Security Analysis of Vendor Customized Code in Firmware of Embedded Device

Muqing Liu(✉), Yuanyuan Zhang, Juanru Li, Junliang Shu, and Dawu Gu

Lab of Cryptology and Computer Security, Shanghai Jiao Tong University, Shanghai, China
liumuqing@sjtu.edu.cn

Abstract. Despite the increased concerning about embedded system security, the security assessment of commodity embedded devices is far from being adequate. The lack of assessment is mainly due to the tedious, time-consuming, and the very ad hoc reverse engineering procedure of the embedded device firmware. To simplify this procedure, we argue that only a particular part of the entire embedded device's firmware, as we called vendor customized code, should be thoroughly analyzed. Vendor customized code is usually developed to deal with external inputs and is especially sensitive to attacks compared to other parts of the system. Moreover, vendor customized code is often highly specific and proprietary, which lacks security implementation guidelines. Therefore, the security demands of analyzing this kind of code is urgent.

In this paper, we present empirical security analysis of vendor customized code on commodity embedded devices. We first survey the feasibility and limitations of state-of-the-art analysis tools. We focus on investigating typical program analysis tools used for classical security assessment and check their usability on conducting practical embedded devices' firmware reverse engineering. Then, we propose a methodology of vendor customized code analysis corresponding to both the feature of embedded devices and the usability of current analysis tools. It first locates the vendor customized code part of the firmware through black-box testing and firmware unpacking, and focuses on assessing typical aspects of common weakness of embedded devices in the particularly featured code part.

Based on our analysis methodology, we assess five popular embedded devices and find critical vulnerabilities. Our results show that: (a) the workload of assessing embedded devices could be significantly reduced according to our analysis methodology and only a small portion of programs on the device are needed to be assessed; (b) the vendor customized code is often more error-prone and thus vulnerable to attacks; (c) using existing tools to conduct automated analysis for many embedded devices is still infeasible, and manual intervention is essential to fulfil an effective assessment.

Keywords: Security assessment · Vendor customized code · Embedded device

This work was partially supported by the Major Program of Shanghai Science and Technology Commission (Grants No.: 15511103002).

1 Introduction

Embedded devices are nowadays widely deployed in not only industrial environment but also in personal residences. While the embedded devices are becoming more and more prevailing, their functionalities are becoming more sophisticated. Smart embedded devices are used to perform home network routing, TV signal receiving and decoding, real-time camera monitoring and security altering, etc. As a result, the code base of current embedded device is much larger compared to previous one, and will become even more complex with the evolvement of smart homes and related Internet-connected devices

Security issues often compose severe threats to embedded devices and applications being deployed. Unfortunately, many embedded devices are designed and implemented without a clear and well-defined security goal. An observation is that the development of embedded device tends to repeat the mistakes once occurred on developing desktop computer systems. However, while manufacturers and researchers invest time and money in testing and securing them, the status of security assessment for those embedded devices is far from well-developed since the assessment tool is insufficient. Since a profusion of embedded devices have been developed by various manufacturers and been used in different environments. this inherent diversity makes the universality of security analysis a very difficult goal to be achieved. Although a multitude of research works have been proposed, less developed tool is universal to different devices. Although a large portion of research works aim to develop novel and automated analysis techniques suitable for embedded device, to the best of our knowledge, most of those techniques are only suitable for a small range of device models. Due to the lack of proper tool, security assessment of embedded device is well-known as a highly skilled procedure and requires expertise, which is still not systematic and practical. Hence, to help developers testing the security of embedded devices, not only should we transit the experience and best practice of security analysis for classical computer systems to the analysis of embedded device, but also should we consider the restriction of tools and formulate proper security assessment procedure that is practical and universal.

To employ effective and in-depth analysis, manual effort is still essential. However, since the manual analysis (e.g., reverse engineering of the firmware of the embedded device) is often time-consuming, the scalability must be effectively controlled to make such analysis feasible. An important aspect of reducing the amount of analyzing work is to elaborately filter out unnecessary targets. Particularly, it is necessary to extract and analyze only those executables related to possible attack surface of the embedded device. Thus the problem is how to locate such executables. The vendor of the embedded device often provides a firmware containing both operating system and the applications. A common observation is that most of the code in the firmware is reused (e.g., OS and standard libraries) and is publicly certificated, and it is often laborious and unnecessary to verify every part of the firmware. Therefore, focusing on the code related to high level operations of the device (e.g., network communication, user interaction) is more likely to find potentially vulnerability.

The target of this paper is to illustrate the security analysis methodology of vendor customized code assessment. Vendor customized code (VCC) denotes to the code fulfilling specific functionalities of the device. For instance, a wireless router may contain particular code to fulfil the authentication of the user. That code usually performs a proprietary authentication implemented by the vendor and is therefore not publicly known. This kind of code is often not fully evaluated by professional security analyst and is error-prone. Our security assessment thus focuses on such vendor customized code trying to find security flaws. In detail, our security assessment focuses on four typical aspects: *protocol cryptographic misuse*, *identity authentication*, *firmware integrity tampering*, and *incorrect patching of known vulnerability*. This helps us concentrate the analysis task and conduct practical operations.

