Attacking TP-Link Suite of IoT Devices (Plugs and Bulbs)

Shamus Neo Zhi Kai
School of Computing
National University of Singapore
e0310778@u.nus.edu

Sashankh Chengavalli Kumar
School of Computing
National University of Singapore
e0129255@u.nus.edu

Ling Zhi Yu
School of Computing
National University of Singapore
e0203418@u.nus.edu

Shen Jia Min
School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University
shen_jiamin@outlook.com

Djambazovska Sara
Swiss Federal Institute of Technology Lausanne (EPFL)
sara.djambazovska@epfl.ch

ABSTRACT

Internet of Things (IoT) devices are interconnected together by various network protocols, with each device performing its own functionality - which may encompass different supporting technologies - as part of an IoT ecosystem. The ease of IoT devices extending the functionalities of objects used daily in systems such as in homes and workplaces has spurred the popularity of IoT devices worldwide. However, as a newly-developed area, the security of IoT devices has yet to have been scrutinised sufficiently, and this leaves increasing numbers of whole IoT ecosystems vulnerable to attacks. In this paper, IoT products from TP-Link, a popular consumer Wi-Fi device producer, are examined closely for vulnerabilities and assessed for possible feasible improvements. In particular, a replacement encryption scheme is proposed and TP-Link's authentication mechanisms are relooked. Additionally, a proof of concept (POC) web interface is developed to showcase the experiment's findings of controlling the IoT devices found on the same network.

Categories and Subject Descriptors

C.2.0 [Computer-Communication Networks]: General — Security and Protection
C.2.2 [Computer-Communication Networks]: Network Protocols

General Terms

Security

Keywords

Kasa Smart, TP-Link LB120 Wi-Fi Smart LED Bulb, TP-Link HS110 Smart Plug, Internet of Things

1. INTRODUCTION

The Internet of Things (IoT), is a system of interrelated devices with the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction [1]. With the capability to connect to the internet, IoT devices can extend the functionality of objects used daily in systems such as in homes and workplaces, and this has spurred the popularity of IoT devices worldwide. As IoT technology makes advancements, the prevalence of IoT devices incorporated in homes and industries like healthcare and hospitality will only increase. However, as the IoT industry is relatively young, there are concerns that the security of IoT devices have not been rigorously examined, and that growing IoT usage will add more vulnerabilities into IoT ecosystems.

Unlike conventional wireless networks, which may have dynamic topologies, IoT devices are deployed on Low Power and Lossy Networks (LLNs) and are constrained with limited processing power and memory [2]. Thus, confidentiality and authentication in IoT are secured by lightweight cryptographic mechanisms, which do not provide the same encryption robustness as schemes used in conventional networks and therefore make IoT networks vulnerable to cryptographic attacks.

Moreover, the increasing deployment of IoT devices used in corporate and home networks has increased the attack surface hackers can exploit drastically, making the security of these devices paramount.

2. MOTIVATIONS

It is known that the first-generation US versions of IoT devices produced by TP-Link, a top provider of consumer Wi-Fi devices worldwide, contained multiple reported vulnerabilities in their devices. Although the vulnerabilities were fixed in the subsequent versions, there is a lack of proper investigation into the devices with the updated UK v2.1 firmware that are available in our region. A report released by Softcheck on the previous version highlighted some vulnerabilities [3]. The objective of this project was to investigate the upgraded versions for these and other vulnerabilities. In addition, we aimed to develop a user interface to

scan the network for vulnerable devices and possibly mount such attacks.

The security of products by TP-Link are examined as a yardstick for security standards in consumer IoT devices. In particular, the TP-Link LB120 Smart LED Bulb and HS110 Smart Plug (both UK v.2.1) were selected for examination as they were supposed to be patched, instead of the older and more vulnerable versions which have already been widely reported previously. The protocols used in communications in the bulbs, TP-Link Smart Home Protocol (TSHP) and TP-Link Device Debugging Protocol (TDDP), were analysed for potential vulnerabilities, with the former used in controlling IoT devices via TP-Link's phone app, Smart Kasa, and the latter used as a testing protocol across all TP-Link devices.

In this paper, the security of TP-Link IoT devices is evaluated through attacks developed with information gleaned from reverse-engineering the devices' affiliated software: via decompiling the Smart Kasa APK and the studying the devices' firmware. Upon successfully hacking and gaining control of the devices, the insight gained from exploiting the devices' vulnerabilities were subsequently used in the development of feasible solutions to improve the security of IoT devices. To demonstrate the experiments' findings, a POC web interface was developed to showcase control over TP-Link's IoT devices and implementations of viable security augmentations for IoT devices.

3. METHODOLOGY

With multiple possible points of entry, various approaches were taken to perform reconnaissance and to compromise the devices.

3.1 Reverse Engineering of TP-Link HS110 Smart Plug Firmware

3.1.1 Obtaining the Firmware

While the older v1.0 (US) firmware was available on TP-Link's website, the firmware for v2.1 (UK) was not published online. Thus, in order to commence reverse engineering, the updated firmware had to be acquired via other methods.

