Uncovering Security Vulnerabilities in the Belkin WeMo Home Automation Ecosystem

Haoyu Liu, Tom Spink, and Paul Patras
School of Informatics, The University of Edinburgh, United Kingdom
Email: {s1783038@sms, tspink@inf, ppatras@inf}.ed.ac.uk

***Abstract*—The popularity of smart home devices is growing as consumers begin to recognize their potential to improve the quality of domestic life. At the same time, serious vulnerabilities have been revealed over recent years, which threaten user privacy and can cause financial losses. The lack of appropriate security protections in these devices is thus of increasing concern for the Internet of Things (IoT) industry, yet manufacturers' ongoing efforts remain superficial. Hence, users continue to be exposed to serious weaknesses. In this paper, we demonstrate that this is also the case of home automation applications, as we uncover a set of previously undocumented security issues in the Belkin WeMo ecosystems. In particular, we first reverse engineer both the mobile app that enables users to control smart appliances and the communication logic implemented by WeMo devices. This enables us to compromise the passphrase guarding the communication over the local wireless network, opening the possibility of eavesdropping on user traffic. We further reveal how an attacker can present a fake device to a WeMo user, through which cross-site scripting can be exploited in order to mislead the user into disclosing private information. Lastly, we provide a set of security guidelines that can be followed to remedy the vulnerabilities identified.**

***Index Terms*—Belkin WeMo, smart homes, security, privacy**

I. Introduction

The home automation scene is rapidly gaining in popularity, with more and more smart devices entering the market every year, which is expected to be valued at almost $80 billion by 2022 [1]. Up until recently, home automation systems were typically proprietary installations, with closed, wired, and centralized control. Originally, these systems were only specified for new-build homes, as retrofitting into existing buildings was complex and costly. However, with the advent of wireless and cloud technology, the significant reduction in sensing and computing costs, and the emergence of rapidly growing do-it-yourself (DIY) "hacker/maker" communities, Internet-connected home automation products have started to reach the market for all consumers, regardless of their housing type or technical knowledge.

Typically, these smart home products utilize an already existing wireless network (e.g. home Wi-Fi), connect to a cloud-based service (normally operated by the manufacturer), and enable their users to control various appliances, such as interior lighting, mains plug sockets, and ancillary systems such as burglar alarms, fire alarms, and door bells. As useful as these systems become, many companies are now competing in this relatively new niche. Unfortunately, in a rush to release products that provide new functionality and more convenience, coupled with a lack of rigorous understanding of embedded systems and network protocols, security and concern for user privacy have taken a back seat. Precisely, poor implementations can leave devices open to exploitation, not only permitting attacks on the objects themselves and jeopardizing their operation, but also side-channel attacks that leak private information, opening up further attacks on a user's personal network and their data.

In this paper, we focus in particular on the Belkin range of WeMo smart home devices. Belkin WeMo has become a market leader that commercializes smart sockets, light bulbs, video cameras, etc. that can be controlled with smartphone apps, or via personal assistants such as Amazon Alexa. We reveal that shortsightedness in the design of the pairing procedure these implement can lead to the leakage of a user's Wi-Fi passphrase. In addition, issues in the design of the mobile application can lead to phishing attacks, permitting further access (through social engineering) to a user's credentials. Although we focus on the Belkin WeMo ecosystem, there are certainly lessons to be learned by all manufacturers entering the home automation market. This leads to the following **key contributions**:

1) We reverse engineer the WeMo smartphone app, uncovering an exploit that enables to disclose the passphrase guarding the communication secrecy in users' home Wi-Fi networks.
2) We craft simple software to emulate a WeMo device, causing it to appear on the smartphone user interface, and then open up a cross-site scripting based phishing attack.
3) We discuss a number of mitigation approaches to address the vulnerabilities found in the Belkin WeMo range of smart devices, along with recommendations generally applicable to commodity home automation products.

Responsible Disclosure. Prior to submitting this paper, we have disclosed to Belkin all the security vulnerabilities identified. The company's engineering team has reviewed all our findings and are working on patches to address the issues we discovered.

