean, which takes advantage of weaknesses in the implementation of the NTP protocol to invalidate HSTS by making its maximum validity date expire [6].

Compared with the above studies, our study concentrates on the user agent implementation and is more systematic than [5]. Unlike [6], the methods we found to bypass HSTS are based on the browser implementation, not other system mechanisms that support HSTS.

III. STORAGE PROBLEM

Browsers must store offline HSTS status including domains, expiry time and *includeSubDomain* directive etc. This storage is as complicated as managing cookies, considering performance, security and storage size. Firefox uses a "least recently used" algorithm to store HSTS statuses of a maximum of 1024 domains to gain an efficient IO. However, it is easy for a patient attacker to overflow the list and cause the browser to deny HSTS directives of later other sites. Though Chrome does not limit the storage size, we find a bypass vulnerability.

A. Threat model

The adversary is a MITM attacker against HTTPS, who intercepts HTTPS connections between the user's browser and the server. The MITM attacker can utilize a host of well-known MITM techniques (e.g., ARP poisoning and DNS pharming attacks) to re-route all the traffic of the victim to himself. Moreover, the attacker owns an online website and valid certificates of his domains.

B. Methodology

There are two main classes in Chrome managing HSTS. Class *TransportSecurityState* maintains an in-memory database containing the list of hosts that currently have transport security enabled. Singleton class *TransportSecurityPersister* deals with writing that data out to a disk as needed and loading it at startup from the "*TransportSecurity*" file. The *TransportSecurityState* object supports running a callback function when it changes. This *TransportSecurityPersister* object registers the callback, pointing at itself. Before creating a HTTP request, in the factory of the class *URLRequestHttpJob*, Chrome checks *Transport_security_state* of *context* to judge whether to upgrade to HTTPS. At startup, Chrome needs to load transport security states from the disk. However, it does not delay startup for this load, allowing the *TransportSecurityState* to run for a while without being loaded. This means that it is possible for pages opened very quickly to

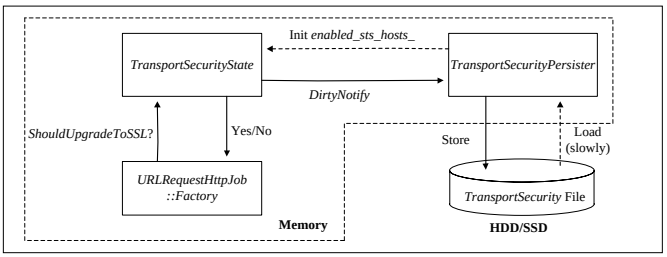

Fig. 1. HSTS process in Chrome.

not get the correct transport security information. The larger the file, the more serious the problem.

C. Experiment and Result

We set up a HTTPS Web server with a valid wildcard certificate, and set HSTS header for each domain. Then, we create ten javascript webworkers to send HTTP requests in the form of *number*.domain.com. The *number* is different in every request, so the size of the *TransportSecurity* file will increase quickly. When the browser loads a dynamic HSTS enabled site (without the "https://" scheme) at startup, we can sniff the HTTP request by Wireshark. Experiments show that 200MB is large enough for a laptop with SSD and i7 CPU. This attack works on all operating systems, but only non-preloaded sites are affected. Chromium community accepted this vulnerability and labeled it "Security Severity-Medium". Developers must use asynchronous file IO carefully, especially for implementing security features.

IV. Certificate Problem

RFC6797 specifies that when connecting to a known HSTS host the user agent "MUST" terminate the connection if there are any errors, whether "warning" or "fatal" or any other error level, with the underlying secure transport. Failing secure connection establishment on any errors should be done with "no user recourse". Moreover, the user agent "MUST" only accept HSTS directives on error-free secure transports. These protection measures are designed to prevent "clicking through" and DoS on browsers.

At the same time, browsers should support self-signed certificates and custom root CAs. The interaction of other existing mechanisms leads to the implementation of HSTS becoming more complicated. We find two risks about HSTS implementation caused by certificates management. Threat model here is the same as section Ⅲ and we assume that the victim is easily tricked when dealing with certificate prompts.

Bypassing Risk. By convention, browsers allow users to add exception certificates for non-HSTS sites. If a user adds a fake certificate in exception for the subdomain before HSTS (with *includeSubDomain* directive) set on the larger domain, there will be a conflict. In Chrome, HSTS cannot protect this subdomain from MITM attackers with that fake certificate until the certificates cache is deleted. It is possible for the attacker to steal domain cookies from HSTS enabled sites. Fortunately, Chrome only caches fake certificates for one week and Firefox does not suffer this attack.

DoS Risk. If the attacker could convince the victim to install the attacker's version of a root CA certificate, he can sniff any HSTS traffic. This type of attack leverage vector is outside of the scope of HSTS, and HPKP is designed to mitigate the problem. However, in this situation, the attacker can set HSTS for any HTTP-only sites. The new mechanism provide the attacker with a new method to implement DoS.

V. Other Concerns

Origin Risk. Browsers must process IDN correctly. FP errors lead to DoS, and FN errors lead to bypassing risks. Firefox and Chrome both had FN errors. Besides, server managers should know the right-to-left host domain name-matching algorithm that means a subdomain cannot disable its HSTS that is set by the larger domain. It is inflexible to use *includeSubDomain* directive in some situations.

Entries Missing. Browsers that support HSTS must check the HSTS policy before sending each HTTP request. It is a difficult job for developers facing more and more new Web standards. E.g., Chrome missed checking HSTS policy when creating WebSockets.

Preload. Because of the open Google preload list, the number of preloaded domains in browsers has increased to 23539 in 2017, from just 1258 in 2015. It is growing fast. There are many unpopular sites, but some popular sites like searching main pages of Google and Baidu are not in the list.

Complex Interaction. Risks above all concentrate on the process of HSTS. However, the browser is a complicated application. It is a challenge to make all features work together flexibly and securely. E.g., HSTS joined with CSP together provide the attacker a new way to sniff the browser history [7].

Acknowledgment

This work is supported by the National Natural Science Foundation of China (No.61572460), National Key R&D Program of China (No.2016YFB0800703), the Open Project Program of the State Key Laboratory of Information Security (No.2017-ZD-01), the National Information Security Special Projects of National Development, the Reform Commission of China [No.(2012)1424], and China 111 Project (No.B16037).

References

[1] Akhawe, Devdatta, and Adrienne Porter Felt. "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness." *U senix Security*. 2013.

[2] HSTS. https://tools.ietf.org/html/rfc6797

[3] Browser support tables for modern web technologies. http://caniuse.com /#search=hsts

[4] Kranch, Michael, and Joseph Bonneau. "Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning." *NDSS*. 2015.

[5] de los Santos, Sergio, et al. "Implementation State of HSTS and HPKP in Both Browsers and Servers." *International Conference on Cryptology and Network Security*. Springer International Publishing, 2016.

[6] Selvi, Jose. "Bypassing HTTP strict transport security." *Black Hat Europe* (2014).

[7] Yan: Weird New Tricks for Browser Fingerprinting. https://zyan.scripts. mit.edu/presentations/toorcon2015.pdf