One of the feasible ways to obtain the firmware and other related information was to directly retrieve it from the flash chip in hardware itself serially. Figure 1 below is reproduced from a previous investigation of the hardware by [4].

Given that the pinouts were labelled on the printed circuit board (PCB), we were confident that the firmware was accessible via tinkering with the hardware through means such as JTAG or UART. However, this method was only to be considered as a last resort as it would risk spoiling the hardware should possible mistakes be made.

Subsequently, a direct approach for the firmware to TP-Link's direct customer support live chat was made, as seen in Figure 2. While they were initially hesitant to provide the firmware without formal disclosed reasons, our continued persistence resulted in them relenting and providing an alternative installation tool that contained the firmware of the devices.

Figure 1. PCB of HS110 Smart Plug [4]

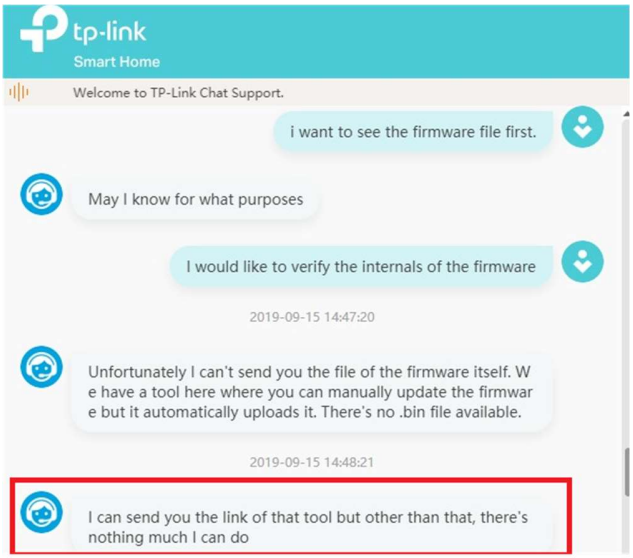

Figure 2. Conversation with TP-Link Customer Support Live Chat

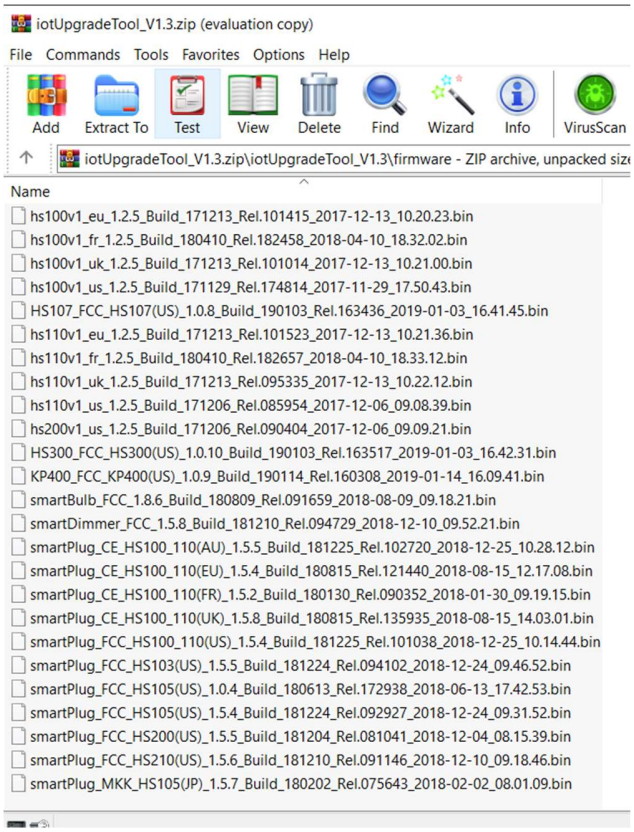

Figure 3. Contents of received tool

Upon examining the tool (Figure 3), it was found that the tool contained the v2.1 (UK) firmware for both the HS110 Smart Plug and the LB120 Smart LED Bulb. While the firmware was unavailable online and not permitted to be released by TP-Link Customer Service, we were still able to obtain the files required for reverse-engineering through social engineering – by getting the customer service officer to unknowingly give access to the firmware.

3.1.2 Commencement of Reverse Engineering

Firstly, the *binwalk* tool was used to extract the firmware from the binary files via the command:

```
binwalk -e
smartPlug_CE_HS100_110(UK)_1.5.8_Build_18081
5_Rel.135935_2018-08-15_14.03.01.bin
```

From the extracted content, a stripped-down version of a Linux file system, *SquashFS*, was discovered in HS110 firmware. We were able to deduce that the plug was running off *BusyBox,* a low-energy consumption operating system, which is commonly found in low-powered devices [3].