II. Related Work

As the adoption of IoT technology accelerates, research identifying the security and privacy risks facing smart devices and services has intensified, spanning industrial IoT [2], home automation [3], and wearable gadget ecosystems [4]. In this section, we briefly review efforts that reveal weaknesses specific to Belkin WeMo devices, on which our work focuses.

Barcena and Wueest analyzed fifty connected home devices, including the WeMo switch, documenting the lack of authentication when connecting to such equipment and an attackers' ability to inject code [5]. Researchers at TwoSix labs demonstrated that it is possible to obtain bootloader-level console access to Belkin Wemo switches, via hardware

This research was funded in part by the UK National Cyber Security Center. The authors would like to thank the Belkin security team for their collaboration.

978-1-5386-9151-9/19/$31.00 ©2019 IEEE

exploitation [6]. Following this, the flash contents could be dumped and root access to the device becomes possible.

Exploits of the firmware update mechanism have been also documented. In particular, Buentello showed that arbitrary firmware could be uploaded to Belkin switches without authorization [7]. This prompted the release of official patches, following which new firmware updates must be signed. Unfortunately, the firmware-signing key later appeared to be included in the firmware itself; this makes it possible to extract the key and sign and push again malicious firmware to victim devices [8].

Due to the lack of message authentication in the Universal Plug-and-Play (UPnP) protocol implementation, Dhanjani shows that any users in the LAN can negotiate a `SmartUniqueID` with a WeMo baby monitor, and then use this to register on the remote server, which enables remote interception of the video stream [9]. The same approach is employed in [10] to facilitate permanent remote control of Belkin WeMo motion sensors and switches.

The feasibility of executing arbitrary JavaScript code within the mobile app by which users control WeMo devices was also demonstrated [11]. This enabled attackers to extract the location of a device and download pictures stored on the phone. Although this vulnerability was patched by Belkin with the WeMo Android app version 1.15.2, in this work we show that Cross-site Scripting (XSS) remains possible and an attacker can exploit this to trick the user into revealing sensitive personal information.

III. The Belkin WeMo Ecosystem

To understand the scope of our reverse engineering work, and where the vulnerabilities that we uncover are rooted, we first give a brief overview of Belkin WeMo system architecture and the communications model, which underpin the operation of this range of devices.

A. System Architecture

The range of automation devices we study include the WeMo Switch (a smart socket that can be used to control appliances), the WeMo Link (a controller for smart LED light bulbs), and the WeMo NetCam HD. The WeMo Smart Home devices considered are powered by the OpenWrt operating system for embedded devices,[1] which is an open-source Linux distribution extensively utilized as replacement firmware for wireless routers. OpenWrt provides over 3,000 standardized application packages, and allows manufacturers to customize their devices with specific functionality. Using OpenWrt as the embedded system greatly decreases the cost of development, and to some degree guarantees the security of the devices, in part due to regular firmware updates.

WeMo devices incorporate wireless communications modules that can connect to the user's home network via Wi-Fi. This enables consumers, after initial configuration, to interact with their smart devices directly using their mobile phones, instead of relying on additional bespoke controllers. In the initial configuration stage, a WeMo device opens a wireless access point in non-encrypted mode (a hotspot), which allows any smartphone to connect to it and configure the home network settings into the device. Once the configuration has been completed, the device will turn off the hotspot and automatically connect to the home network via Wi-Fi. We illustrate this procedure in Figure 1.

[1]OpenWrt - Wireless Freedom, https://openwrt.org/

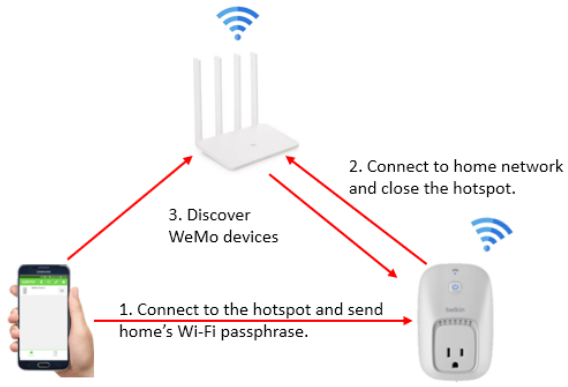

Fig. 1. Initial configuration of a WeMo device: (1) user connects to a hotspot spawned by the smart device, and transmits home network credentials; (2) device connects to home network and switches off hotspot; (3) user discovers and connects to WeMo device using mobile app.