To employ the assessment, we first summarize to what extend could the prevailing commodity program analysis techniques and automated security assessment tools be applied to existing embedded device systems. The issues of analysis of embedded devices are concluded and we propose some solutions to address them practically. The next goal is to find vendor customized code and analyze it. To this end, we present a systematic security assessment procedure to help conduct firmware reverse engineering and vendor customized code searching. We expect our proposed procedure to dispel misconceptions and mystifications of embedded devices' security assessment, and further promote the analysis efficiency of typical embedded devices. Notice that our assessment is not trying to answer questions like "are there any security flaws in the device" or "are some functionalities of this device is secure". Instead, our methodology is to answer the questions that how a vendor feature is implemented, and whether the implementation (vendor customized code) violates some expectations for the feature.

To evaluate the effectiveness of our methodology, we demonstrate the experimental results using five embedded devices including two wireless routers, one smart camera, one modem, and a smart CDN device. By adopting our methodology, vendor customized code can be located accurately and the amount of code needed to analyze is reduced significantly for each device. What's more, the extraction of vendor customized code allows us to employ in-depth security assessment, which reveals critical security flaws among those devices that are not discovered before.

2 Issues of Firmware Analysis

In this section we briefly review the state-of-the-art tools and techniques proposed for embedded devices' firmware analysis, and discuss their deficiencies. Although many classical security analysis techniques are applicable for embedded device's code, corresponding analysis tools may not, or at least not fully, adapted. We summarize major issues of current analyses in the following.

2.1 Firmware Acquiring

Unlike commodity personal computer, the executable code of operating system and applications of embedded device are not easily accessed. Obtaining firmware of the device is usually the most direct and the only way of analyzing target programs. To obtain the firmware of a device, two aspects should be concerned: the updating process of firmware, and the storage format of the device. Both of them are the frequently utilized sources to help acquire complete or part of the firmware.

The most common way for users to update their embedded devices is to uploading an firmware image provided by the vendor via a specific interface (an upgrade page for web management in most routers, upgrade utilities for Apple Airport, HP Printer, etc.) Thus analyst has the chance to intercept this process and extract the firmware image. Although previous studies often utilize the accessed URL of the update process and use crawler to download firmware packages from vendor's website, especially for large scale analysis [1], many devices often perform automatic and silent update checking to upgrade firmware. For those upgrade routines in which firmware packages are not direct accessible to users, manual analysis is still required to trace the upgrading agents and network traffic (and obtain the firmware).

Another major source of firmware is the storage medium of devices. For most embedded devices, softwares and configurations are stored in their local storage, such as ROM and flash. Thus analysts can manually dump stored content for further unpacking. This technique for device repairing and memory forensics has been widely applied to firmware analysis [2] with the growing concern of embedded security. To hamper firmware dumping, SoCs of some vendor may encrypt the firmware in ROM. In this case, although advanced dumping methods such as half-blind attack [3] can be applied, much manual intervention are required and sometimes the analyst may make use of known vulnerability (e.g., memory corruption) to help dump data.

2.2 Firmware Unpacking

In most cases, the obtained firmware is a single image that requires to be separated into different parts according to it's original layout organization. State-of-the-art tools such as Binwalk [4], FRAK [5], and BAT [6] provide functionalities for standard format firmware unpacking. By scanning signatures of common file systems and file formats in firmware images, individual files or a whole file system will be identified and extracted automatically and recursively. If the header of a common file system is identified, it is used as the identifier to split the image, and the cut out filesystem part is able to be mounted on another normal Linux system. After that, files inside the image are available to access.

In our practice, however, although those unpacking tools are sufficient for most Linux based firmwares, which contains a common file system (*squashfs*, *jffs2*, etc.), there are still many so-called monolithic firmware images of Real- Time Operating System (RTOS), which are packed with proprietary formats. For those images, universal tools often fail to identify and unpack them.

Another situation is that the acquired firmware is only a raw image of storage dump instead of a well formatted package, and there often does not exist the concept of file in such image. In this case, manual effort on determining the entry point of the raw bootloader becomes the last resort to recover possible system kernels and software applications, even if it is very complicated and tedious.

2.3 Code Debugging

Debugging code on embedded device is inconvenient and often impossible. A commercial off-the-shelf embedded device is rarely enabled debugging functionality of its main board. Comparing with a development board, a released device has no debugging peripherals to monitor the running state of the CPU and memory. Meanwhile, a full debugging solution including technique documents and debugger softwares is not provided either. Thus, it is usually impossible to directly debug a device like what developers do.