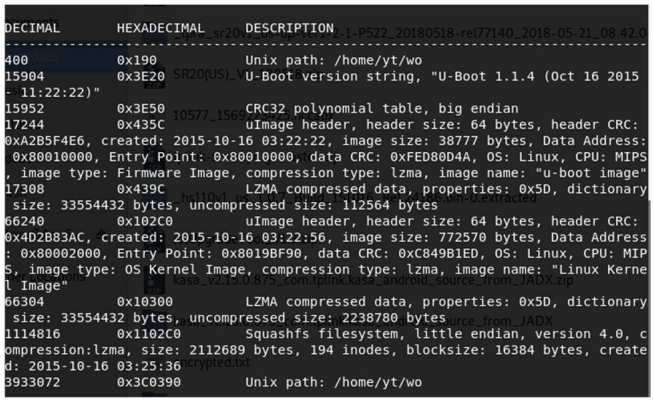

Figure 4. Contents of firmware extracted from binary

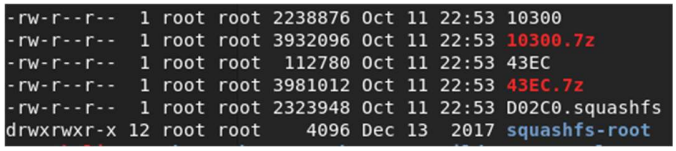

Figure 5. Contents of HS110 firmware binary

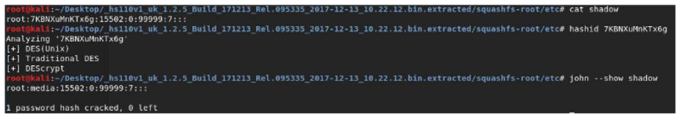

Figure 6. Screenshot of cryptanalysis of */etc/shadow* folder of *squashfs-root* file system

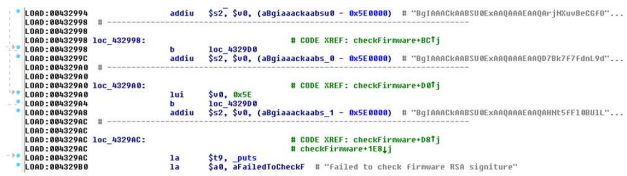

Figure 7. Assembly code detailing checks against hardcoded RSA values [3]

Next, the contents of /etc/shadow of the *squashfs-root* filesystem were inspected, and the password of the root user was found encrypted and not hashed – and therefore vulnerable to cryptoanalysis. Eventually, the password was cracked and discovered to be "media".

Subsequently, analysis with the *IDA Pro* disassembler was done on the `/usr/bin/shd` folder, which contains the main server application and the information for HS110's major functionalities. It was discovered that the firmware's RSA signature is checked against hardcoded values (Figure 7), which indicated the impracticality of flashing a custom firmware [6] into the plug.

3.2 Patents for TP-Link Proprietary Debugging Protocols

While TP-Link's patents documented that custom packets could be designed to issue commands to the plugs, the newer versions of V2.1 (UK) device firmware have since disabled the debugging protocols [7].

After multiple failed attempts to enable the debugging protocols through port triggering - sending packets to all closed ports to a device to "trigger" the opening of specific ports - and modifying the firmware, we decided to redirect our efforts on other potential entry points.

3.3 Decompiling TP-Link Kasa Smart App

The Kasa Smart Android application could be decompiled with the help of online tools or Kali-Linux's *apktool* to provide a way of viewing its source code.

```
public static byte[] m7377b(byte[] bArr) {
    if (bArr != null && bArr.length > 0) {
        int i = -85;
        for (int i2 = 0; i2 < bArr.length; i2++) {
            byte b = (byte) (i ^ bArr[i2]);
            i = bArr[i2];
            bArr[i2] = b;
        }
    }
    return bArr;
}
```

Figure 8. Source code from decompiled Kasa Smart App [3]

From Figure 8, it was evident that little to no code obfuscation was done, as most of the Java code was still readable – class names, methods and variables were all intact.

Further analysis revealed a section of code that indicated how data encryption was done before it was sent on the network. Commands were encrypted with an autokey cipher with a hardcoded 1-byte key, likely because embedded IoT devices are constrained by limited power, memory and processing power. Thus, PKI or elliptic curve cryptography would be too computationally intensive and infeasible. This discovery indicated the feasibility of exploiting the weak encryption schemes and lack of authentication used by the bulb and plug – to craft rogue commands and control the devices once the encryption was broken via cryptanalysis.

3.4 Sniffing Traffic Between Application and Smart Plug

TCP and UDP stealth scans were conducted with *nmap* to scout for open ports on the smart plug.

From Figure 9, results from the scan revealed that only port 9999 was open. It was then evident that the plug only received commands at port 9999, through the TP-Link SmartHome Protocol.