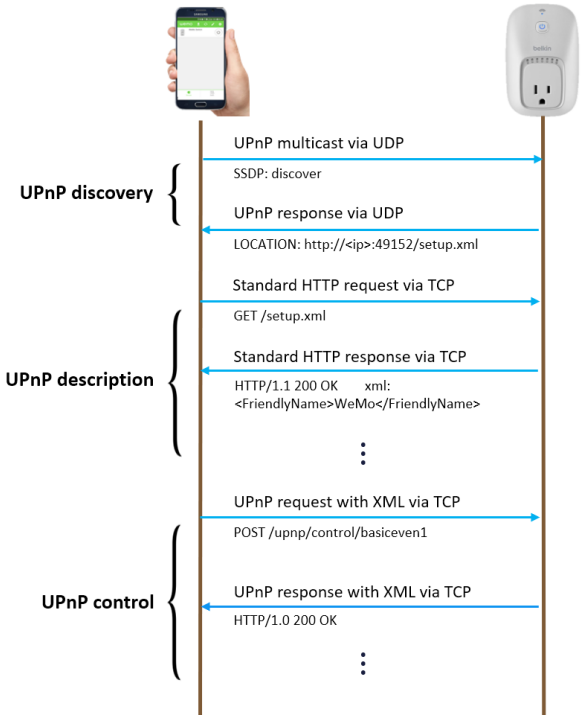

Fig. 2. Typical UPnP communication between WeMo mobile app and WeMo Switch. Device discovery performed over UDP, device description obtained through HTTP request/response, device control achieved through SOAP messages.

B. Communications Model

The communication between the WeMo mobile app and smart devices is built upon UPnP, a set of networking protocols including User Datagram Protocol (UDP), Hyper-text Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP), and Simple Service Discovery Protocol (SSDP). Each of these protocols are used for different purposes, but together implement the functionality required for device discovery, description, control, and eventing. UPnP control and eventing messages can only occur after the mobile app has obtained the UPnP description of a WeMo device. We illustrate a typical UPnP session between a smartphone and WeMo Switch in Figure 2 and discuss the different phases next.

1) Device Discovery: The process of locating smart appliances available on the Local Area Network (LAN) is referred to as device discovery and takes place every time a new device is connected. The discovery process starts with the mobile app sending an SSDP/UDP-multicast packet. This contains a field called supplied type (`ST`) that indicates the specific type of device or service, which the app is searching for. Any device that receives this packet (and matches the supplied type) will reply with its IP address, and the port on which the UPnP server is running. Figure 3 shows an example of real traffic generated by the mobile app and respectively a WeMo Switch device during the discovery stage, which we intercepted by running network packet capture on the home Wi-Fi access point. Note the device location returned at line 10, which will be used in subsequent messages.

```
1  M-SEARCH * HTTP/1.1
2  ST: urn:Belkin:service:basicevent:1
3  MX: 1
4  MAN: "ssdp:discover"
5  HOST: 239.255.255.250:1900
6
7  HTTP/1.1 200 OK
8  CACHE-CONTROL: max-age=86400
9  DATE: Sat, 01 Jan 2000 00:03:32 GMT
10 LOCATION: http://10.22.22.1:49152/setup.xml
11 OPT: "http://schemas.upnp.org/upnp/1/0/"
12 ST: urn:Belkin:service:basicevent:1
13 USN: uuid:Socket-1_0-XXX
```

Fig. 3. WeMo Mobile app ↔ Switch communication during discovery phase. App sends multicast message indicating type of device searched (line 2); matching device responds with IP address and port of UPnP server (line 10).