To implement the task of debugging, analysts utilize alternative measures. For instance, when using GDB stub [7] to debug, the device should execute a piece of stub code to build a remote debugging tunnel before booting. Then, the stub downloads and executes the original firmware. Also, if the system of the device is an embedded Linux System, analyst could attach debug server to specific running process and debug it. But to implement those functionalities inevitably involves manual intervening. As the vendors are not likely to provide the debugging privilege to normal user, analyst should either insert such a stub before booting process or gain a root privilege of the devices, which are both not easy to achieve.

2.4 Code Emulation

Emulation is a promising alternative way to achieve dynamic code analysis and inspect any run-time information. State-of-the-art emulators support most architectures (MIPS, ARM) that embedded devices are adopting. The overhead for emulation is also acceptable (four times slower than the execution of native code according to [8]). But a main restriction for emulation is that it often requires a full system emulation to execute the code correctly. For an embedded system image, it is not emulated as easy as desktop systems. Desktop operating system only requires few I/O devices (hard drive, screen, etc.) to boot up. Those I/O devices work with a clear protocol to emulate. For a embedded device, on the contrary, because neither peripherals details nor board support package of a devices are easily known for an analyst, full or approximate system level emulation is usually impossible.

Previous works [9] try to address this issue by utilizing process-level emulation or run an ELF executable file in another emulated Linux environment. Unfortunately, this technique is not always effective since a large portion of embedded software access NVRAM to load/store the configurations, which is essential to the execution but hard to be accurately emulated.

3 Vendor Customized Code Analysis

3.1 Target

As a complex combination of tightly connected softwares, an embedded system consists of many parts and components such as bootloader, operating system, daemon software, shared libraries, etc. Among them, most are auxiliary components. For example, DHCP daemon in a router, decompression libraries in bootloader are used to be implemented using open-source code base or mature solution, which are vetted beforehand and rarely have vulnerabilities. So, for a specific embedded device, we should focus on those **vendor features** which are implemented by the vendor self, and only specific for one or a series of device. The code to implement those features is often the particular part of the system that accepts user's input, which means the attack surface is restricted to this part of code. We denote this part of code as the **vendor customized code**. In a word, in the firmware of an embedded device, vendor customized code indicates to those proprietary code which are implemented for some vendor features.

In the following sections, we will demonstrate our security concern about vendor features and our methodology to locate and assess vendor customized code. In detail, we try to answer the following investigative questions of common functionalities related:

- **Q1**: *Is the protocol properly protecting private data transferring on the Internet?* Since the "Never roll your own cryptography" principle is not familiar to non-expert developers, many home-brewed cryptographic procedures are used in an embedded device. Meanwhile, due to the inherent complexity of cryptographic libraries, cryptographic misuse becomes another critical problem [10]. Thus any proprietary protocol should be assessed to find potential cryptographic flaws.
- **Q2**: *Can the device properly authenticate a granted user accessing this device?* Nowadays many embedded devices utilize an HTTP management interface for user to set up device configurations. Vendor may exclude unnecessary session modules, which are commonly used in current web authentication application from the web server program of the device. This introduces new authentication factors and potentially causes various security problems. Another common problem in embedded devices is that vendor may intentionally leave some backdoors to access the device conveniently [11, 12]. This often tends to be a serious security threat and leads to security breach to the device.
- **Q3**: *Could the integrity of firmware in this device self be preserved?* Modification attacks against firmware [5] often inject malicious code into a device, and the final user turns to be the specific victim. Since consumers are usually not able to distinguish a refurbished device and will not know if the software in a device has been modified, attacker could modify the original executables on the device before it is sent to the end-user to fulfil the attack. Meanwhile, attacker could also intercept the firmware update process of the device and inject malicious image if the integrity checking is missing. For these reasons, we should consider whether the device adopts a robust code integrity checking scheme to protect the system against any unauthorized code's execution.
- **Q4**: *Have previous vulnerabilities been correctly patched?* Some failed patches [13, 14] for PC software or mobile software has been witnessed in recent years. This would also happen to embedded devices, and thus additional assessment on a patched code is still essential.

3.2 Searching Vendor Customized Code

Black-Box Behavior Analysis. Part of vendor features are obvious such as video capturing for a smart camera, while much more of those are hidden. For example, automatic upgrading of a device may not be acknowledged by the users, but clues of this behavior can be found in the network record.

It is what we concerned of to discover vendor features in this stage of analysis. In one hand, we test any functionalities of target device, by feeding data normal or abnormal and recording the response. In the other hand, during the blackbox analysis, the network traffic is captured which reveal the corresponding host, port and protocol for any network connection.

Executable Retrieving. In this section, we will show how we derive binary code, which enable our white-box analysis for target devices.

We derive the code from two major sources, one is a running device, and the other one is device firmware package for device upgrading or recovering.

We can obtain software code of a running device, if we can access the console of this device. Possible methods are, utilizing previous vulnerability to get a shell, manufacturing a malicious upgrading firmware package to get a backdoor, connecting to the console of device via UART interface directly, etc. If any of those attempts succeed, we can easily collect binary codes, executable files and runtime information.

