Received March 23, 2022, accepted April 5, 2022, date of publication April 11, 2022, date of current version April 20, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3166175

HTB: A Very Effective Method to Protect Web Servers Against BREACH Attack to HTTPS

RAFAEL PALACIOS[1], ANDREA FARIÑA FERNÁNDEZ-PORTILLO[1], EUGENIO F. SÁNCHEZ-ÚBEDA[1], AND PABLO GARCÍA-DE-ZÚÑIGA[2]
[1]ICAI School of Engineering, Institute for Research in Technology, Pontifical Comillas University, 28015 Madrid, Spain
[2]Department of Cybersecurity. Siemens, 80333 Munich, Germany

Corresponding author: Rafael Palacios (rafael.palacios@iit.comillas.edu)

ABSTRACT BREACH is a side-channel attack to HTTPS that allows an attacker to obtain victims' credentials under certain conditions. An attacker with a privileged position on the network can guess character by character a secret session key just by analyzing the size of the responses returned by the server over HTTPS and encrypted. Heal the Breach (HTB) is the proposed technique to mitigate BREACH attack by randomly changing the size of server responses through a modified gzip library. The attacker needs a precision of one byte in the size of the responses to be able to determine if a guess character is part of the secret token. Since the modified gzip library introduces randomness in the size of the response, BREACH becomes ineffective. The only way to circumvent this protection is to make several requests and compute the average size of the response, which minimizes the random effect. Mathematical and experimental results show that, for a random variation of the size from 1 to 10 bytes, an attacker needs to analyze 500 times more packages to obtain enough accuracy and surpass this mitigation. However, if the number of requests increases it is easier to isolate and block the attack using standard Intrusion Detection Systems (IDS). Compared to other mitigations, the approach presented in this paper is very effective, easy to implement for all websites hosted in the server, and produces a negligible increase in normal traffic.

INDEX TERMS BREACH, CRIME, gzip library, HTTPs, side-channel attacks.

I. INTRODUCTION

Standard encryption algorithms have been widely analyzed, and do not contain any mathematical flaws that could be easily exploited. However, there are so called side-channel attacks that collect and analyze information about the data being manipulated by a software system or hardware device [1] and is leaked in different ways. Side-channels attacks based on compression were first investigated in 2002 by Kelsey [2]. He found a way to attack encryption through compression. At ekoparty 2012, Thai Duong and Juliano Rizzo presented CRIME; a compression side-channel attack against HTTPS traffic [3], [4], taking advantage of the fact that HTTP headers show the size of the contents even when the contents are encrypted. CRIME attack was assigned CVE-2012-4929 by MITRE Corporation and it is still an open issue. Later, at Black Hat Europe 2013, Be'ery and Shulman announced TIME [5], an attack against compressed HTTPS responses. Their work uses timing (instead of the ciphertext length) to harvest the information leaked by compression.

The demonstration given with CRIME at ekoparty showed how an attacker might execute this attack to recover the headers of an HTTP request. These headers contain cookies, being these cookies the primary vehicle for web application authentication after login, making this attack rather important. However, the attack also relied on a relatively little-used feature of TLS compression. By disabling TLS/SSL-level compression, this particular attack was completely mitigated. However, at Black Hat conference in Las Vegas on August 2013, BREACH attack was presented by security researchers Gluck, Harris and Prado [6]. BREACH (backronym of Browser Reconnaissance and Exfiltration via Adaptative Compression of Hypertext) attacks HTTP responses instead of HTTP requests. Even if TLS-level compression is disabled, it is very common to use gzip at the HTTP level. Besides, it is also very common that secrets (such as CSRF tokens) and user-input are included in the same HTTP response, and therefore in the same compression context. All this allowed the same attack demonstrated by Rizzo and Duong, but without relying on the TLS-level com-

The associate editor coordinating the review of this manuscript and approving it for publication was S.K. Hafizul Islam.

pression [7]. Very recently, attacks based on BREACH have been implemented to extract information from databases that combine compression and encryption [8].

This paper presents a mitigation called Heal the Breach (or HTB), which randomly modifies the size of HTML responses before encryption is applied. This randomness makes BREACH attack implementations infeasible, and modifications of such implementation to circumvent HTB become too slow and easy to block through standard network protection.

The rest of the paper is organized as follows. Section II describes the details of BREACH attack, using for the examples code from a common web application. Section III describes the way in which BREACH is implemented to obtain a secret key, describing all the steps needed for an effective attack. After BREACH has been described, different mitigation techniques are explained and discussed in Section IV. The proposed mitigation technique, HTB, is introduced in Section V, along the statistical foundations that demonstrate a tremendous number of queries needed to overcome this mitigation. Section VI shows experimental results and describes the physical setup that was implemented and the code needed (available on GitHub [9]) to replicate experiments. Final conclusions are shown in Section VII.

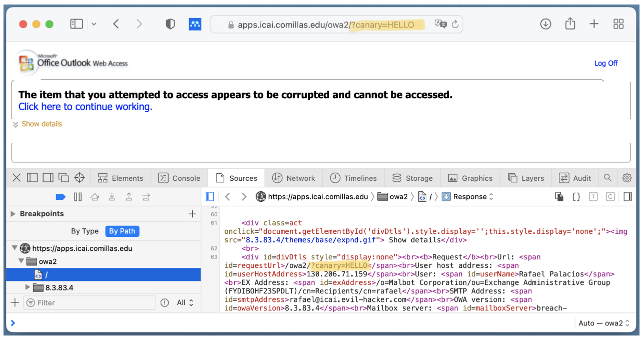
II. BREACH ATTACK

CRIME could be mitigated by disabling TLS/SPDY compression (and by modifying gzip to allow for explicit separation of compression contexts in SPDY), but BREACH attacks HTTP responses. These HTTP responses are typically HTML code compressed using gzip, which is much more common than TLS-level compression. All standard web browsers such as Chrome 99, Firefox 97, Safari 15 and Edge 96, support gzip compressed contents to reduce bandwidth.

Many web applications deliver secrets (such as CSRF tokens) in the HTTP responses. It is also common for web applications to reflect user-input (such as URL parameters) in HTML response bodies. As described in [10] many websites, including Google and Facebook, use this kind of tokens to identify users in some web pages.