2) Device Description: After a WeMo device has been discovered, the mobile app will send standard HTTP requests to the UPnP server running on that device. Through such requests, the mobile app queries for the device's basic information, including device name, serial number, Medium Access Control (MAC) address, and supported services. This information is stored in several Extensible Mark-up Language (XML) files in the device's firmware, which are returned when a request is made. Figure 4 shows parts of the response from a WeMo Switch device sent during discovery (the actual contents of certain fields having been purposely altered/obfuscated for confidentiality reasons).

```
1 <friendlyName>WeMo Switch</friendlyName>
2 <serialNumber>221531K1000000</serialNumber>
3 <UDN>uuid:Socket-1_0-XXX</UDN>
4 <UPC>123456789</UPC>
5 <macAddress>XX:XX:XX:XX:XX:XX</macAddress>
```

Fig. 4. Snippet of device description returned by a WeMo Switch after a HTTP GET request. The values have been obfuscated for confidentiality reasons.

3) Device Control: After retrieving the list of services supported by a WeMo device, users are able to send control messages to query or alter the attributes and behavior of that device. Such a control message is formed as a SOAP message. SOAP is a protocol that largely relies on HTTP and XML, and defines a unique HTTP header called `SOAPAction`, indicating the intended service, which such a HTTP request will manipulate. An example SOAP message is shown in Figure 5, which is used to query for the device name (lines 5–7).

```
1 POST /upnp/control/basicevent1 HTTP/1.1
2 SOAPAction: "
      urn:Belkin:service:basicevent:1#GetFriendlyName"
3
4 <s:Envelope xmlns:s="http://schemas.xmlsoap
      .org/soap/envelope/" s:encodingStyle="http:
      //schemas.xmlsoap.org/soap/encoding/"><s:Body>
5 <u:GetFriendlyName
       xmlns:u="urn:Belkin:service:basicevent:1">
6 <FriendlyName></FriendlyName>
7 </u:GetFriendlyName>
8 </s:Body> </s:Envelope>
```

Fig. 5. Example of a SOAP-formatted UPnP control message, used to query the name of a device.

SOAP message bodies are encapsulated in XML, and in this case the request contains the specific action that the WeMo device should perform. Upon receipt of the message, the WeMo device will respond in a similar fashion, to inform the controller of the result of the action.

IV. Adversarial Model

Next, we discuss the capabilities that we expect an attacker to have in order to tamper with WeMo devices and identify a set of attack scenarios. With these in mind, we will discuss our reverse engineering efforts and reveal new ways in which the WeMo ecosystem can be exploited for nefarious purposes (Section V).

A. Attacker Capabilities

We assume the attacker is within wireless communication range of the victim's home infrastructure and has the necessary tools to sniff wireless traffic (e.g. the *Wireshark* network packet analyzer). With such tools, the attacker will be able to intercept victim's traffic during the smart device setup phase. We also expect the attacker has basic knowledge of the Android app ecosystem and is familiar with mobile app decompiling tools (e.g. *apktool*). We further assume the attacker has already downloaded a copy of the official WeMo Andoroid app. As such, the attacker may be able to identify vulnerable parts in the app code, which handle home Wi-Fi passphrase encyphering. Lastly, we expect the attacker is familiar with HTTP and UPnP, and can write simple code to send UPnP requests/responses, or uses specific open-source tools that reduce this implementation burden (e.g. *Flask*).

B. Attack Scenarios

We distinguish two main classes of attacks on the WeMo ecosystem: (1) causing unwanted stress and damage; and (2) gaining access to victim's home Wi-Fi network, to exfiltrate private data.

1) Causing Unwanted Stress and Damage: There is the possibility that third parties may wish to cause unwanted stress to the target user, or attempt their psychological manipulation, for instance following relationship breakups or to affect someone's decision making abilities through sleep deprivation and harassment; being able to take control of appliances would certainly enable this. Furthermore, an attacker may wish to interfere with household appliances, causing damage to them or to their users, e.g., by frequently switching on/off a smart device or activating it under unsafe conditions.