The other method is to decompose the firmware package. *file* utility and *binwalk* [4] are used to identify known format for file and data blob. If the steps above failed, which usually happens to a monolithic operating system, we try to find the correct base address of the image, and then get a more precise disassembly code.

Vendor Customized Code Locating. The final step of our searching is to locate some code snippets which are responsible for the vendor features we concern about.

This step plays an important role on minimizing the range of code analysis by restricting it into few small pieces.

In detail, two code locating schemes can be followed depending on whether we can access to the running device, respectively. If we are able to access to the running device, listening port/alive connection can be used to infer the responsible process. For example, we can execute ‘lsof -i TCP:80 -n’ or ‘netstat-ln’ command to get the process ID for the web server running in the devices. Although such utilities may not installed in the devices, we can upload a static linked utility to the device, since we have root privilege. Some devices adopt tailoring embedded Linux system, which have no required utilities (‘nc’, ‘wget’, ‘chmod’, etc.), to upload a executable file directly. In this case, we first copy an existing file which already have ‘X’ permission, and then overwrite the copy by ‘echo’ arbitrary bytes into this copy. As far as we known, since all the shell for Linux enable ‘cp’ and ‘echo’ command, this method is universal for all devices which adopt Linux operating system. We are also able to obtain the corresponding program binary by following the symbolic link, which is located at ‘/proc/PID/exe’. Any shared libraries which are loaded in the process can be also found in ‘/proc/self/map files/’ or ‘/proc/self/maps’. To confirm whether a process is related to some features, we kill the running process, and observe whether the vendor features are still working.

The other scheme is a static string analysis based. We perform keyword string searching for every extracted executable from the whole firmware image. For a specific feature, some critical keyword should be used by the code. For example, if the target feature is a HTTP server running in the device, *GET*, *POST*, *HTTP*, *Host* can be a set of keywords. If we find an executable contains specific keywords, we choose it as the candidate executable for the following security assessment.

3.3 Security Assessment of Vendor Customized Code

Our assessment is to recovery any details of vendor customized code. As the analysis scope has been minimized and restricted, coarse-grained manual static assessment is feasible in an acceptable period of time, to understand the basic behavior for the VCC. But there are also many pieces of code are complicated, and also not easy as well as necessary to understand. For example, some encoding/decoding functions may be significant for protocol analysis, which only contains encoding algorithm. We do not have to be aware of the details of this implementation, but only need obtain some output for some specific input. In this case, We try to debug, or emulate the piece of code. Some executable files depend no peripherals, thus we run and debug it in an QEMU emulated Linux system for the corresponding architecture. We also emulate those routines by unicorn engine, if no unknown initialized global variables or I/O will be accessed during the execution of those routines. We may also need to derive the input for specific output. In this situation, we utilize angr to perform a dynamic symbolic execution. To achieve this, we feeding the routine a symbolic input, and explore the path satisfying the constraints for output data at exit point of the routine.

4 Experimental Evaluation

In this section, we investigate five commodity embedded devices in Chinese market including two wireless routers, one modem, one smart Content Delivery Network (CDN) device, and a smart camera. We first report our analyzing results of firmware and vendor customized code of these devices. Then we discuss the security assessment of vendor customized code focusing on the issues mentioned in Sect. 3.3, and report the discovered vulnerabilities.

4.1 General Analysis

The five devices we choose to assess are:

- *TP-LINK WR740nv5:* a wireless router produced by TP-LINK, the world largest WLAN device manufacturer.
- *TOTOLINK A850R*: a wireless router produced by TOTOLINK, a networking equipment vendor in Korea.

- *HUAQIN HGU421:* a fiber optic modem used by *China Telecom*, the major ISP in China.
- *Thunder Money Maker:* a smart CDN device manufactured by Thunder Corporation.
- *Yi Smart Webcam*, a smart camera released by Xiaomi Inc., one of the leading Chinese consuming electronics cooperations.

For each device, we test every functionality of it, try to observe how the device behaves and record any network event as mentioned in Sect. 3.2. After that we figure out some vendor features for each devices:

- Different from many other routers of TP-LINK that adopt embedded Linux system, *TP-LINK WR740nv5* is based on VxWorks, which is a prevailing RTOS. We find that this device will check if a user uploaded firmware package is valid or not.
- As a tiny CDN node, *Thunder Money Maker* shares customer's bandwidth to Thunder CDN Network, and earns commission from Thunder Corporation according to the amount of uploaded data. It will automatically download upgrade package via HTTP protocol. Moreoover, we observed the device frequenly prompt SSL request to the URL *kjapi.peiluyou.com*.
- *TOTOLINK A850R* is a Linux based wireless router. Common features for home router are provided via an HTTP based interface. During our test, we found that no cookie is used to authenticate web users (the response contains no 'Set-Cookies' header for a successful login request), which suggests that device uses an uncommon way for authentication.
- *Yi Smart Webcam* allows users to bind their mobile phones to the camera, then the real-time captured video can be watched in a corresponding app on user's phone. The format of video stream tansfered via Internet is unkonwn. Thus we guess the data is encrypted or some-kind proessed. Also we notice that the device listens on a strange TCP port, and response a magic string when a client connects to it.
- *HUAQIN HGU421* allow users to login to configure the device via a WEB interface. But only a low-privileged account is given to user.