Subsequently, the traffic at port 9999 between the Kasa Smart application and the plug was captured with Wireshark. With cryptanalysis of the autokey cipher, a Lua script was created to enable Wireshark to decrypt the cipher in real time.

```
Starting Nmap 7.80 ( https://nmap.org ) at 2019-10-11 23:33 +08
Nmap scan report for HS110 (192.168.1.109)
Host is up (0.033s latency).
Not shown: 999 filtered ports
PORT     STATE SERVICE VERSION
9999/tcp open  abyss?
MAC Address: 50:D4:F7:47:CA:FB (Tp-link Technologies)
```

Figure 9. Results from *nmap* network scan

Source	Destination	Protocol	Length	Info
192.168.1.136	192.168.1.109	TCP	74	35910 → 9999 [SYN]
192.168.1.109	192.168.1.136	TCP	58	9999 → 35910 [SYN,
192.168.1.136	192.168.1.109	TCP	54	35910 → 9999 [ACK]
192.168.1.136	192.168.1.109	TPLINK-SMARTHOME	87	35910 → 9999 [PSH,
192.168.1.109	192.168.1.136	TCP	54	9999 → 35910 [ACK]
192.168.1.109	192.168.1.136	TPLINK-SMARTHOME	684	9999 → 35910 [PSH,
192.168.1.136	192.168.1.109	TCP	54	35910 → 9999 [ACK]
192.168.1.136	192.168.1.109	TCP	54	35910 → 9999 [FIN,
192.168.1.109	192.168.1.136	TCP	54	9999 → 35910 [ACK]
192.168.1.109	192.168.1.136	TCP	54	9999 → 35910 [FIN,
192.168.1.136	192.168.1.109	TCP	54	35910 → 9999 [ACK]

Figure 10. Network traffic intercepted at port 9999

From Figure 10, the decrypted traffic revealed that the commands sent to the plug adhered to a JSON format. Thus, with the knowledge of encryption algorithm and the format of the commands, sufficient information was obtained to craft rogue packets to control the TP-Link devices.

When tested, the crafted packets sent to the plug were able control the behaviour of the plug, and this validated that the plug did not perform authentication on received commands.

4. ATTACKER INTERFACE

The topology shown below demonstrates the possible scenarios where an attack can come from both the internal and external network.

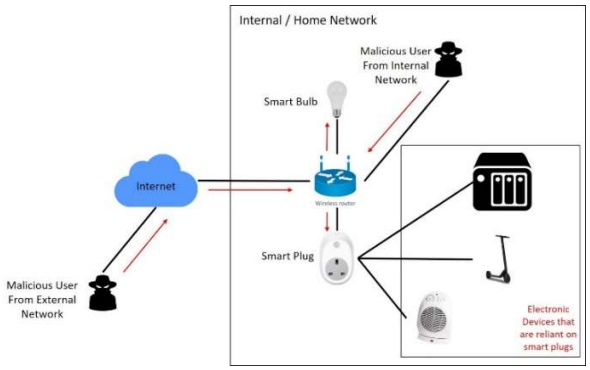

Figure 11. Network topology depicting possible attack scenarios

If the router is configured to port forward traffic for port 9999 to the smart devices, hackers are potentially able to send malicious crafted commands into the internal network. However, the chances of a normal user manually configuring port forwarding is low therefore, this is unlikely to happen. Nevertheless, given that some routers and network devices still turn on Universal Plug and Play (uPnP) where the device automatically tell the router to perform any Port Forwards that they require [8] by default, an attack can still happen.

This report focuses more on how attacks can happen from the internal network which is assumed to be trusted, but the concepts are similar and can be applied to attacks coming from the external network as well since the commands are not authenticated.

In order to demonstrate this ability, we developed a web interface that allows us to discover vulnerable devices on the network, and select a device to attack. This interface may also be used to discover vulnerable devices for defence.

4.1 Construction

The application was developed using Node.js and ReactJS, and uses Python and bash scripts to post JSON payloads to the vulnerable devices. The interface is intended to be used on a device connected to the local network to which the device is linked, in order to enable discovery of the device and TCP communication with it.

4.2 Exploits

4.2.1 Plug

Smart plugs can be used to control the power supply to any device, and may be used to control these devices remotely. The ability for malicious actors to control the supply is particularly harmful, and is discussed further in Section 5.

To demonstrate the degree of control and potential damage this can cause, we allow the user to turn on/off the power supply, and rapidly toggle the power supply to the plug. This shows the ease with which this vulnerability can be exploited.

4.2.2 Bulb

Similar to the plug, the bulb can also be controlled from the web interface. In the case of the bulb, we are able to control the brightness and the colour temperature, offering virtually complete control over the lighting. This degree of control for any host on the network is extremely intrusive, and should not be possible. We allow users to specify any arbitrary sequence of controls, and implemented the ability to remotely flash a message using morse code from the web interface.

5. IMPACT OF EXPLOITS

The potential impact of these exploits of IoT devices like smart plugs are massive.

For example, many consumers use these smart plugs to control the duration that power is supplied to the devices that are connected to these plugs throughout the day. For example, automatically shutting off a charger that is charging the battery of a Personal Mobility Device (PMD) to prevent overcharging and causing a fire. Upon successful exploitation of the smart plug, one can potentially turn on the plug continuously, supplying power to the charger and overcharging the battery which might lead to a battery or electrical fire.

Others might use a smart plug to track power consumption for devices that are turned on 24 hours such as network attached storage devices. Upon successful exploitation of the smart plug, one can potentially turn off the plug, causing the network attached storage to turn off abruptly which might potentially lead to data loss or corruption if there are no redundant power supply attached to it.