2) Unauthorized Network Access and Data Exfiltration: As we will show in Section V-B, by utilizing flaws in the WeMo device setup protocol, it would be possible to steal the passphrase guarding the confidentiality of the communications across a victim's home Wi-Fi network. To achieve this, an attacker could either emulate a "fake" device or resort to social engineering practices, e.g., send a "free" device to a target user. With this, a range of attacks with severe implications to user privacy could be mounted, as follows:

1) **Phishing:** A bait device could be used to trick a user into revealing personal information, such as the username and password for their WeMo account. Such details could be subsequently used to guess a victim's credentials for other services. Indeed, recent studies indicate 43-51% of users reuse the same password for different web sites, or employ simple transformations over the same password to derive others, which an attacker can easily guess [12].
2) **Data and Identity Theft:** Once a person's Wi-Fi passphrase is compromised, an attacker could deploy a packet sniffer configured on a promiscuous wireless interface, along with an automated script launching ARP spoofing between the default gateway and targeted users, to intercept all user traffic over the wireless network, which can be later inspected closely offline. This could further permit access to the home Wi-Fi access point and the deployment of Secure Socket Layer (SSL) stripping tools that compromise the confidentiality of HTTPS sessions [13]. Ultimately, this could result in exfiltration of sensitive data, including bank account information or personal details in view of fraud, and in worst case scenarios may lead even to identity theft.
3) **Access to Restricted Material:** Conceivably, the victim may be a government employee or the home automation infrastructure could belong to a competitor business. A user that can compromise the supporting Wi-Fi infrastructure by exploiting WeMo device on such premises, could subsequently access classified material or data with high commercial value.

V. Security Analysis

In this section we present our reverse engineering methodology that allows us to extract a home Wi-Fi passphrase and compromise user privacy. We then explain our efforts to emulate a fake WeMo device and subsequently launch XSS attacks to obtain personal data.

A. Reverse Engineering the Mobile App

Mobile applications ("apps"), and especially Android apps, are typically more easily accessible than embedded devices' firmware. If app developers neglect to apply adequate protections when compiling the source code to an `apk` file (an Android application package), others can easily decompile it, and obtain the majority of the source code. Compared with analyzing machine code, decompiled source code, due in part to its readability, enables security researchers to swiftly gain a deep understanding on the structure of the mobile app, and to start exploring its potential vulnerabilities.

In this study, we have utilized *apktool*, *dex2jar*, and *jd-gui* to decompile the WeMo Android App version 1.20.1. In this process, we confirmed that the WeMo app is developed on the Apache Cordova platform, which is an open-source mobile app development framework, and allows developers to use Hyper-text Markup Language (HTML), JavaScript, and Cascading Style Sheets (CSS) to design the user interface. Unfortunately, this also enables several common front-end attacks. A number of plugins are provided by the platform to help bridge front-end interactions with the mobile device's underlying functionality, such as file I/O, and network communications. By using *apktool*, we obtained the complete set of HTML, JavaScript and CSS files, and additional relevant configuration files.

```
1 <allow-navigation href="http://*" />
2 <allow-navigation href="https://*" />
3 <allow-intent href="http://*/*" />
4 <allow-intent href="https://*/*" />
5 <access origin="mailto:*" launch-external="yes" />
6 <access origin="*" />
```

Fig. 6. Whitelist configuration extracted from WeMo app `config.xml`. Launch of external URIs is permitted (line 5).

Through reading the configuration file `config.xml`, we discovered that WeMo developers have included a plugin (*cordova-plugin-whitelist*) that is capable of preventing Cross-Intent scripting and XSS attacks, by prohibiting unwanted Uniform Resource Identifier (URI) requests, even if the HTML and related JavaScript code may be vulnerable. However, as shown in Figure 6, WeMo developers misconfigured this plugin that allows both external and internal URIs, making possible the XSS attack that we later uncover. In order to gain a deeper understanding of the underlying functionality of this app, we harness the *dex2jar* tool to convert compiled `dex` files to `jar` files, and further used *jd-gui* to obtain most of the Java source code.

B. Home Wi-Fi SSID and Passphrase Leakage

Recall the communication model and the device setup stage shown in Figure 1. By packet analysis, we observe that the home Wi-Fi passphrase is sent to the WeMo device from the mobile in this phase, and this connection is unencrypted. We further find that a packet containing partially encrypted data enclosed by the XML tag `<password>` is sent to the mobile app at the end of this stage. Analysis of the decompiled Java code reveals methods involved in cryptographic packet generation. Although the decompiled code does not preserve parameter names, and the control flow is not 100% accurate, we are still able to understand the original encryption scheme, by observing several class names in these two methods and by making a few simple assumptions.