Deflate [11], the basic algorithm in the gzip compression library, takes advantage of repeated strings to shrink the compressed payload. Therefore, if a web page contains a secret token and reflects a user-input containing a section of the secret token, that web page will compress better than the same page with a random user-input. Consequently, the attacker can use the reflected URL parameter to guess the secret, one character at a time. A description of how BREACH works can be found at [12]. To be successful with the BREACH attack, the following quite common situations must be met at the server:

- The server must use HTTP compression.
- The website must reflect user-input in HTTP response body.
- HTML code must include a secret token that could be used to impersonate the victim.



The item that you attempted to access appears to be corrupted and cannot be accessed.
Click here to continue working.
Show details

```
    <div class=act
onclick="document.getElementById('divDtls').style.display='';this.style.display='none';"><img
src="8.3.83.4/themes/base/expnd.gif"> Show details</div>
    <br>
    <div id=divDtls style="display:none"><br><b>Request</b><br>Url: <span
id=requestUrl>/owa2/?canary=HELLO</span><br>User host address: <span
id=userHostAddress>130.206.71.159</span><br>User: <span id=userName>Rafael Palacios</span>
<br>EX Address: <span id=exAddress>/o=Malbot Corporation/ou=Exchange Administrative Group
(FYDIBOHF23SPDLT)/cn=Recipients/cn=rafael</span><br>SMTP Address: <span
id=smtpAddress>rafael@icai.evil-hacker.com</span><br>OWA version: <span
id=owaVersion>8.3.83.4</span><br>Mailbox server: <span id=mailboxServer>breach-
```

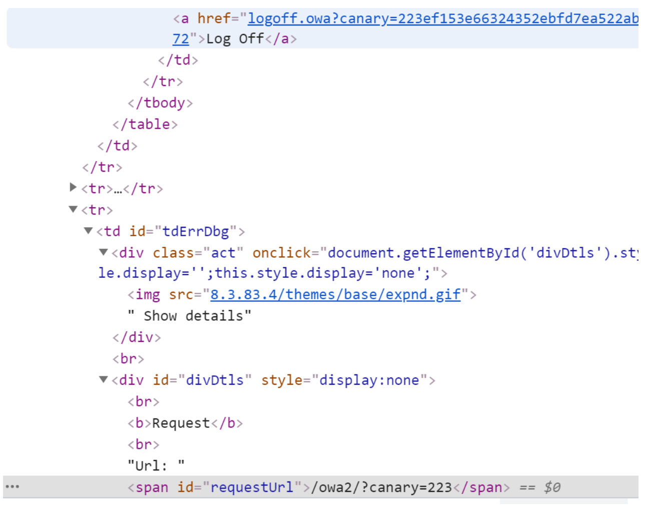
FIGURE 1. Example of webpage which reflects user-input in HTTP response body.

- Also, some known prefix needs to be known to help bootstrap compression.

For all the experimental testing process in our research, we used the output of an old version (that was already patched) of Outlook Web Access (OWA). Any other webpage that uses secret tokens for validation and reflects user-input in the response will also be vulnerable, but we used OWA in our experiments to work in a real and well-known environment. OWA is a web-based application to access Microsoft Exchange mail with a very similar interface to the Outlook client mail program. This old version of OWA, had some search pages in which user-input was reflected in case of error, showing the parameters sent as GET variables to the page in a section called "show details." These pages also included a secret token associated to a "logout" button. If an attacker was able to guess that token, it was possible to make calls to the application using the correct token (called "canary") to impersonate a legitimate user and get access to their email messages. Figure 1 shows an example of such webpage, including the graphical (browser) output on the left and the source HTML code on the right, highlighting the user-input string reflected on the page. A small index.php was used to simulate the responses of the OWA server upon a wrong request by adding the request URL to the "show details" section.

On Figure 2 one can see the section of HTML code where the secret token is written (within the "logout" button), and the section where the user-input is reflected. The distance between the secret token and the user-input is not very large, and within the limits of the deflate algorithm so it takes advantage of this string repetition to apply compression. In the first line in Figure 2 includes the secret (variable "canary") that the attacker wants to discover, and the last line in the figure shows the reflected user-input (canary = 223). Because this string appears twice, the compression algorithm will make this response to be smaller than a request with a different user-input.

It is important to note that the attack is agnostic to the version of TLS/SSL and does not require TLS-layer compression. Additionally, the attack works against any cipher suite because it uses the size of the compressed contents before applying encryption.



```
                <a href="logoff.owa?canary=223ef153e66324352ebfd7ea522ab
                72">Log Off</a>
              </td>
            </tr>
          </tbody>
        </table>
      </td>
    </tr>
  ▶ <tr>…</tr>
  ▼ <tr>
    ▼ <td id="tdErrDbg">
      ▼ <div class="act" onclick="document.getElementById('divDtls').st
        le.display='';this.style.display='none';">
          <img src="8.3.83.4/themes/base/expnd.gif">
          " Show details"
        </div>
        <br>
      ▼ <div id="divDtls" style="display:none">
          <br>
          <b>Request</b>
          <br>
          "Url: "
…         <span id="requestUrl">/owa2/?canary=223</span> == $0
```

FIGURE 2. Sample HTML code of the response.

Theoretically, BREACH can be exploited with a few thousand requests, and could be executed in under a minute [13]. The number of requests required will mostly depend on the secret size and the length of the alphabet that is used to build the secret string (typically hexadecimal characters).

The attack is based on the fact that many webpages are compressed before they are sent to the user, in order to save bandwidth. The size of the compressed response can be obtained even though the response itself will be encrypted, thus TLS does not protect from this attack. The most common tool for webpage compression is gzip which implements a lossless compression algorithm that takes advantage of strings of characters that are repeated throughout the document to save space. Deflate is a combination of LZSS and Huffman coding. LZSS, proposed in 1982 by Storer and Szymanski [14] is a variant of LZ77 [15]. These algorithms are substitution coders in which a string can be replaced by a reference to a previously seen string in the text being compressed. Thus, the higher the number of repeated strings in the document, the more compression this algorithm will be able to perform. Web pages in HTML language are suitable for this type of compression because they are text files and use tags, styles names, and object IDs that are repeated several times in the text.

LZSS is slightly better than its predecessor LZ77, because it omits the chance of compression if the offset/length pair takes more space that the string being replaced. Omitting a potential compression is always an option, and the decompression algorithm is not affected if that compression chance was not exploited. This technique was evaluated as a possible mitigation but was later disregarded in favor of a most effective way described below.

