CTX: Eliminating BREACH with Context Hiding

Dimitris Karakostas*
University of Athens
dimit.karakostas@gmail.com

Aggelos Kiayias*
University of Edinburgh
akiayias@inf.ed.ac.uk

Eva Sarafianou*
University of Athens
eva.sarafianou@gmail.com

Dionysis Zindros*
University of Athens
dionyziz@di.uoa.gr

Abstract

The BREACH attack presented at Black Hat USA 2013 has still not been mitigated, even in the latest versions of TLS, despite the new developments and optimizations presented at Black Hat Asia 2016. BREACH and similar attacks pose a threat against all practical web applications which use compression together with encryption. We present a generic defense method which eliminates problems that arise from compression detectability features of existing protocols. We introduce CTX, Context Transformation Extension, a cryptographic method which defends against BREACH, CRIME, TIME, and any compression side-channel attack in general. CTX operates at the application layer and uses context hiding in a per-origin manner to separate secrets from different origins in order to avoid cross-compressibility.

1 Introduction

In 2012 CRIME [1] showed for the first time that side-channel compression attacks can be successful against TLS. CRIME targeted HTTPS requests and has since been mitigated by disabling compression at the TLS level [2].

In 2013 TIME [3] and BREACH [4] introduced an attack vector that exploited compression on HTTP responses to compromise TLS. This vector takes advantage of the characteristics of the DEFLATE algorithm [5], the basis of most compression applications, in order to steal secrets from applications using stream ciphers.

In 2015 RC4 is considered insecure [6] which forces most websites to use AES block ciphers. Services like Facebook also tried to prevent BREACH [7]

*Research supported by ERC project CODAMODA, project #259152.

using secret masking, although this method protected CSRF tokens only and the fundamental aspects of BREACH were still not mitigated.

In 2016, both Rupture [8] and HEIST [9] introduced new threats regarding compression side-channel attacks. Rupture showed that BREACH can evolve to attack major web applications and steal secrets that were not previously considered as targets of BREACH. It also incorporated statistical methods to bypass noise induced from block ciphers or random data included in the response plaintext.

HEIST demonstrated that compression-based attacks, such as CRIME and BREACH, can be performed solely in the browser by a malicious website or script. It does not require Man-in-the-Middle agents since it abuses the way responses are sent at the TCP level.

These attack techniques, which pose an imminent threat to online security and privacy, have still not been mitigated.

Our work introduces a generic defense method which disqualifies compression detectability features of existing protocols. CTX is a cryptographic method which defends against any compression side-channel attack. It prevents cross-compressibility by separating the secrets from different origins and using context hiding in a per-origin manner.

The existing suggested defense for BREACH [10] includes disabling compression or completely bypassing compression, which results in significant performance penalties. On the other hand, there has not been proposed a solution that keeps compression intact and solves the security issues. It is not known if such a solution is even possible. Our method lies between the two options regarding compression usage. We achieve a good balance by slightly reducing compression size and time performance while achieving full security.

We release an open source implementation of CTX in popular web frameworks both for client-side and server-side web applications. Our implementation runs at the application layer, is opt-in, and does not require modifications to web standards or the underlying web server.

We conclude that if secrets are separated by origin at the application level using the CTX defense, compression side-channel attacks are mitigated.

2 Theoretical analysis

2.1 CTX architecture

CTX depends on separating secrets based on origin. Origin is used to describe the party that generated the secret. The origin can be either the web application or a user. A CTX origin should not be confused with origins of same-origin policy [11], which is a completely different notion. Thus, for any secrets A and B generated from the same origin, whoever is able to change

the secret A can also know the secret B with no violation of the application privacy contract.

CTX is used to protect HTML and other content-type responses of web applications as they travel on the network. It protects only HTTPS responses, not HTTPS requests. The mitigation of the CRIME attack resulted in compressionless HTTPS requests and hence no protection against compression side-channel attacks is required.

It is up to the application developer to decide which portions of the response are sensitive and must be protected as secrets. Sensitive data does not only include high-value secrets such as passwords and CSRF tokens, but also user data that the developer wishes to keep private. Some examples are the bodies of email messages in Gmail, the chat messages received or sent to a friend on Facebook, the contents of documents and spreadsheets in Google Docs and the list of online friends on Facebook or Google Hangouts, as they contain all the important contacts of the victim. Practically any piece of information which is only accessible when logged in is potentially a secret and should be CTX protected.

The developer should also separate the HTML plaintext into contexts. Each context contains portions of the plaintext. Some of these contexts do not typically need compression-security protection, e.g. static HTML portions that are accessible on a website even when logged out. However, sensitive data, as mentioned above, require compression-security protection.

The minimum amount of origins is one origin for the entire response, in which case CTX is not protecting any part of the plaintext, and the maximum is one origin per character. The latter would result in the best possible security under CTX, although compression would be effectively disabled possibly resulting in poor performance. This is the case with defenses such as secret masking.