Figure 7 shows the pseudocode of WeMo's encryption scheme that we discovered. In particular, WeMo uses Password-based Encryption (PBE) to generate a cryptographic key for symmetric encryption before encrypting the Wi-Fi passphrase. A password is generated based on information that an attacker could infer (e.g. device's MAC and serial number). The password is used to derive parameters that are subsequently employed by industry-standard hashing and encryption algorithms, in order to encipher the Wi-Fi passphrase. Finally, the Wi-Fi passphrase is encrypted with a standard Advanced Encryption Standard (AES) algorithm, using the symmetric key generated in the previous step (line 6).

Finding: As this algorithm utilizes a device's MAC address and serial number to construct the symmetric key used for encrypting the Wi-Fi passphrase, and both MAC address and serial number will be transmitted unencrypted, anyone who intercepts

```
function encrypt(WiFiPassphrase):
  Password <- MACAddress
      [0:6] : SerialNumber : MACAddress[6:]
  Salt <- Password[0:8]
  IV <- Password[0:16]
  SymmetricKey <- MD5(Password, Salt)
  CiphertextBits <- AES_CBC_PKCS5Padding
      (WiFiPassphrase, SymmetricKey, IV)
  return Base64_Encode(CiphertextBits)
```

Fig. 7. Pseudocode of the encryption algorithm implemented by the WeMo app to encypher the Wi-Fi passphrase sent to WeMo devices at setup.

the traffic during the setup stage is capable of reconstructing the symmetric key, thereby obtaining the Wi-Fi passphrase in plaintext. In essence, the IoT device set-up procedure is insecure.

C. Emulating Fake Devices

Emulating a fake device involves listening for, and responding to UPnP requests. To accomplish this, we first intercept the network traffic between the mobile app and a WeMo Switch device, and collect several XML description files that the mobile app would request, as well as the responses to UPnP control messages. A UPnP server is in essence an HTTP server, since data packets are encapsulated in HTTP-style message. Therefore, in this study, we use *Flask*[2] (a light-weight web framework written in Python), to emulate the UPnP server found on WeMo devices. The *Flask* server broadcasts SSDP responses with its own local IP address. Once a mobile app sends an SSDP request to discover devices, the *Flask* server would respond to the mobile app and start communicating with it. Because only limited functionality was implemented on WeMo devices, it is not difficult to completely emulate a full device. Provided a subset of the communications model is implemented correctly – notably UPnP discovery, description and control – the mobile app would add a fake device to its list seamlessly, which opens the door to XSS attacks, as we discuss next.

D. Cross-site Scripting Attack

Previously XSS attacks [11] could be launched, because the mobile app did not implement appropriate input sanitization. The app extracts the required information (such as device name and MAC address) from the received XML file into a JSON string at the back-end, and then passes this string to the front-end, where the JavaScript implementation takes over for the following operations. A simple injection attack would be to alter the `FriendlyName` field, by inserting several quotation marks and a curly bracket to end the JSON string, followed by malicious XSS code.

We discovered that WeMo has patched this vulnerability with a very simple approach. Input sanitization was only partially implemented, by checking the existence of single and double quotation marks. If these are present in the `FriendlyName` field, the string inside the quotation marks (along with the quotation mark characters themselves) will be stripped, but the remainder of the string will be considered to be legal, and passed on to the subsequent processes. We could not recover the entire sanitization logic because the corresponding part of the source code proved unfeasible to decompile. However, after running several tests, we discover that the latest version of the mobile app likely only applies rules towards quotation marks, and no filtering rule for angle brackets is implemented (which should be considered as a necessity to prevent XSS attacks). Note that `FriendlyName` will be enclosed in HTML and displayed in the app. Due to the relatively permissive standards of HTML (and HTML parsers in general being quite forgiving), we successfully construct a basic XSS payload without using quotations, as shown in Figure 8.

```
<u:GetFriendlyNameResponse
     xmlns:u="urn:Belkin:service:basicevent:1">
<FriendlyName><script src=http://10.0.2.18
    :49152/ll/xss.js></script></FriendlyName>
</u:GetFriendlyNameResponse>
```

Fig. 8. XSS payload constructed without quotation marks.