In addition to LZSS, a Huffman coding is applied in Deflate. Huffman coding replaces each symbol in the document with a code, associating the shortest codes with the most frequent characters. One of the main difficulties encountered when carrying out BREACH attacks is precisely due to Huffman coding. The most frequent letter in a webpage tends to be coded using Huffman algorithm and sometimes it is difficult to find the correct character of the secret because the size of the compressed response does not behave as expected. The fact that Huffman coding induces some unexpected results requires a bit more sophisticated implementations of the attack.

III. IMPLEMENTATION OF THE BREACH ATTACK

A robust implementation was designed to exploit BREACH attack against the old OWA server. The exploit developed, which is available at the GitHub repository of this paper [9], is a Python program that makes two requests for each of the hexadecimal characters [’0’, ’1’, ’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’, ’a’, ’b’, ’c’, ’d’, ’e’, ’f’] that can be used in the secret token. Theoretically, one request with the wrong character of the secret will yield a larger response size (one more byte) that the response obtained with the correct character of the secret. In the following example, where the token starts with character ’a’ we can see this difference of one byte for two requests (Note that the size of response can easily be obtained using curl Unix command):

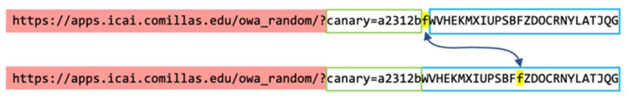
curl –header “Accept-Encoding: gzip” -o file.gz URL

https://apps.icai.comillas.edu/owa2/?canary $=$ 7 $\rightarrow$ size 1606

https://apps.icai.comillas.edu/owa2/?canary $=$ a $\rightarrow$ size 1605

However, the characters that follow the user-input, such as “</span>” or any other HTML tag, may be compressed as well. In addition, Huffman coding generates problems because of the variation of the frequency of the characters as more letters are added to the token string. In the case of the OWA webpage, the most frequent character is letter ’e’, so guessing secrets that included such character where more difficult to guess. In other pages, the most frequent character could be different than letter ’e’, but Huffman coding will use the most frequent one. To minimize problems related with Huffman coding and surrounding HTML tags, instead of making only one request per possible character in the alphabet, it is very common to make two requests using a padding string. In our python implementation of the attack, all requests include a randomly generated padding string that includes each of the letters in the alphabet once, so it has a length of 26 characters. The padding string was created in this way to avoid character repetitions, while keeping the frequency of all the letters. The first request is created with the guess character located after the already known secret and before the padding string. In the second request, the guess character is in the middle of the padding string (see Figure 3).

- The first structure puts the under-test character before the 26-character-long padding string: URL (red) + name of the token variable (known prefix) with the already discovered secret (green) + tested character (yellow) + padding text (blue).
- The second structure puts the under-test character in the middle of the padding string: URL (red) + name of the



```
https://apps.icai.comillas.edu/owa_random/?canary=a2312bfWVHEKMXIUPSBFZDOCRNYLATJQG
https://apps.icai.comillas.edu/owa_random/?canary=a2312bWVHEKMXIUPSBFfZDOCRNYLATJQG
```

FIGURE 3. **Two queries to find out if 'f' is the next character in the secret token.**

variable with the already discovered secret (green) + first half of the padding (blue) + tested character (yellow) + second half of the padding (blue).

The length of the responses to these requests is obtained within the program with the function: response.headers.get ('content-length') and both lengths are compared. If the first response's length is smaller than the second, the character is considered as the one that follows the secret.

The algorithm does this comparison between both requests made for each character deciding if the under-test character is a candidate. Other implementation may assume that the first character in the alphabet that meets the size condition is the actual character in the secret. Nevertheless, our implementation runs all the characters in the alphabet, then decides which one is part of the secret key in accordance with the results obtained:

- Just one candidate (the typical result), then that character is added to the already discovered secret string.
- Empty list of candidates, then try an alternative query method.
- More than one candidate, then try an alternative query method. This situation does not happen very often in a normal server, only while launching the attack against a server that implements HTB mitigation presented in this paper.

Several alternative methods were developed to overcome special situations in which just two queries per potential character do not yield a trustful result. These methods use the same padding string but reducing the length by 1, 2, or 3 characters. We call Method #1 to the one that uses the original 26-character-long padding string. Method #2 is the one using 25 characters padding, and Method #3 the one that uses 24 characters. Method #4 was also implemented to use a padding 23 characters long, but in our experiments was never applied. By changing the padding length also, the total size of the output changes, but uncompressed response is always the same for two consecutive queries (test character at the beginning or in the middle). Also, altering the padding length changes the uppercase letters surrounding the guess character which avoids potential compression artifacts unrelated to the secret. As a final attempt, a different padding string is used, but in all our test the character was found before reaching that option.

IV. PREVIOUS WORKS ON MITIGATIONS TO THE BREACH ATTACK

BREACH exploit does not attack a software with a vulnerability, situation that could rapidly be solved by patching the software. Attacks such as BREACH, cannot be patched because they rely on flaws of a lower-level protocol. However, there are several requirements that must be met to be successful with the attack. The ability of the attacker to sniff the network is difficult to avoid, especially in WiFi environment with public access, and absolutely out of the control of the website owner. There are other mitigations related to the server that have been described [6], [10], but none of them are easy to implement. One such solution is to disable the server's compression because without compression it is impossible to get size information about characters in the secret token. However, if compression is disabled, the network traffic will increase dramatically affecting performance and increasing computing and network costs. Other mitigations include reprogramming the webpages, to avoid reflecting user-input, avoid the use of tokens, to refresh the tokens very frequently, or to add randomly generated text to change the output on every call. All these mitigations require a lot of programmers' hours and audit processes.

Several researchers have proposed mitigation methods aimed to protect servers in a global way. The objective is to protect all websites and pages hosted in the server by implementing these mitigations. As mentioned before, disabling compression is a global way to mitigate BREACH but has severe consequences in increasing bandwidth. Effective global mitigations can be divided into two main groups: methods that modify the webpage and methods that modify the compressor.