Given that the smart plug can scan for other wireless networks, one can also use the smart plug as a proxy to perform further reconnaissance and wardriving to compromise even more Wi-Fi networks.

6. ENHANCING DEVICE SECURITY

Directly following from the previous section, it is apparent that the impact of vulnerability can be far-ranging and can include loss of property, life. Hence, it is important to decide on a clear security definition and condition the choice of protocols and software architecture on this definition.

6.1 Review the Trade-offs

The security definition of the protocols used by such IoT devices must be strong enough to prevent the misuse of this system by other parties and prevent the leakage of user information. Primarily, the control of IoT devices over the network must prevent, at the very least, denial of service attacks (DoS), control hijacking and usage monitoring by unauthorized hosts. Authentication and encryption schemes, when applied suitably can be used to satisfy these security definitions.

A primary concern with IoT devices is the limitation of CPU power and memory. The use of stronger protocols is of paramount importance in the face of the potential damage that these vulnerabilities can cause, even at the cost of a slightly higher latency in response to commands.

6.2 Improving Encryption

Some immediate failings are notable in the use of encryption schemes in this device. These are particularly worrisome given that issues apparent in previous versions of the firmware were not patched. The encryption scheme used in this device is an autokey cipher, with a hard-coded secret. Autokey ciphers are simple encryption schemes that are notoriously vulnerable to many known cryptanalysis methods, such as plaintext-only attacks with dictionaries. However, in this case, the key size was also small

enough for a simple brute force attack to be successful in very little time.

In order to investigate the possibility of using alternative encryption schemes, we experimented with running a set of algorithms and obtaining a rough comparison measure of their execution time on these devices. Without access to a similar processor, we were not able to time the encryption on a comparable processor, and flashing the device with rewritten firmware was not possible. However, we could measure the average elapsed time by testing these algorithms on an average CPU, and by using the MIPs (Millions of Instructions Per Second) performance metrics, estimate their run time on the target device's core. MIPs is a general benchmark of how many instructions a processor can handle in a single second.

The processor incorporated in TP-Link HS110 Smart Plug is MIPS32 24K core, a high-performance, low-power, 32-bit MIPS RISC core designed for custom system-on-silicon applications [12].

Its performance peaks can reach up to 625 MHz, giving the highest frequency available in 32-bit synthesizable cores for embedded systems. The official MIPs specifications [15] provide figures showing an execution of 604 MIPs for this core, at its average 400 MHz speed. For the four core processors used to run the experiments, these numbers are predictably greater, namely, the Intel Core i7 7500U performs 49,360 MIPs at 2.7 GHz. These details together with the runtime test results obtained, allowed for an approximation of the number of instructions that are executed while running these encryption algorithms. This was further used to estimate the execution time of the same programs on the slower CPU. Calculations showed execution time overhead of these algorithms of around $\frac{MIPs_A}{MIPs_B} \rightarrow 81.72185$ times on the RISC core.

$$\frac{49360 \times 10^6 \times Ic}{sec} \times execution\ time = Ic$$

$$MIPs_A = \frac{Ic}{Execution\ time_A \times 10^6}$$

$$Ic = MIPs_A \times Execution\ time_A \times 10^6$$

$$Execution\ time\ _B = \frac{Ic}{MIPs_B \times 10^6}$$

$$Execution\ time_B = \frac{MIPs_A}{MIPs_B} \times Execution\ time_A$$

$$where\ A = Intel\ Core\ i7\ 7500U,$$
$$B = MIPs32\ 24k\ core$$

Table 2 below compares the run times in each of the different implementations, tested on a 2.7 GHz Intel Core i7 7500U processor for 1000 iterations of encrypting a 42-byte string. We used simple calls to get the current millisecond clock time value and subtracted the start time from the end time of the algorithm running for 1000 iterations, to calculate elapsed milliseconds. After getting the elapsed time, we could calculate the average time per encryption call, by dividing the total elapsed time by the number of iterations. Having these results, and the MIPs performance specifications of both the testing and the target CPU, we could approximate the execution time on the actual IoT device.

However, it should be noted that the MIPs benchmark is proven most accurate when comparing processors with a similar architecture, since some architectures may require substantially more instructions than others for running the same program. Due to a lack of a comparable MIPS processor, we could only rely on these variations not being too severe for our case.

The first alternative we experimented with was the use of the Hummingbird2 Authenticated Encryption schemes [20]. Even out-of-the-box implementations were only tolerably slower than the autokey cipher and are more secure. On an average processor, it took about 1.64ms per encryption call using Hummingbird2, leading to an estimated execution time of 0.13s on the slower core. Taking into consideration the security enhancement, the latency trade off seems tolerable. However faster mechanisms, with the same or better security guarantees do exist, so we continued by exploring different alternatives, focusing on symmetric key block ciphers.