4.2 Firmware Analysis

We first try to gain access to each device, as mentioned in Sect. 3.2. We find four of them disclose an access terminal via different ways.

- For *WR740* and *HGU421*, we direct access them via TTL.
- For *A850R*, a known PoC of remote command execution vulnerability is used.
- For *Thunder*, we replace the upgrading package during the auto-upgrading process, and implant a backdoor.

We also scan for any known-formated blob for those device with firmware package provided. The results are listed in Table 1.

After the primary analysis, we show that it is able to retrieve executable files (ELF files) for four Linux based devices (from known format file system image, or download

Table 1. Firmware format and content analysis

Device	Firmware format	File carving result
Thunder	Zip file	None
WR740	Unknown	Some zlib compress image
A850R	Unknown	A standard squashfs file system
WebCam	Unknown	A jffs2 file system
HGU421	–	–

from a running device). The only device whose executables are not able to be extracted is *TPLINK WR740*. To address, we utilize the memory dumping functionality provide by UART console to directly retrieved the application code. Since no UART pin is provided on the mainboard of *WR740*, we distinguish the corresponding pins of AR9331 microcontroller according to its datasheet [15], and directly welded two jumper wires to lead them out (as depicted in Fig. 1). Finally we conduct memory dumping to obtain a raw image of the memory. After the dumping, we obtain the memory image containing kernel and application.

We also decompress all compressed blob in the firmware package, list all the strings appearing in those images using *strings* utility. Some strings such as "*auto-booting…*", "*I'm booting now……*", and "*Press Ctrl + C or Shift + C to stop auto-boot…*" indicate the existence of the bootloader. To determine the base address of the bootloader, we find a piece of code of switch statement by searching the switching jump instruction (*jr $v0)*. Before swiching jump instruction, few instructions (*sltiu $v0, $v1, 9; beqz $v0, default; nop;..; li $v0, 0x8046FE70; addu $v0, $v1; lw $v0, 0($v0); jr $v0)* reveals the virtual address (0x8046FE70) and size (9) of the jump table. It's also able to find the starts of most case blocks, because case blocks are usually after the end (a jump instruction to the end of swich statement) of another case block. Since all the case blocks should has a corresponding pointer in the jump table, we can easily deduce the bootloader is loaded to the address of 0x80400000. Once the base address is determined, this image can correctly disassembled and most of the binary code is readable.

4.3 Vendor Customized Code Searching

To locate the vendor customized code out of all the retrieved executable code, various criteria are applied to five devices as mentioned in Sect. 3.2. In the following, we demonstrate how we locate the corresponding vendor customized code for each vendor feature, and show the amount of assessment is significantly reduced after it (Table 2).

– As *TP-LINK WR740nv5* has a fully unknown formated firmware, we firstly found the bootloader has a recovery functionality interacting with users on the serial port. After loading the bootloader in IDA, some prompt strings such as *Usage error, please try %s-help* lead us to the code for the recovery mode. During the analysis of firmware recovering routines, we confirmed that the entire operating system compressed in a special region of flash is also a piece of the firmware package. Therefore, the decompressed operating system as well as applications were

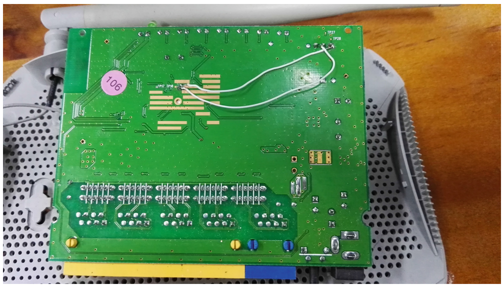

Fig. 1. Jumper Wires connect to UART pins of AR9331

Table 2. Vendor feature and size of customized code for each device.

Device	Feature	Unpacked firmware size	Vendor customized code
WR740	Firmware verification	2.4M (3868 functions)	83 functions
WR740	Packet forwarding	2.4M (3868 functions)	46 functions
Thunder	Upload statistic reporting	54.6M	3.9M shared library
A850R	User authentication	29.1M	476K executable file
WebCam	Video encryption	33.3M	272K executable file
HGU421	User authentication	No firmware package obtained	836K executable file

obtained. Then, we tried to figure out whether the device is able to verify the validity of an uploaded firmware package. We locate this feature by searching the URL for upgrading page in the HTTP. This helped us pinpoint the logic of upgrading and confirmed the missing of firmware integrity check.