A. METHODS THAT MODIFY THE WEBPAGE

Dimitris Karakostas, Eva Sarafianou, Aggelos Kiayias and Dionysis Zindros proposed CTX in 2016 [16], Context Transformation Extension, a cryptographic method which operates at the application layer and separates secrets from different origins in order to avoid cross-compressibility. For this, the attacker uses frameworks to tag data individually with an identifier called "origin" which identifies where the data came from. This robustly mitigates BREACH, but not in every circumstance. CTX does not protect data in embedded JavaScript because it surrounds the protected data in <div> tags, so this would break the page as the developer aimed to program it. Also, CTX is not robust for XML and JSON formats because to automatically undo the transformation, this method relies on JavaScript. But with other formats, there is no automatic way of undoing the transformation because they do not have a way of containing instructions, they are only data formats. Thus, this method finds some weaknesses and does not find a solution for all setups.

Token Masking Frameworks are procedures to mask tokens, so that the token's representation changes in each of the responses. A fixed-dictionary implementation was first introduced by Alawatugoda [17] to identify in a web page strings with a format compatible with security tokens or other secrets. The method was later improved to facilitate the implementation on a web server [18] and variants were introduced to protect against CRIME attack [19]. There are frameworks such as debreach proposed by Paulsen in his MS Thesis [20] that generate one-time pads to mask tokens from webpages, unmask them and validate their security automatically, the developer only needs to install a module similar to the PHP layer of web servers that pre-process the outputs. Each time the webpage is requested, the one-time pad with which the token is masked is different and protects the token from the upcoming compression process, so the attacker will not be able to identify if the plaintext input matches the token or not. This method is only useful for tokens because with other data, the webpage might be corrupted, and it would break the view that the developer designed for the page. Therefore, token masking frameworks could be a mitigation for BREACH, but it needs to be able to identify tokens correctly, otherwise it will break applications.

In a review about new developments on BREACH, Karakostas and Zindros [21] expressed their concerned about the possibility of using BREACH to steal session cookies, and proposed an extension of the Content Security Policy. They propose a new value called same-origin-only to specify how a cookie should be treated. Unfortunately, such parameter should be approved by W3C Web Application Security Working Group before it can be implemented in websites.

B. METHODS THAT MODIFY THE COMPRESSOR

SafeDeflate [22] is a modification of Deflate, the standard compression algorithm, to change the compression ratio and avoid the leak of information about secret tokens. It prevents LZ77 from leaking information from strings composed by characters within the alphabet used to create secrets. This alphabet is configurated by the developer, so the type of secrets generated on the server depend on the alphabet used and the length configured. SafeDeflate is an easy and effective method but some of its drawbacks are an increased bandwidth because any string that looks like a secret (written with characters in the secret alphabet) will not be compressed, and an overhead in computing power to pre-process webpages for finding potential secret strings. It is not a very general method but a particular application that requires manual configuration of the alphabet for each website. In terms of bandwidth, it can increase greatly depending on the structure of the secret token (alphabet used) and its compression ratio is as high as 146% worse than unmodified Deflate, even when the alphabet is limited to hexadecimal digits.

deBreach strategy was also implemented as is a modified Deflate compressor that is able to exclude sensitive data from compression [23]. Developers do not need to identify sensitive data manually, instead it is identified thanks to a static analysis, by explicit byte ranges or through string matching. Once this data is detected, the result of the analysis is used to rewrite the HTML code so that sensitive data is tagged with some annotations before the webpage is sent to the modified compressor. This enhanced compressor does not apply LZ77 compression to the detected sensitive data, avoiding the compression between the content of the secret and the attacker-controlled supplied text. Theoretically the bandwidth does not increase significantly as long as the secret strings and user-inputs are identified properly. Such identifying phase has a cost in the computing power used during the page processing before applying compression.

V. PROPOSED MITIGATION METHOD

Heal the Breach (HTB) is the method to mitigate the BREACH attack proposed in this paper and involves modifying the length of the compressed HTML responses of the server. Since gzip is the most standard compression algorithm for HTML webpages, by modifying the gzip library used by the web server, it is possible to protect the whole website without the time-consuming and expensive task of reviewing pages that contain secret tokens and reflect the variables of the request. Compared with other advanced mitigation methods based on modifying compression, HTB does not require additional computing power and the increase of network traffic is negligible.

HTB produces a randomized response size by creating a fake filename in the header section of the gzip file format which is meaningless for the browser. It was initially considered the possibility of randomly skipping opportunities of compression when the algorithm finds repeated strings, since it is not mandatory to take advantage of all repetitions found. This would have been similar to other compression-based mitigation approaches in which compression is avoided in sections of text that look like token strings. However selective compression does not provide enough granularity and the file sizes produced in this way could be somehow predictable. More importantly, the increment in bandwidth could be very significant if the algorithm misses several crucial compression chances, as it happens in other mitigation proposals [22].

The implementation of the modified gzip library, called gzip_randomizer, uses a parameter to specify the maximum length of a fake filename that is added to the compressed version of the output. The function generates a random number, using a uniform distribution (C rand function initializing the random seed with /dev/random), within the range specified. Then it creates a fake filename with the length obtained.

According to RFC 1952 [24] in gzip format the three first bytes in the file work as a magic number to indicate that the filetype is gzip compressed with Deflate, and the fourth byte is a flag that indicates if the original uncompressed filename is provided. After 6 more bytes, the filename is included as a null-terminated string (see Figure 4). Depending on the random number obtained, shorter or longer filenames could be included at this point. Most of the tests performed inserted a maximum of 10 characters (9 characters in the filename plus the end-of-string character '\0'). However, it was tested that all browsers could manage filenames of length 100 without problem.

Every time gzip-randomizer is called with parameter 10, a new random number is generated, and the length of the resulting compressed file will be 1 to 10 bytes longer that the original gzip function. As it is shown on Figure 5, this small variation in the size produces enough confusion to

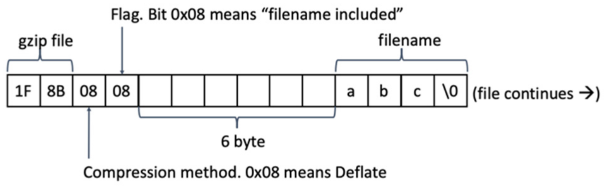

Flag. Bit 0x08 means "filename included"
gzip file
filename
1F
8B
08
08
a
b
c
\0
(file continues →)
6 byte
Compression method. 0x08 means Deflate

FIGURE 4. gzip file structure. Example using filename "abc."