This XSS attack may appear less dangerous than the previous documented one [11], as the WeMo developers have improved the permission requirements of the application. However, it is still possible to trick users' into revealing sensitive information, especially information related to Belkin, by making the malicious script jump to a phishing site. Because the user interface is actually inside an Android WebView component, which implements most of the functionality of a web browser (without the address bar), phishing becomes even more unnoticeable to the users. Figure 9 illustrates such an example, which is a fake Belkin login website displayed in the mobile app. It is not uncommon that people tend to reuse their passwords across several online services [12], thus this XSS attack exposes WeMo users to dangerous credentials theft risks.

Finding: A fake device can exploit insufficient input sanitization in tandem with lax HTML specification, to launch XSS scripting attacks that can facilitate phishing.

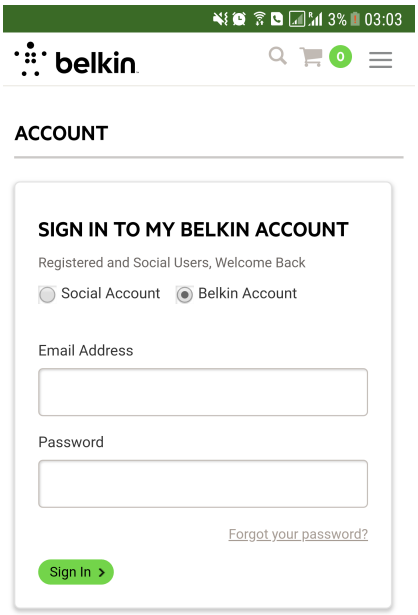

Fig. 9. Fake Belkin login webpage served by an emulated device that exploits XSS for phishing purposes.

[2]Flask - web development one drop at a time, http://flask.pocoo.org/

VI. Security Recommendations

Based on our security analysis, we make a number of recommendations to mitigate the vulnerabilities that we discovered.

a) Stricter encryption: Passphrase leakage should be in part attributed to the fragile encryption scheme that WeMo chose. Even if we did not reverse engineer the mobile app, it is still theoretically feasible to recover the encryption algorithm and obtain the symmetric key through brute-forcing. Hence, we recommend that WeMo should stop using the PBE with a device's MAC-address and serial number, to generate the cryptographic key. The Diffie-Hellman (DH) key exchange algorithm [14] could constitute a more robust alternative, if only aiming at mitigating this vulnerability at a software level. This is because passive packet interception would become insufficient for recovering the cryptographic key and subsequently decrypting the Wi-Fi passphrase. We note however that active man-in-the-middle attacks could still be used to compromise the DH algorithm. In general, without the presence of a certificate authority or pre-shared information between two parties, any cryptographic algorithm may be vulnerable to man-in-the-middle attacks during the initial handshake stage. Therefore, to completely solve this security issue, we would recommend a security by design approach, whereby a random Wi-Fi passphrase is generated for each product upon manufacturing and tagged on the product as the pre-shared key that will only be known to the user and product. This way, at the setup stage an attacker would be unable to decipher intercepted traffic, and subsequently compromise the communication secrecy on the home network.

At the time of writing, Belkin are implementing a different encryption algorithm to mitigate the issue we identified.

b) Source code obfuscation: Mobile apps may always contain important information regarding the program structure design and specific algorithms. In this study, we performed reverse engineering on the WeMo app and recovered the encryption scheme for the Wi-Fi passphrase. Code obfuscation appears crucial if developers wish to ensure hackers cannot easily decompile and read the source code. It is worth noting that code obfuscation should not be perceived as an impeccable protection against adversaries, since reverse engineering can still be performed on heavily obfuscated code if investing substantial amounts of time and human effort. However, this increases the cost of mounting attacks and we recommend this approach to be applied to strengthen the WeMo products. One common solution used for this is to rename all the classes, fields, and methods with meaningless names, so that understanding the general structure of the program or locating a certain snippet of code becomes challenging. For instance, ProGuard [15] is an efficient tool recommended by the Android official site, which could be used to shrink, optimize, and obfuscate Android code.