The most commonly used symmetric key cryptosystem in securing IoT devices, is the Advanced Encryption Standard (AES), standardized by the NIST (National Institute of Standards and Technology). The AES is based on a SPN (Substitution–Permutation Network) structure, encrypting 128-bit blocks with a key of length 128, 192 or 256 bits. In our experiments we focused on the case of AES-128, using a 128-bit key. This cipher version consists of 10 rounds repeating four elementary functions, SubBytes, ShiftRows, MixColumns and AddRoundKey, on blocks interpreted as 4 x 4 matrices. Its large keyspace makes it immune to brute-force attacks, and its simplicity allows for the use of hardware acceleration. The implementation of this algorithm showed satisfying results, namely an estimated average execution time of around 5 ms, while offering one of the highest security available, with the best attack known being only slightly better than brute force [17].

Experimental results in [13] from encryption implementations in different IoT applications have shown that using the AES algorithm assisted with a hardware accelerator is most energy efficient and is ideal when timing is a major constraint, whilst the XTEA algorithm is ideal for resource/area constrained microcontrollers.

This led us to proceed with our experiments by testing the XTEA block cipher. Designed as an extension of TEA (Tiny Encryption Algorithm) [14][16], it remains as one of the smallest and simplest algorithms, yet fastest and most efficient cryptographic schemes. XTEA's simplicity can be demonstrated by the fact that the encryption and decryption algorithms can be represented in seven lines of C code each. This block cipher is based on a Feistel network with a block size of 64 bits and a 128-bit key. The internal F-function is really simple and composing only of bit shifts, XORs and additions. Running this concise algorithm lead to an estimated execution time of 0.045 s per encryption on the Smart Plug.

ChaCha20 is a high-speed stream cipher developed by Daniel J. Bernstein [19], as a variant of the Salsa20 cipher. It is based on ARX operations (modular addition, rotation and XOR), with 20 rounds, operating with key sizes of 128 and 256 bits. To authenticate the encrypted messages, ChaCha20 is generally used with MAC (message authentication code) Poly1305. It operates by creating a key stream which is then XORed with the plaintext. It has been standardised with RFC 7539 [18]. The execution time of this algorithm while using a key length of 256 bits with a 128 bit nonce, was ranked best among the algorithms tested, with an overhead of only 2.5 ms per encryption compared to the autokey cipher.

In order to establish a shared secret key to be used for encryption by the symmetric key algorithms analyzed above, we considered using elliptic curve cryptography. Given its features like low power, lightweight and robustness, the emerging Elliptic Curve Diffie Hellman(EC-DH) is a reasonably secure key generating algorithm for these IoT gadgets. In ECDH, all the computations are done on a fixed elliptic curve, namely, the short Weierstrass curve defined by

$$y^2 = x^3 + ax + b$$

In our tests we used the NIST recommended curve P-384, also known as secp384r1. Defined over the field Fpn, where pn is a 384 bits long prime, it provides 192 bits of security level, as recommended by NIST. The value of the b coefficient is provided by NIST, and in order to minimize the field computations, the value of a is chosen to be -3. The prime used for P-384 is $2384 = 2128 - 286 + 232 - 1$. The size of the keys and primes used in ECDH is given in Table 1. By using the "Double and Add" algorithm [9], the only operations performed on points defined on the elliptic curve, are point addition and point doubling. Using the results found in [10] we can approximate the power consumption for the different cryptographic operations while using this algorithm. The key generation consumes about 26 Watts, and 0.8 Joules of Energy. We have approximately the same power and energy consumption for generating the public key and the shared secret key. From the specification of the TP-Link HS110, we get the maximum power consumption for this device is 3.68KW, so the key generation would only take insignificant 0.68% of it.

Furthermore, we approximated the average time for executing this algorithm, obtaining an AES-128 encryption after successful shared key generation. The latency on a 2.7 GHz processor compared to the autokey encryption was of around 1,34 ms per encryption, including the key generation. By using the specifications of the processing power of the target device, we could approximate an execution time of 0.11s per iteration of the algorithm. The shared key generation takes the largest share, with 0.1059s.

Considering the security enhancement that is gained with 192 bits of security, and the fact that key generation needs to be done only once, removing the time overhead required for it in all subsequent data sharing, we found this encryption acceptable.

	Prime	Private Key	Public Key	Shared Key
P-128	256	256	512	256
P-384	384	384	768	384

Table 1. Sizes of keys and the primes used in ECDH

Cipher	Average encryption time in ms	
	Intel Core i7 2.7 GHz	MIPs32 2Kc 400 MHz
Autokey	0.015625	1.2769
ChaCha20	0.046875	3.8303
AES-128-CBC	0.0625	5.1076
XTEA	0.5625	45.9685
ECDH with AES128	1.359375	111.0902
Hummingbird2	1.640625	134.0745
ECDH Shared Key Generation	1.29687	105.9826

Table 2. Average execution time per encryption of a 42-byte string on the Intel Core, compared to MIPS32

Other lightweight cryptographic techniques exist that can effectively deal with different security attacks, including the one described in this paper, as they are suitable for resource-constrained IoT devices. Such defence measures should aim at improving the security of the IoT systems while maintaining their employability. The study [11] compares different LWC Algorithms with the classical AES encryption method. The results show the existence of lightweight options like LBlock, LED, Piccolo, PRESENT and TWINE, offering a high security level of up to 84% while having very short key size of 80 bits and a 64 bit block size.