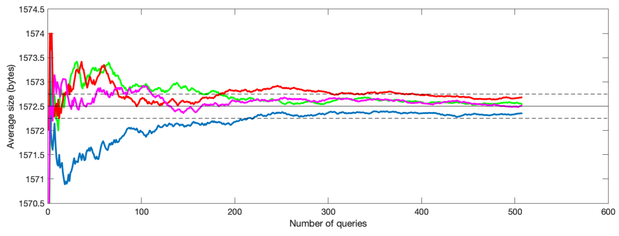


FIGURE 5. Error in the file size minimizes as a function of the number of queries while computing the mean value. Figure shows 4 experiments, all converging towards 1572.5.

make the attack extremely more complicated. Within the exploit program, comparing the size of the response when the under-test character is inserted before the padding string or in the middle of the padding string, yields unpredictable answers. The size of each request will only be reliable after making several queries with the same input and computing the average size. It can be seen on Figure 5, that by repeating the same query and computing the mean size, the values obtained for several runs tend to converge towards the theoretical value of 1572.50.

The number of queries necessary to obtain a file size accurate enough can be obtained by statistical analysis. The response's length is no longer fixed due to the random size introduced by HTB; instead, it can be statistically modeled as a random variable. In particular, the response's length depends on the under-test character (called x) used in the request:

$$L_{ox} = L + \varepsilon$$
$$L_{xo} = \begin{cases} L - 1 + \varepsilon, & xissecret \\ L + \varepsilon, & xisnotsecret \end{cases}$$

where, L_{ox} is the stochastic response's length when x is placed after (or middle) the padding string and it is not compressible, and L_{xo} is the length when x is placed before the padding string and next to the already known secret. L is the deterministic response's length and ε is the random noise added by means of the proposed modification of the compression that randomly increments the response's length up to n.

Due to the proposed implementation based on the rand function, the random variable follows a discrete uniform, i.e. $\varepsilon \sim U\{0, n-1\}$. Thus, $L_{ox} \sim U\{L, L+n-1\}$ and $L_{xo} \sim U\{L, L+n-1\}$ if x is not the secret, or $L_{xo} \sim U\{L-1, L+n-2\}$ otherwise.

Modified exploit algorithm for dealing with HTB mitigation will need make several (n) queries in order to get a simple random sample for L_{ox} and L_{xo}. The mean values of the lengths obtained are calculated to minimize the random effect, $\overline{L_{ox}}$ and $\overline{L_{xo}}$, respectively. If the mean response's length $\overline{L_{xo}}$ is smaller than $\overline{L_{ox}}$, the character x is considered the one that follows the secret.

In order to properly adjust the value of the parameter n, we have estimated, as a function of n, the minimum number of queries q needed to successfully determine if x is the character that follows the secret or not.

According to the central limit theorem [25], [26], for a given value of n, the sample mean of the random variable L_{ox} can be approximated by the following normal distribution if q is large (e.g. $q > 30$):

$$\overline{L_{ox}} \sim N\left(E\left(L_{ox}\right), \frac{\sigma_\varepsilon^2}{q}\right) = N\left(L + \frac{n-1}{2}, \frac{\frac{n^2-1}{12}}{q}\right)$$

Based on the theoretical distributions of $\overline{L_{ox}}$ and $\overline{L_{xo}}$, it is possible to set an interval with a confidence level of $1- \propto$ [27]:

$$\text{LOX}: L + \frac{n-1}{2} \pm z_{\propto/2} \cdot \frac{\sigma_\varepsilon}{\sqrt{q}}$$
$$\text{LXO is secret}: L + \frac{n-1}{2} - 1 \pm z_{\propto/2} \cdot \frac{\sigma_\varepsilon}{\sqrt{q}}$$

The margin of error of the confidence intervals is half the confidence interval:

$$m = z_{\propto/2} \cdot \frac{\sigma_\varepsilon}{\sqrt{q}}$$

Thus, the minimum number of requests can be estimated as a function of the margin of error, the confidence level and n:

$$q = \left(z_{\propto/2} \cdot \frac{\sigma_\varepsilon}{m}\right)^2 = \left(z_{\propto/2} \cdot \frac{\sqrt{\frac{n^2-1}{12}}}{m}\right)^2$$
$$= z_{\propto/2}^2 \cdot \frac{\frac{n^2-1}{12}}{m^2} = z_{\propto/2}^2 \cdot \frac{n^2-1}{12 \cdot m^2}$$

Since non-overlapping confidence intervals are required to distinguish between the means of $\overline{L_{ox}}$ and $\overline{L_{xo}}$ when x is the secret, then m should be set to 0.25. In this way, the maximum error of a theoretical size difference of 1 byte will be at least 0.5. If we wish to have this margin of error with 95% confidence, then substituting the appropriate values into the expression gives the number of queries needed

$$q = z_{\propto/2}^2 \cdot \frac{n^2-1}{12 \cdot m^2} = 1.96^2 \cdot \frac{10^2-1}{12 \cdot 0.25^2} = 508$$

Therefore, if $n = 10$, meaning that the size may increase up to 10 bytes, then 508 queries are required to successfully determine if x is the character that follows the secret or not, with a 95% confidence that the error in the estimated size is smaller than 0.25. So we need 508 queries to estimate the size of L_{ox} response and another 508 queries to estimate the size for the L_{xo} response. Then by comparing these sizes one can conclude if x is part of the secret or not.

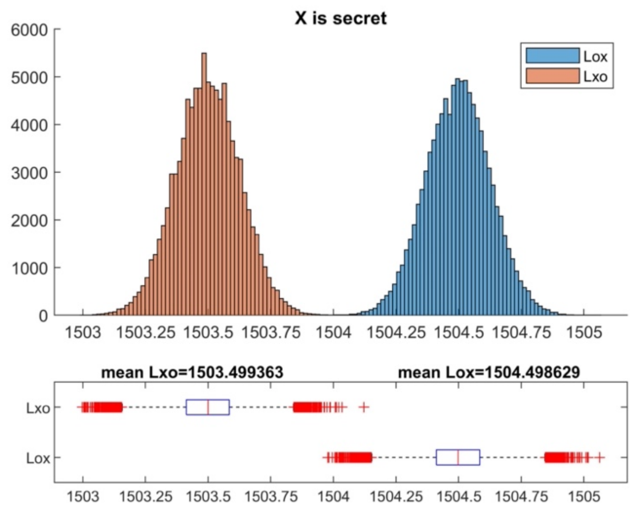


FIGURE 6. Distributions of the size estimation (n = 10, q = 508) for requests Lxo (left gaussian) and Lox (right gaussian), being x the correct secret character.