Following our disclosure to Belkin, their developers are implementing source code obfuscation.

c) URI filtering: Currently, one of the most common and efficient mechanisms of preventing XSS attacks is to set a Content Security Policy (CSP), which contains a group of HTTP headers to restrict unwanted cross-origin requests [16]. Cordova has a standard plugin, `cordova-plugin-whitelist`, that implements this functionality. However, although WeMo developers have included this plugin in the app, they did not properly configure it, therefore the latest version of the WeMo app can still send requests to arbitrary origins. We suggest that developers should re-configure this plugin, by only allowing requests for domains relevant to WeMo's remote access server.

d) Input sanitization: CSP is capable of prohibiting unexpected URL requests and is usually configured on the server side. However, this mechanism sometimes cannot be harnessed correctly by developers, especially when a web app needs plenty of cross-origin resources and developers have to set relatively general rules. More importantly, this mechanism itself cannot eliminate XSS injection points, but only restrict requests. Uber's website was once vulnerable to a XSS attack in spite of the existence of CSP headers [17]. XSS could be triggered due to a latent injection point and relatively permissive CSP rules. Hence, it is still necessary to implement input sanitization for the purpose of fully eliminating possible injection points. We recommend that WeMo developers, and in general developers of mobile apps for IoT device control, use an open-source XSS-sanitization module, such as `js-xss` [18], to accomplish this.

Belkin are addressing the last two points through a UPnP implementation update, which will include a secure element to be required by firmware and apps for local communication.

VII. Conclusion

In this paper we undertook a security analysis of the Belkin WeMo ecosystem. We reverse engineered the official mobile app and communication protocols, which enabled us to uncover how home Wi-Fi passphrases can be leaked. We further showed how an attacker can emulate a fake device, and continue to exploit XSS to lure the user into disclosing personal information. Based on our findings, we gave a set of recommendations that can help mitigate the vulnerabilities identified and prove useful to the home automation industry at large.

References

[1] Research and Markets, "Home automation system market - global forecast to 2022," April 2017.

[2] Q. Yan *et al.*, "A Multi-Level DDoS Mitigation Framework for the Industrial Internet of Things," *IEEE Comm. Mag.*, vol. 56, pp. 30–36, Feb 2018.

[3] P. Morgner *et al.*, "Insecure to the Touch: Attacking ZigBee 3.0 via Touchlink Commissioning," in *Proc. ACM WiSec*, pp. 230–240, 2017.

[4] J. Classen *et al.*, "Anatomy of a Vulnerable Fitness Tracking System: Dissecting the Fitbit Cloud, App, and Firmware," *ACM IMWUT*, March 2018.

[5] M. B. Barcena and C. Wueest, "Insecurity in the Internet of Things," *Security Response, Symantec*, 2015.

[6] J. Tanen, "Breaking BHAD: Getting Local Root on the Belkin WeMo Switch," Apr 2016.

[7] D. Buentello, "Belkin Wemo - Arbitrary Firmware Upload," Apr. 2013.

[8] M. Davis, "Belkin WeMo Home Automation Vulnerabilities," July 2014.

[9] N. Dhanjani, *Abusing the internet of things: blackouts, freakouts, and stakeouts*. O'Reilly Media, Inc., 2015.

[10] V. Sivaraman *et al.*, "Network-level security and privacy control for smart-home IoT devices," in *Proc. IEEE WiMob*, pp. 163–167, Oct. 2015.

[11] S. Tenaglia, "Breaking BHAD: Injecting Code into the WeMo App Using XSS," Apr. 2016.

[12] A. Das *et al.*, "The tangled web of password reuse.," in *NDSS*, 2014.

[13] J. Clark and P. C. van Oorschot, "SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements," in *IEEE Symposium on Security and Privacy*, pp. 511–525, May 2013.

[14] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[15] E. Lafortune, "ProGuard." https://sourceforge.net/projects/proguard/, 2018.

[16] W3C, "Content Security Policy Level 3." https://www.w3.org/TR/CSP3/, Sept. 2016.

[17] StamOne, "DOM XSS - auth.uber.com," Oct. 2017.

[18] Z. Lei, "XSS - Sanitize untrusted HTML with a configuration specified by a Whitelist," 2018.