A study of similar performance evaluations of different symmetric key algorithms was done in IoT devices [13], and the tested symmetric key block ciphers were AES, RC6, Twofish, SPECK, and LEA in GCM (Galois/counter) mode with all supported key sizes (128, 192 and 256 bits). Using these results, we can see that there are good encryption solutions for resource-constrained IoT devices.

6.3 Alternative Authentication Methods

6.3.1 Avoid setting default hardcoded credentials

As discussed in Section 3, HS110 Smart Plug used hardcoded RSA values in its embedded firmware to prevent flashing and also used a default password for the *Squashfs-root* file system, which is indicative of "CWE-798: Use of Hard-coded Credentials" [21]. This practice leaves multiple areas for exploitation by attackers, given that once the credential is cracked, its value may be easily found online and used by more attackers. Further, since the source code is embedded and shipped along with the device, the value of the hardcoded credential is likely to be discovered by reverse-engineering attempts. Once a particular device's credential is discovered and exploited, it follows that all other manufactured products of the same model would be vulnerable to the same exploit, effectively weakening the layer of security which the credential was supposed to provide. By issuing hardcoded credentials in IoT devices, whole IoT ecosystems may be brought down and its impacts are amplified by the sheer number of exploited IoT devices, as observed in the Mirai Botnet attack [22].

To mitigate the risks of using hardcoded credentials, several credential management solutions are available and may be integrated during the software architecture and design phase [21]:

1. A first login mode may be set in the Kasa Smart application, forcing a password requirement for users to set a new and strong password on the device.
2. Store passwords and credentials outside of the code in secured and encrypted files or strongly protected databases, to limit access from foreign and local users,
3. For credentials that must be hardcoded, the access control of these entities should be limited, only enabling access control for necessary features.
4. Strong one-way hashes should be used to stored passwords in secured configuration files or databases with only required access control.

6.3.2 Application ID Authentication through OpenID Connect

There is presently no authentication of commands received by the devices, presumably because of the constraints in the devices' processing power and memory. As a form of lightweight authentication, a mechanism for application ID authentication may be done on the device. A relevant study regarding secured authentication for IoT Applications proposed applying OAuth2.0, an industry-standard authorisation protocol, to perform application ID validation in IoT devices, to ensure that no fake application can send requests to the devices [23].

To implement application ID validation, we consider OpenID Connect 1.0 - an authentication layer built on OAuth2.0 - to be viable in the implementation of such a mechanism.

Using OpenID Connect's authorisation code flow, during registration of the TP-Link device, the Kasa Smart app is directed to an Authorisation Server that is trusted by TP-Link. The app subsequently prepares and sends an authentication request via a RESTful API call, and upon authentication of the user's identity by the server, the user's consent is requested for the application to access the user's identity. Subsequently, the app is issued with an authorisation code which may be exchanged for an identity (JSON Web Token) through a server-side call to the authorisation server. The identity token is signed by the authorisation server via the JSON Web Signature [24], and recipients are then able to verify its authentication, integrity and non-repudiation [25].

Once the unique identity token is obtained, the application sends it to the TP-Link device to be encrypted and stored, as part of the authentication mechanism.

Whenever commands are sent to the device, the identity token is included for the device to validate the application's identity. If an invalid identity token is used in a "fake" request, it would be ignored by the device. The flow discussed above may be seen in Figure 12 below.

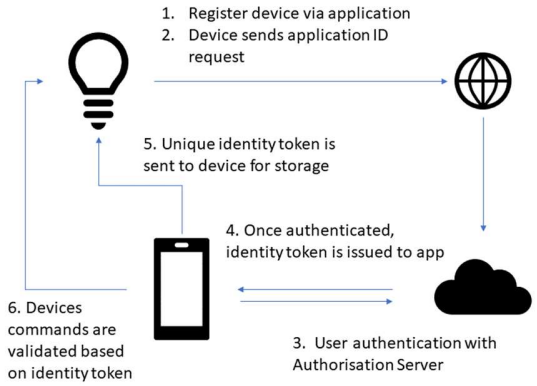

Figure 12. Flow for proposed application ID validation

7. CONCLUSION

Through an iterated process of reverse engineering, we successfully identified several security vulnerabilities still present in the TP-Link devices - a leading producer of IoT devices. Moreover, by developing a web interface as a proof of concept, the feasibility of discovering and exploiting security vulnerabilities in an industry-standard IoT product was portrayed. This endorses the need for new security mechanisms for not only TP-Link devices, but other IoT devices as well. Followingly, we explored and evaluated the feasibility of potential defence mechanisms for IoT devices, with a focus on encryption enhancements. Overall, our paper highlighted the host of potential vulnerabilities existing in the security mechanisms of current-day IoT devices, and demonstrated the extent of its exploitation - a foreshadowing of the impact on IoT ecosystems, should current security standards for IoT devices remain unchanged.