As an example, Figure 6 shows the distributions of the mean sizes after a Montecarlo simulation. The most likely value of the mean size for L_{xo} when x is the secret is 1503.5 because the size without randomness is 1499 and the average added length is 4.5. On the right side of Figure 6 we get a mean size for request L_{ox} of 1504.5, which is one byte larger that L_{xo}, as it was expected. In this large simulation (100,000 replicates were executed) we can see that the size estimated for L_{xo} is typically between 1504.25 and 1504.75 (+/- 0.25 around 1504.50). For the request L_{ox} or L_{xo} when x is not the correct secret, the average values obtained match the blue gaussian on the right. If in one case the value is as high as 1504.74, and the second request as low as 1504.25, the difference is still less than 0.50 and it will be considered zero, which is the right choice. But due to the 95% confidence level, sometimes a larger error is obtained, and the exploit may wrongly conclude than x is part of the secret. There is also a risk of missing x as a valid character in the secret if we get a blue value smaller than 1504.25 and an orange value larger than 1503.75 because the difference will be smaller than 0.50, hence making the program reject x as a secret character.

VI. EXPERIMENTAL RESULTS

In order to test the proposed approach to mitigate BREACH attacks it was decided to prepare a server able to support the standard compression and the randomized compression simultaneously. The webpages used for the testing phase were those of the OWA interface described before. One server folder called 'owa' was used to generate responses in the standard way, allowing Apache to use the system's gzip compression. Then a second folder, called owa_random was created to produce a response generated with gzip plus a program called gzip_randomizer that changes the length of the output without modifying the HTML contents. The system used for testing is a Linux CentOS version 8, with 4 Intel Xeon CPU E5-2650 @ 2.00GHz and 8 GB RAM, running Apache 2.4.37.

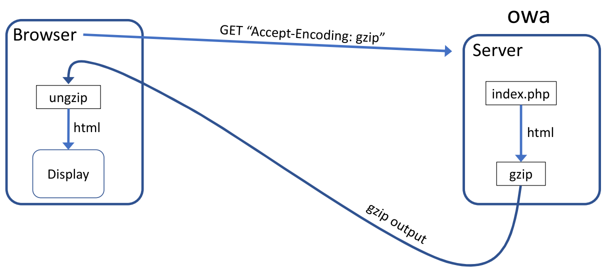


FIGURE 7. Request from standard browser that accepts gzip encoding. Standard system without HTB protection.

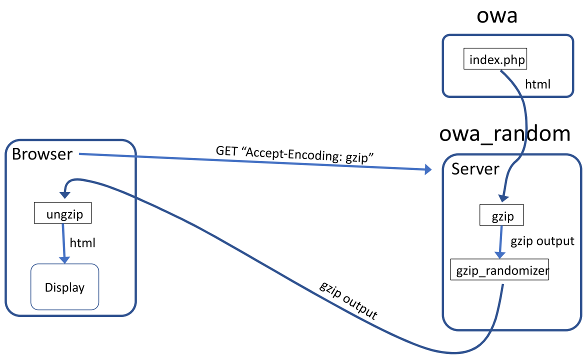


FIGURE 8. Server uses gzip_randomizer (HTB protection) to modify the length of the gzip output.

The browser-server communication process is shown in Figure 7. Calling to owa folder with any parameters will generate an error page like Figure 1. Internally in the web server, the HTML code generated by index.php will be compressed by Apache using the systems gzip library. This happens if the Browser identifies itself as capable of managing gzip output by using header "Accept-Encoding: gzip." Figure 7 shows the process of sending the request from the Browser to the owa folder, then the server using gzip library to send the compressed response, and finally the browser uncompresses the response to obtain plain HTML, sending it to the display.

In the case of owa_random folder, a php program available in the paper's GitHub [9] was implemented to obtain the same HTML code directly from owa folder, and then create the compressed version using gzip and gzip_randomizer before returning the output to the browser. Since gzip_randomizer only changes the length of the compressed output without altering the contents, the final output on the browser will look exactly the same.

All common browsers (Chrome 99, Firefox 97, Safari 15 and Edge 96) support gzip compression and therefore they send requests including the "Accept-Encoding" clause in the header. In case that the browser does not support gzip or requests are made with other programs such as curl or wget with a modified header, Apache automatically avoids compression. Consequently, gzip is not called in either process presented in Figure 7 or Figure 8 and the HTML code is returned directly. However, in this case it will not be possible for the attacker to find the secret token because without compression it is not possible, taking into account that the communication between browser and server is HTTPS encrypted.

To be consistent with the possibility of avoiding compression, owa_random was implemented to check if the header of the request contains the directive "Accept-Enconding: gzip." If we are in a situation in which compression is not requested, it just calls owa and returns the contents without calling gzip nor gzip_randomizer.

By configuring the server in this way, it is possible to make tests in the standard way or using HTB mitigation, which is a good solution for research purposes. The final implementation involves changing the system's gzip library for the one implementing randomized-length filenames, so in that way all the websites hosted in the server will be automatically protected.

The main objective of the experiments was to run the exploit against owa folder and then against owa_random, measuring execution time and most importantly the number of queries required to obtain the secret token.

Without mitigation, the exploit was able to find all 32 characters of the secret canary in 1 minute and 45 seconds, using 1216 queries. In 6 cases the first method used by the exploit (using a padding string consisting of 26 characters) did not find any candidate characters in the alphabet of 16 characters, so method #2 was invoked. In the case of this particular token, if method #1 was not able to find the answer, then method #2 did; there was no need to use method #3 or greater. Many different canary strings were tested to ensure the robustness of the exploit program. In general, the first digits are a bit harder to find because the strings are shorter, and compression may occur not just with the secret but also with other text in the webpage. Also, letter 'e' is harder to find because it is the most frequent letter in the page and very likely to be coded in a different way due to Huffman coding. This is an example of the output provided by the exploit program:

```
==============================
Program finished
10b3d7cfe0e92615475486facb8d293a
2222 2 2
Num queries: 1216
```

So, the number of queries matches the expected value: 32 (secret length) + 6 (uses of method 2) = 38 cycles. For each cycle we need 16 × 2 queries (being 16 the size of the alphabet), hence the total number of queries is $(32+6)*32 = 1216$.