- For *Thunder*, we focused on the implementation of its unique upload statistic reporting functionality, which is directly related to user's reward. We found a *dcdn client* process is listening to one particular port (4693), and more than 10 different IP addresses are connected to that port. We also found in this process a library *libdcdn client.so* is loaded. Searching for meaningful names in symbol tables, five potential relevant functions were located.
- For *TOTOLINK A850R*, we aimed to find executables related to user authentication. We first statically disassembled every binary code file on the device to determine which one is the web server. After the code reverse engineering, we identify the file with name 'boa' as the device's web server. Therefore, the following in-depth analysis could concentrate on this executable.
- Our black-box analysis indicated that the *Yi Smart Webcam* encrypts the data before sending it to the user's mobile application. The very particular vendor feature we concerned about is how the data is encrypted. The corresponding process was first identified according to the network connection information. Then we obtained the executable of the process and further searched for functions responsible for data

encryption. We found symbols for AES encryption function in wolfSSL library occurred in the executable. This helped us narrow the assessment scope of executable code to only one function.

- As mentioned before, the *HGU421* modem sets a restricted privilege to its normal user. The assessment of this device is therefore trying to find how the device authenticate an user. Similar to the feature locating of *TOTOLINK A850R*, we used the URL for login page as the indicator to search every executable. We found the executable *uhttpd* as the corresponding file, and we could focus on analyzing the handler for requesting to that page in this executable.

4.4 Security Assessment

In this section, we will demonstrate our concrete secure assessment for each device.

Device Modification. Among the five devices we analyzed, we found TP-LINK WR740nv5 and Thunder Money Maker fail to check the integrity of the code. We detail the vulnerability and related attack as follows.

TP-LINK We manually analyzed the routine for firmware verification in *TPLINK* and obtained the exact format of firmware. According to our analysis, an MD5 checksum is contained in the header of the firmware package, the uploaded package will be accepted only if the checksum of remaining data corresponds to the one in the header. So the attacker can easily modify the package to inject some malicious code into the firmware and then repack it. We also emulated the MD5 hash function in the *unicorn* engine [16], and the emulated code also executes correctly with correct result returned. This means even if the checksum function is proprietary, the attacker can simply reuse it to re-create a valid firmware package.

To demonstrate the effectiveness of the modification, we inject code into the firmware's packet forwarding routine to duplicate any packet to a specific host. This malicious firmware leads the device to send the entire network traffic to a malicious server, which proves the threat of the firmware modification attack.

Thunder. As the major functionality of this device, the file uploading and its corresponding CDN protocol of Thunder Money Maker is very complex, and the communication between account server and the device is encrypted by an unknown encryption algorithm. However, we conduct a device firmware modification attack to circumvent the encryption. We directly reverse engineer the 'libdcdn client.so' file, which is responsible for D-CDN function in this device. By searching for meaningful names in exported symbols, we can locate five potential relevant functions. After a manual analysis, how each function returns statistical data (i.e., in return value, arguments or global variables) is understood. Then we modify each function to report a tampered statistical data, uploaded it to the server to replace the original one and check whether the uploading speed shown in the mobile application is changed. Through this testing we pinpoint the specific function for accounting. By maliciously adjusting the upload speed of this function, we can cheat the server to earn much more money than what we should deserve.

Corrupted Authentication

A850R We extract the binary file of the web server, *Boa*, in the *squashfs* filesystem of A850R. The handle for login request is located by searching login path ‘boafrm/formLogin’. The handle contains only 172 assembly instructions, so it’s easy to recover its logic. Instead of keeping sessions for login users, *Boa* stores the user’s IP into the management information base (MIB) as an item ‘LoginIP’. Different from normal session management, this mechanism in web server has two problems:

1. Any clients behind a NAT box in LAN will share same address connecting to the router, which means privilege will be leaked to the whole subnet if one user logins to the HTTP interface.
2. An attacker in LAN may simply discover and gain the authorized IP address by ARP spoofing.

List 1.1 Authentication Code in A850R

```
bool check_login(input_user, input_pass)
{
  char user[16], pass[16];
  char user0[16], pass0[16];
  load_from_mib(USERNAME, user);
  load_from_mib(PASSWORD, pass);
  //load_from_mib(SUPER_USER, user0);
  //load_from_mib(SUPER_PASS, pass0);
  if (0==strcmp(input_user, user) && 0==strcmp(input_pass,
      pass))
  {
    return true;
  }
  if (0==strcmp(input_user, user0) && 0==strcmp(input_pass,
      pass0))
  {
    return true;
  }
  return false;
}
```

We find that, request not from “loginIP” will also be allow to access the management interface, if correct username and password are provided in ‘Authentication’ header of HTTP request. There is another vulnerability in the routine to check ‘Authentication’ header which is shown in List 1.1. To check the validity of ‘Authentication’ header, the web server allocates two pairs of username and password on the stack. One is filled with the actual user’s information, which is loaded from MIB, while the other is left to be uninitialized. Then the server compares the user’s input with these two pairs of information, authorizing the user if either of the information is matched. Since both of the uninitialized stack strings may be empty, it is possible for attacker to bypass the authentication with an empty username and password. What make things worse is that the CPU of A850R is a big-endian architecture. The most significant byte of a word is often zero, if the word is a small interger, pointer to static array, or return address to some position to the code section. This makes the success rate of the attack much higher, comparing with same attack against little-endian architecture. To figure out the success rate of an attacker, we employ 10 malicious attempts, and all of them succeed.