The portions of the plaintext within contexts of different origin are then forced to compress separately, i.e. not cross-compress. However, compression is achieved within each context. In order to acomplish this, CTX generates a pseudo-random permutation of the secret alphabet for each origin. The secret alphabet is by default the alphabet of ASCII bytes (0 - 128). In order to randomly permute the secret alphabet, we use the Fisher–Yates shuffle algorithm [12]. This algorithm puts all the elements into a hat and continually determines the next element by randomly drawing an element from the hat until no elements remain. The Fisher–Yates shuffle algorithm produces an unbiased permutation meaning that every permutation is equally probable.

Secrets are then permuted by the server using the generated permutation of the corresponding origin prior to TLS encryption and network transmission. Upon arrival on the client side, the inverse permutation is applied to decode the secret. The same permutation is applied to all secrets of the same origin. That way, better compression is achieved intra-origin.

Each time the server issues an HTTPS response, new per-origin permutations are generated. The power of the BREACH attack lies to the assumption that we can perform multiple requests to the target website while the transmitted secret remains the same. Since new alphabet permutations are generated per HTTPS response, the statistical analysis performed by Rupture is no longer feasible.

2.2 A detailed example

Each time CTX protects a secret, three parameters need be defined: the secret, the origin, and the permutation alphabet. In our example we will use the permutation alphabet of ASCII printable characters, which consist of ASCII codes 9-13 and 32-126.

The first secret that needs to be protected is the string "secret1" which is generated by the origin "testsorigin1". In order to protect this secret, CTX will generate a random permutation of ASCII printable characters. Each character in the permutation corresponds to a single ASCII printable character, so that the first character in the permutation corresponds to ASCII 9 and the last to ASCII 126. CTX will then apply the permutation on the secret. Suppose that the correspondence of printable-permutation characters is:

· s → 0

· e → f

· c →)

· r → 4

· t → *

· 1 → l

(...)

The permuted secret will then be "0f)40*l".

A second secret is the string "secret2" from origin "testorigin2". Following the same procedure, CTX will generate a permutation which is described as follows:

· s → t

· e → 9

· c → (

· r → l

· t → j

· 2 → 2

(...)

The second permuted secret will then be "t9(l9j2".

Permutations are randomly generated per origin, so the two permutations for "testorigin1" and "testorigin2" are not dependent and each is needed in order to reverse permute the corresponding secret.

3 Implementation

Our open source implementation of CTX can be used on popular web frameworks for both client-side and server-side web applications.

3.1 Server-side

3.1.1 CTX protected HTML response

The HTML response plaintext consists of a plain HTML structure along with CTX-transformed parts. Each CTX part is annotated using an HTML div tag structured as: <div data−ctx−origin='i'>xyx where i is an integer origin ID and *xyx* the permuted secret after applying permutation for origin i.

Separately in the same response, a JSON will be included.

```
[
'abc',
'cab',
'bac',
...
]
```

where *'abc'*, *'cab'*, *'bac'* are the permutations used to permute secrets of origin 0, 1, and 2 respectively. The JSON is included in a

<**script type**="application/json" **id**="ctx−permutations">

</**script**>

tag in the HTML body.

3.1.2 Developer's actions

We have implemented the server-side CTX defense for the Django [13], Flask [14], and Node.js [15] web frameworks.

A developer should install the CTX package for the corresponding framework in order to use CTX defense. For example, the developer of a Django project should add django-ctx to the installed apps, add the ctx processor to the context_processors setting and use {% load ctx_tags %} to load the ctx tag library in the template.

In order to protect secrets, they should protect them with the ctx_protect tag {% ctx_protect secret origin alphabet %}. The origin and alphabet parameters are optional. If no such parameters are passed, the alphabet is the ASCII alphabet and the origin is a randomly generatered id string with 10 lowercase letters.

After all *ctx_protect* tags that use an origin for the first time, the developer should include the {% ctx_permutations %} tag to include the permutations used for each origin. It is proposed that it is included before the </body> HTML tag.

3.2 Client-side

On the client-side, the browser runs a Javascript library for the inverse permutation on load. It searches for the tag with id *ctx-permutations* to access the JSON table with the stored per-origin permutations and performs the inverse permutation for all ctx tags. The developer should add the script <script src='ctx.js'> </script> before the </head> tag.

3.3 Experiments

We have conducted several experiments to evaluate the performance of web services protected by CTX. The results of these experiments are overwhelmingly positive and should be taken into account when considering incorporating CTX in a web service.

The CTX parameters that affect performance are basically 4: the number of origins, the total amount of plaintext response, the amount of secrets in the response, and the distribution of secrets to origins. Each parameter affects the performance differently and will be examined thoroughly in the following sections.

Our experiments focused on each parameter separately, so the results reflect the performace under each one independently. A combination of the parameters may result in slightly different results when used in real-world systems.

In all our tests we use an HTML web page where each secret is a string of English literature.

3.3.1 Origins

The number of origins mainly affects the overhead of data in the response. The time overhead was found to be insignificant so it will not be included

here. The more origins are used the bigger the response, both compressed and uncompressed, is expected to be.