The number of queries per second depend on the quality of the internet connection, and ranged from 10 to 15 q/s. The time and number of queries could be reduced in average by half, if the loop that goes through all the characters in the alphabet stops as soon a candidate is found. Without any mitigation, in those difficult digits in which method #1 fails and we need to change methods, the answer was always an empty list of potential candidates. Nevertheless, the exploit algorithm is good as it is because with the introduction of HTB mitigation there are some cases in which several potential candidate letters are found after analyzing the complete alphabet.

When HTB mitigation was introduced, the exploit that was so effective in finding any kind of canary secrets on the owa installation, was completely unable to find correct letters. Since any request returned a length with a variance from 1 to 10, it was a matter of luck which characters were selected as candidate or if several potential candidates were found (hence changing method).

Based on the statistics previously presented, a more advanced exploit was implemented, that for each request a set of 508 queries were executed and the mean size was computed. (This variant of the exploit is also available at the GitHub project of the paper [9]). The results obtained are the following:

```
==============================
Program finished
10b3d7cfe0e92615475486facb8d293a
2223 2 2
Num queries: 633984
==============================
real 978m43.159s
```

Now the number of queries is 633,984 instead of 1216 and the total time is 979 minutes instead of 1.75 minutes, with a speed of 10.8 q/s. The speed is very similar in terms of queries per second, because gzip_randomizer does not overload the server.

It is interesting to notice, that comparing this result (with HTB) with the previous case (without HTB), the first number '3' in the secret was found using method #3 instead of using method #2 as before. This is because while running method #1, the system found 2 potential candidates: character '3' and character '5'. Extracting the details from the log file created during these tests we can see that in the case of character '5' there is a small error in the estimation of the size:

```
1569.52 1570.58 ['3']
1570.32 1570.83 ['3', '5'] → wrong size estimation for '5'
```

The correct size for character '5' is 1570.50 for both requests, with '5' next to the already known secret string "10b" and the character '5' in the middle of the padding string, because it cannot be compressed either way. The estimation for the first response is less than 0.25 smaller than 1570.50, but the second estimation is larger than 1570.75 because of the 95% confidence. Therefore, a difference of 0.51 in size of both queries was obtained, making the program think that the first request yields a smaller size that the second, hence selecting '5' as a candidate for the secret string. Nevertheless, the character was solved with method #3, and the full secret was obtained. Taking into account that 95% confidence is quite good, the small mistakes that eventually occur, are overcome due to the robustness of the exploit procedure.

TABLE 1. Number of queries required.

	No HTB	HTB n=10	HTB n=100
Queries per character	32	16,256	1,638,944
Total queries for a 32-byte secret	1,024	520,192	54,446,208

As a summary of the results section, Table 1 shows the minimum number of queries required for different situations. The number of queries is related with the level of difficulty achieved by the mitigation. The results are shown for different configurations of the parameter n in HTB, and without mitigation, always assuming a 16-character alphabet for typical secret generated with hexadecimal symbols.

VII. CONCLUSION

Heal the Breach or HTB is a very effective mitigation against BREACH attack that can be implemented in a global way in the webserver by replacing gzip library with the proposed method. HTB will protect all websites hosted in the server without the need of changing webpages or programs running behind the webpages.

Experimental results show that a normal exploit against BREACH is totally ineffective after adding HTB mitigation. The only way to circumvent HTB would be to make too many more queries to minimize the random effect in order to obtain reliable size information. Nevertheless, increasing the number of queries from 1k, which may be legitimate traffic, to nearly 1M which is obviously an attack, makes it feasible to block the exploit through Intrusion Detection Systems (IDS). Moreover, the number 508 which represents how many times harder it is to break the system, was obtained for maximum random size of $n = 10$ bytes. For a value of $n = 100$, which yields a filename length supported by all standard browsers, the number of queries required for each request is in the range of 50k.

In terms of side effects induced in the server, there is minimal impact. Other previously known mitigations require processing webpage contents to detect text that contains secret keys, and they avoid compression of sections of webpages; these actions utilize CPU and increase bandwidth used. However, implementing HTB mitigation, CPU overload is negligible because the compression algorithm runs in the same way as without HTB, the only difference is generating one random number and writing a few extra characters at the beginning of the output. The traffic increases minimally, because an average of 5 extra characters in 1570 bytes compressed output is just a 0.3% increase. Considering that the original file size is 4600 bytes, a change from 1570 to 1575 is insignificant compared to the firstly proposed mitigation consisting of cancelling compression all together.

ACKNOWLEDGMENT

The authors would like to thank BREACH author Angelo Prado, for many conversations and hints about the implementation of the attack, and also would like to thank several students who worked on different implementations BREACH exploits and attempts to modify compression algorithms: Antonio Vázquez, Nicolás Dulio, Gonzalo Pastor, and Eduard Alsina.

REFERENCES

[1] A. F. Rodriguez, L. H. Encinas, A. M. Munoz, and B. A. Alcazar, "A modular and optimized toolbox for side-channel analysis," *IEEE Access*, vol. 7, pp. 21889–21903, 2019.

[2] J. Kelsey, "Compression and information leakage of plaintext," in *Proc. Int. Workshop Fast Softw. Encryption*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 2365, 2002, pp. 263–276.

[3] J. Rizzo and T. Duong. (2012). *The CRIME Attack*. Accessed: Jun. 13, 2021. [Online]. Available: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1d134dff_1_222

[4] D. Fisher. (2012). *CRIME Attack Uses Compression Ratio of TLS Requests as Side Channel to Hijack Secure Sessions | Threatpost*. Accessed: Jul. 22, 2021. [Online]. Available: https://threatpost.com/crime-attack-uses-compression-ratio-tls-requests-side-channel-hijack-secure-sessions-091312/77006/

[5] T. Be'ery and A. Shulman, "A perfect CRIME? Only TIME will tell," in *Proc. BlackHat Eur.*, 2013.

[6] Y. Gluck, N. Harris, and A. Prado, "BREACH: Reviving the CRIME attack," Tech. Rep., 2013, pp. 1–12.

[7] P. G. Sarkar and S. Fitzgerald. (Jun. 2014). *Attacks on SSL a Comprehensive Study of Beast, Crime, Time, Breach, Lucky 13 & RC4 Biases*. [Online]. Available: https//www.isecpartners.com/media/106031/ssl_attacks_survey.pdf

[8] M. Hogan, S. Eskandarian, and Y. Michalevsky, "DBREACH: Database reconnaissance and exfiltration via adaptive compression heuristics," in *Proc. Black Hat*, 2021.