To further derive how this issue has been introduced, we intended to also compare an old version of the firmware for A850R. We found no old version firmware of A850R on the Internet, so we take an old version of N301RT to compare. We find that the authentication process is very similar to that of AR850, but the uninitialized variables are initialized as *SUPERUSER* and *SUPERPASS* in MIB.

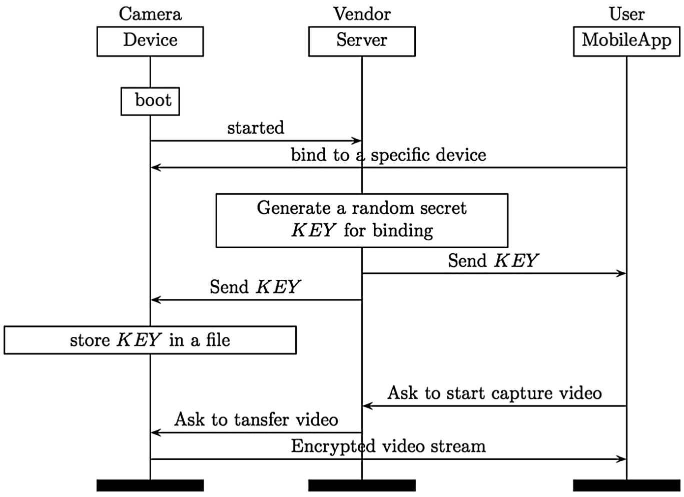

Fig. 2. Xiaoyi key agreement

Thus, we can confirm that it is the incomplete patching procedure that causes this issue.

HGU421 Our analysis of the *uhttpd* executable suggests a DoS vulnerability in this device. The vendor implements the parser (shown in List 1.2) for HTTP header of basic access authentication [17], which separates the username and password with a colon in a base64-encoded HTTP header. After a call to *strchr(header,':')*, the web server does not check whether the return value is NULL or not (i.e., whether a colon exists). Then the server uses the return value to separate the authentication info. As a consequence, an attacker may send a crafted request to crash the web server.

List 1.2 Authentication Code in HGU421

```
bool check_auth_header(char * header)
{
   char *user, *pass;
   base64decode(header);
   user = header;
   pass = strchr(header, ':');
   pass[0] = '\0';
   pass ++;
   return check_user_info(user, pass);
}
```

Insecure Protocol
Yi Smart WebCam We search any file that contains keywords 'encrypt' in the jffs2 filesystem extracted from the firmware image. The executable binary named 'remote' has been noticed since it's the only executable file containing AES related symbols from WolfSSL. We confirm 'remote' is responsible for video capture and transfer, because we also find symbols for video codec in this binary. Since this executable is not large, we manual analyse it to recover the key agreement protocol (Fig. 2). Our future analysis discovers at least two issues in *Yi Smart Camera*:

1. The vendor only encryption the first two blocks of each video stream using AES-128 with ECB mode. It seems that to make the encrypt/decrypt faster on a tiny embedded device as well as on a mobile device, the vendor abandons a standard encrypt procedure. This allows a Man-In-The-Middle attack to decode most of the video stream.
2. The key agreement of the video transmission protocol contains serious problem. Since a secret key needs to be shared between the camera and mobile application, the key is first generated by the vendor's server and is transferred to both the mobile application and the device after a binding operation. However, the camera launches a special daemon service that listens to a TCP port (38888), and echoes this session key if received arbitrary request. Then any attacker captured the encrypted stream can also get the secret key via this TCP port. It seems that this port is used to debug, but the vendor forgets to remove before releasing the device.

5 Related Work

5.1 Static Binary Code Analysis

Disassembler is required for most binary code analyses. Due to the complexity and diversity of different instruction sets, many disassembly engines (udis86, diStorm3, etc.) can only support i386/x86-64 architecture and is not feasible for embedded system analysis. IDA is the state-of-the-art universal disassembler for most of processors. Many previous cross-architecture works [18, 19] are based IDA's disassembly result.

However, to acquire a precise analysis result, the disassembling of IDA involves many interactive processes, which requires the participation of expertise. Capstone is another multi-platform, multi-architecture disassembly framework that supports ARM, ARM64, MIPS etc. But some frequently used instruction sets such as MSP430, 8051 and AVR, are still not supported.