8. REFERENCES

[1] TechTarget. 2019. Internet of things (IoT). Retrieved November 5, 2019 from https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

[2] A Mechanism for Securing IoT-enabled Applications at the Fog Layer. November 2, 2019. Retrieved October 20, 2019, from https://www.mdpi.com/2224-2708/8/1/16/pdf

[3] Softcheck. 2016. Reverse Engineering The TP-Link HS110. Retrieved November 5, 2019 from https://www.softscheck.sg/reverse-engineering-the-tp-link-hs110/

[4] OpenWRT. 2017.OpenWRT Project: TP-Link HS110. Retrieved November 3, 2019, from https://openwrt.org/toh/tp-link/hs110

[5] Technopedia. What is Busybox?. Retrieved 28 October, 2019, from https://www.techopedia.com/definition/27267/busybox.

[6] Tech-faq. 2011. Flashing Firmware. Retrieved 1 November, 2019, from http://www.tech-faq.com/flashing-firmware.html.

[7] 张向明, 李智威. 2011. Data communication method, system and processor among CPUs. (Jan. 2011). Patent No. CN102096654A, Filed Jan. 1st., Issued Sep. 18th., 2013.

[8] Homenet Howto. UPnP, automatic port forward. Retrieved 6 November, 2019, from https://www.homenethowto.com/ports-and-nat/upnp-automatic-port-forward/

[9] Wikipedia. 2019. Elliptic curve point multiplication. Retrieved 3 November, 2019, from https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication.

[10] T. Banerjee, M. Anwar Hasan. (2018). Energy Efficiency Analysis of Elliptic Curve based Cryptosystems. Technical Report of the Centre for Applied Cryptographic Research, University of Waterloo, CACR.

[11] Nilupulee A et al, Next Generation Lightweight Cryptography for Smart IoT Devices: Implementation, Challenges and Applications. (2019) Retrieved 3 November, 2019 from http://www.jkjmanagement.com/wfiot/papers/1570506899.pdf

[12] MIPS. 2008. MIPS32® 24Kc™ Processor Core Datasheet. Retrieved 7 November, 2019, from https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00346-2B-24K-DTS-04.00.pdf

[13] Daniel A.F.Saraiva, et al. 2019. PRISEC: Comparison of Symmetric Key Algorithms for IoT Devices. Sensors 2019, 1919, 4312. https://doi.org/10.3390/s19194312

[14] D. J.Wheeler and R. M.Needham, “TEA, a Tiny Encryption Algorithm,”in Fast Software Encryption: Second International Workshop, FSE’94 Leuven, Belgium, 14-16 December 1994, Proceedings, 1994, pp. 363–366

[15] Wikipedia. 2019. Millions of instructions per second. Retrieved 14 November, 2019, from https://en.wikipedia.org/wiki/Instructions_per_second

[16] Roger M. Needham and David J. Wheeler. Tea extensions. Technical report, Computer Laboratory, University of Cambridge, October 1997

[17] Bogdanov, A.; Khovratovich, D.; Rechberger, C. Biclique Cryptanalysis of the Full AES. In Advances in Cryptology—ASIACRYPT 2011; Lee, D.H., Wang, X., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 344–371

[18] Y.Nir, Check Point, A. Langley, Google, Inc. 2015. ChaCha20 and Poly1305 for IETF Protocols. Retrieved 9 November, 2019, from https://tools.ietf.org/html/rfc7539

[19] Daniel J. Bernstein. 2008. The ChaCha family of stream ciphers. Retrieved 13 November, 2019, from https://cr.yp.to/chacha.html

[20] Engels D., Saarinen MJ.O., Schweitzer P., Smith E.M. (2012) The Hummingbird-2 Lightweight Authenticated Encryption Algorithm. In: Juels A., Paar C. (eds) RFID. Security and Privacy. RFIDSec 2011. Lecture Notes in Computer Science, vol 7055. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-25286-0_2

[21] Common Weakness Enumeration. CWE-798: Use of Hard-coded Credentials (3.4.1). (September 2019). Retrieved 4 November, 2019 from https://cwe.mitre.org/data/definitions/798.html

[22] CSO Online .The Mirai botnet explained: How teen scammers and CCTV cameras almost brought down the internet. (March 2018) Retrieved 1 November, 2019, from https://www.csoonline.com/article/3258748/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-down-the-internet.html

[23] Sandeep Kumar Polu. 2018. OAuth based Secured authentication mechanism for IoT Applications, International Journal of Engineering Development and Research (IJEDR), ISSN:2321-9939, Vol.6, Issue 4, pp.409-413, December 2018, Available :http://www.ijedr.org/papers/IJEDR1804075.pdf

[24] OpenID. 2014. OpenID Connect Core 1.0 incorporating errata set 1. Retrieved 5 November, 2019, from https://openid.net/specs/openid-connect-core-1_0.html#CodeFlowSteps

[25] M. Jones, Microsoft, J. Bradley, Ping Identity, N. Sakimura. 2014. JSON Web Signature (JWS) draft-ietf-jose-json-web-signature-41. Retrieved 5 November, 2019 from https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41