[9] R. Palacios. (2021). *GitHub for PAPER-Heal-the-Breach*. Accessed: Nov. 17, 2021. [Online]. Available: https://github.com/iit-asi/PAPER-Heal-the-Breach/

[10] D. Karakostas and D. Zindros, "Practical new developments on BREACH," in *Proc. Black Hat Asia*, 2016, pp. 1–11.

[11] P. Deutsch. (May 1996). *DEFLATE Compressed Data Format Specification Version 1.3*. Accessed: Jun. 13, 2021. [Online]. Available: https://www.ietf.org/rfc/rfc1951.txt

[12] P. Ducklin, "Anatomy of a cryptographic oracle—Understanding (and mitigating) the BREACH attack," *SOPHOS Naked Secur.*, pp. 1–23, 2013. [Online]. Available: https://nakedsecurity.sophos.com/2013/08/06/anatomy-of-a-cryptographic-oracle-understanding-and-mitigating-the-breach-attack/

[13] M. Mimoso. (2013). *BREACH Compression Attack Steals HTTPS Response Secrets | Threatpost*. Accessed: Jul. 22, 2021. [Online]. Available: https://threatpost.com/breach-compression-attack-steals-https-secrets-in-under-30-seconds/101579/

[14] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, Oct. 1982.

[15] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.

[16] D. Karakostas, A. Kiayias, E. Sarafianou, and D. Zindros, "CTX: Eliminating BREACH with context hiding," Univ. Athens, Athens, Greece, Univ. Edinburgh, Edinburgh, U.K., Tech. Rep., 2016.

[17] J. Alawatugoda, D. Stebila, and C. Boyd, "Protecting encrypted cookies from compression side-channel attacks," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 8975, 2015, pp. 86–106.

[18] I. Sankalpa, T. Dhanushka, N. Amarasinghe, J. Alawathugoda, and R. Ragel, "On implementing a client-server setting to prevent the browser reconnaissance and exfiltration via adaptive compression of hypertext (BREACH) attacks," in *Proc. Manuf. Ind. Eng. Symp. (MIES)*, Oct. 2016, pp. 1–5.

[19] J. Alupotha, S. Prasadi, J. Alawatugoda, R. Ragel, and M. Fawsan, "Implementing a proven-secure and cost-effective countermeasure against the compression ratio info-leak mass exploitation (CRIME) attack," in *Proc. IEEE Int. Conf. Ind. Inf. Syst. (ICIIS)*, Dec. 2017, pp. 1–6.

[20] B. Paulsen, "Debreach: Selective dictionary compression to prevent BREACH and CRIME," M.S. thesis, Dept. Comput. Sci., Univ. Minnesota, Minneapolis, MN, USA, Tech. Rep., 2017.
[21] D. Karakostas and D. Zindros, "New developments on BREACH," in *Proc. Real World Crypto*. Stanford, CA, USA: Stanford Univ., Jan. 2016.
[22] M. Zieliński, "SafeDeflate: Compression without leaking secrets," Tech. Rep., 2020, doi: 10.13140/RG.2.2.13657.75363.
[23] B. Paulsen, C. Sung, P. A. H. Peterson, and C. Wang, "Debreach: Mitigating compression side channels via static analysis and transformation," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 899–911.
[24] P. Deutsch, *GZIP File Format Specification Version 4.3*, document RFC 1952, May 1996.
[25] H. Fischer, *A History of the Central Limit Theorem*. New York, NY, USA: Springer, 2011.
[26] S. G. Kwak and J. H. Kim, "Central limit theorem: The cornerstone of modern statistics," *Korean J. Anesthesiol.*, vol. 70, no. 2, pp. 144–156, Apr. 2017.
[27] T. T. Soong, *Fundamentals of Probability and Statistics for Engineers*. Hoboken, NJ, USA: Wiley, 2004.

RAFAEL PALACIOS was born in Madrid, Spain, in 1966. He received the B.S. and M.S. degrees in mechanical engineering from the ICAI School of Engineering, Comillas Pontifical University, Madrid, in 1990, and the Ph.D. degree from Comillas Pontifical University, in 1998.

He joined the Department of Electronics, ICAI School of Engineering, as an Assistant Professor, and the Institute for Research in Technology, as a Researcher, in 1998. He obtained, a Tenure, in 2004, and became a Full Professor, in 2020. He has been the Head of the Programs in telecommunications engineering and computer science, since 2012. He also helped to create the master's program in cybersecurity and was a coordinator, from 2019 to 2021. From 2001 to 2002, from 2009 to 2010, and from 2017 to 2018, he was a Visiting Professor with the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA. He carried out research with the Department of Aeronautics and Astronautics, Sloan School of Management, and the MIT Energy Initiative.

ANDREA FARIÑA FERNÁNDEZ-PORTILLO received the B.S. degree telecommunication engineering, in 2020. She is currently pursuing the double master's degree in telecommunication engineering and cybersecurity with the ICAI School of Engineering, Pontifical Comillas University, Madrid, Spain. She has been a Teaching Assistant at ICAI, since 2017. She was an Intern with the Institute for Research in Technology, in 2021.

EUGENIO F. SÁNCHEZ-ÚBEDA received the degree in industrial engineering (major in electronics) and the Ph.D. degree from Comillas Pontifical University, Madrid, Spain, in 1991 and 1999, respectively. He is currently a Research Staff Member with the Institute for Research in Technology, and a Senior Associate Professor with the ICAI School of Engineering, Comillas University, where he is also a Lecturer in statistics and machine learning. Since 1991, he has been involved in more than 80 research projects in collaboration with industry. His research interests include artificial intelligence, machine learning, forecasting, nonlinear statistical modeling, statistical learning algorithms, data visualization, and their application to energy markets.

PABLO GARCÍA-DE-ZÚÑIGA received the B.S. degree in telecommunications engineering and the M.S. degree from Comillas Pontifical University, in 2015 and 2017, respectively, and the M.S. degree in telecommunications engineering from the École supérieure d'électricité (currently CentraleSupélec), in 2017. He has experience in the cybersecurity sector as a Technical Governance, a Risk Consultant, and a Compliance Consultant for cybersecurity in the General Directorate of the Civil Guard, and as a Governance Expert with the Central Division of Cybersecurity, Siemens. Nowadays, he carries out his professional activity as an Information Security Manager at Allianz Direct, specializing in cloud-native environments and DevSecOps.