In our experiment, we use a 650KB page and a fixed amount of secrets, which comprise 1% of the page, and tested the use of a number of origins in the range [0, 50]. The worst case scenario, using 50 origins, resulted in a 12% overhead in the compressed response. In other words, the CTX-protected response is expected to be 1.12x the size of the unprotected, which is practically insignificant considering the security benefits. In comparison, disabling compression would result in 976.8% overhead.

3.3.2 Total response

The size of the total response affects the impact of CTX on protected secrets. Time is again not an issue here and will not be described.

Our experiment tests CTX's performace on web pages ranging from 13KB to 650KB. A base of comparison could be Google Inbox's main page, which is roughly 550KB. We maintain the percentage of secrets and the origins the same throughout the test to 1% and 50 respectively.

Our results show that as the response grows larger, the overhead caused by CTX is minimized. Specifically, for a typical 13KB web page, protecting 1% of it with CTX would add a 228% overhead. A 650KB page on the other hand would suffer an overhead of only 13%. Disabling compression alltogether would add overhead that ranges from 500% to 91% for the same web pages.

3.3.3 Amount of secrets

The percentage of the web page that is protected by CTX also affects the application's performance. In this case, we consider a 650KB web page, a size representative of Facebook's home page, which is protected by CTX using 50 origins.

Our experiment tests the effect when 1% up to 50% of the web page is protected. Results show that the bigger the protected part is, the bigger the effect of CTX on performance will be. Specifically, protecting 1% of such a web page would result in a 5% size overhead, whereas protecting 50% of it would result in a 35% overhead.

This is also the only test that demonstrated a time overhead. Specifically, the 1% case introduces a 1ms overhead, while the 50% case adds 45ms of server-side work.

However, it should be noted that our tests are very strict. A typical website response consists mainly of HTML code or libraries that usually need not be protected. In this case, the amount of secrets in the response would not exceed 1% of the total response, in which case the CTX overhead

as shown by our experiments is totally acceptable. For example, Facebook and Gmail typically only need protect approximately 0.5% of the response.

In comparison, disabling compression would again result in 976.8% load overhead and a network transmittion time overhead that, depending on the client's and the server's network, may be up to seconds.

4 Related Work

Our work is based on both the idea of disabling compression, a successful security mechanism against all compression attacks, as well as the idea of masking secrets, also a perfectly successful security mechanism, employed by Facebook and others. However, these two previous defense ideas lack in compression performance, which we improve on and provide a balance between security and performance. Our implementation is inspired by the authors of the original BREACH paper who wrote in the defenses section "One approach that completely solves the problem is to put user input in a completely different compression context than that of application secrets. Depending on the nature of the application and its implementation, this may be very tedious and highly impractical." This was a good idea, which was not elaborated further by Prado etc.

SafeDeflate [16], published in 2016, is also a proposal to prevent BREACH and CRIME by eliminating compression detectability. It is a modification of the standard Deflate algorithm in which compression ratio does not leak information about secret tokens. However, the size for the dataset compressed when using SafeDeflate is 200%-400% times larger than the size of the compressed dataset if the standard Deflate algorithm is used. Our implementation increases the size by only 5% of the non-CTX protected size.

5 Future Work

CTX defense is the natural conclusion of BREACH, TIME, Rupture, and any compression side-channel attack, since it combines security with compressibility. CTX should be implemented for other web frameworks, such as Ruby on Rails and Lavarel, and can be extended for other encoding standards, such as UTF-8. CTX can also be implemented for web frameworks which build API services and return other data formats like JSON. This way the data served from the database to the user will be transmitted over the network in a secure way.

References

[1] J. Rizzo, T. Duong: The CRIME attack, Ekoparty, 2012

[2] D. Goodin, Crack in Internets foundation of trust allows HTTPS session hijacking, Ars Technica, 2012

[3] Beery, Tal, and Amichai Shulman. "A perfect CRIME? Only TIME will tell." Black Hat Europe 2013 (2013).

[4] Y. Gluck, N. Harris, A. Prado, BREACH: Reviving the CRIME attack, Black Hat USA, 2013

[5] P. Deutsch, DEFLATE Compressed Data Format Specification, RFC 1951, 1996

[6] A.Popov, Prohibiting RC4 Cipher Suites, RFC 7465, 2015

[7] Chad Parry, Christophe Van Gysel, Preventing a BREACH Attack, 2014

[8] D.Karakostas, D.Zindros: Practical New Developments in the BREACH Attack, Black Hat Asia, 2016

[9] M. Vanhoef, Tom Van Goethem: HEIST: HTTP Encrypted Information can be 220 Stolen through TCP-windows, Black Hat USA 2016

[10] Jacob Kaplan-Moss, Security advisory: BREACH and Django, 2013

[11] A. Barth, The Web Origin Concept, RFC6454, 2011

[12] R. Fisher, F. Yates: Statistical tables for biological, agricultural and medical research, 1938

[13] [online] URL: `https://www.djangoproject.com/` [cited November 2106]

[14] [online] URL: `http://flask.pocoo.org/` [cited November 2106]

[15] [online] URL: `https://nodejs.org/en/` [cited November 2106]

[16] M. Zielinski, SafeDeflate: compression without leaking secrets, 2016