5.2 Dynamic Binary Code Analysis

To perform dynamic analysis, analyst may execute programs in a firmware with an emulated environment. As common emulators contain no peripheral details, previous works [20–22] try to fully or approximately emulate those peripherals. Those solutions face different problems such as short of documentation or misbehavior of the emulated code. Avatar [23] is another platform that is able to connect emulated code with physical device to achieve a fully emulated environment, but until now it's not publicly released. For Linux-based firmware, although full system emulation is not possible, the file system in the firmware can be mounted and programs could be emulated. Cui et al. [9] run startup script in an alternative Linux system to emulate a running system. Meanwhile, executing an emulated process in QEMU user mode is also an alternative way. Those techniques are suitable for Linux based firmware in our practice. Even though, missing MTD devcie in file system, User Defined Instructions in MIPS, or any differences between emulated and real environment may also cause problems.

5.3 Heavyweight Program Analysis

Dynamic taint analysis and dynamic symbolic execution are prevalent dynamic analysis techniques for program analysis of desktop or mobile platform. However, most tainting and symbolic execution engines aiming for x86, ARM or JVM cannot be adapted to MIPS or other instruction sets used by embedded devices. Also, those emulating based engines are limited by the restriction of emulators, which we have mentioned before. The gap between concrete values needed by those engines and incompleteness of emulation or debugging environment is still unbridgeable now.

Static taint analysis and symbolic execution benefit from the property of not relying on runtime concrete value, and make significant uses in firmware analysis. Nevertheless, large scale taint analysis and symbolic execution encounters many issues such as path explosion, low speed of constraint solvers, difficulties for pointer identification. Project angr [24] fulfill the requirements that solving a combination of path and input data from a start point to trigger some target, or collecting possible behaviors for further manual analysis starting from a critical point. But its usage scenarios are very limited because of the overhead.

5.4 Automatic Firmware Analysis

Many works have been done on automatic firmware analyzing. Some work are scalable but perform no in-deepth analysis. A. Costin [1] scan in thousands of firmware images for specific artifacts with known problems. FIRMADYNE [25] run startup script of the firmware filesystem, in a emulated Linux system with NVRAM shared-library replaced. FIRMADYNE also find known by executing exploits from Metaspolit Framework. There are some automatic techniques have been presented to discover unknown vulnerabilities. Firmalice [26] utilizes a symbolic model of authentication bypass flaws to determine the required inputs to perform privileged operations. FIE [27] developed a symbolic engine to find memory-safety bugs in MSP430 open-source softwares.

6 Conclusion

As the embedded devices are becoming more and more complex, state-of-the-art security analysis techniques and tools are not adequate to address real-world analysis tasks. In this paper we systematically study the limitation of embedded device analyzing tools and inefficiency of automatic analysis for embedded devices. We argue that current techniques and tools are still not universal for automatic security assessment, and currently we should still acknowledge the necessity of manual intervention for an effective assessment. We then suggest a practical and comprehend security assessment procedure that focuses on common weak points of embedded design and implementation. Guide by this assessment procedure, we reveal critical security flaws in five real-world devices.

References

1. Costin, A., Zaddach, J. Francillon, A., Balzarotti, D., Antipolis, S.: A large scale analysis of the security of embedded firmwares. USENIX Security. USENIX Association (2014)
2. Cui, A., Stolfo, S.: Print me if you dare: firmware modification attacks and the rise of printer malware (2011)
3. Goodspeed, T., Francillon, A.: Half-blind attacks: mask rom bootloaders are dangerous. In: Proceedings of the 3rd USENIX Conference on Offensive Technologies, p. 6. USENIX Association (2009)
4. Heffner, C.: Binwalk-firmware analysis tool. https://code.google.com/p/binwalk/
5. Cui, A., Costello, M., Stolfo, S.J.: When firmware modifications attack: a case study of embedded exploitation. In: NDSS (2013)
6. Hemel, A., Kalleberg, K.T., Vermaas, R., Dolstra, E.: Finding software license violations through binary code clone detection. In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 63–72. ACM (2011)
7. Ji, J.-H., Woo, G., Park, H.-B., Park, J.-S.: Design and implementation of retargetable software debugger based on gdb. In: Third International Conference on Convergence and Hybrid Information Technology, ICCIT 2008, vol. 1, pp. 737–740. IEEE (2008)
8. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (2005)
9. Costin, A., Zarras, A., Francillon, A.: Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. arXiv preprint arXiv:1511.03609 (2015)

10. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 73–84. ACM (2013)
11. A vulnerability and a hidden admin account all inside sitel ds114w routers! https://rootatnasro.wordpress.com/2015/01/04/a-vulnerability-anda-hidden-admin-account-all-inside-sitel-ds114-w-routers/
12. More than 60 undisclosed vulnerabilities affect 22 soho routers. http://seclists.org/fulldisclosure/2015/May/129
13. Cve-2015-3864. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3864
14. Cve-2014-7169. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7169
15. Ar9331 highly-integrated and cost effective ieee 802.11n 1x1 2.4 ghz soc for ap and router platforms. https://www.openhacks.com/uploadsproductos/ar9331datasheet.pdf
16. Quynh, N.A., Dang, H.-V.: Unicorn: next generation cpu emulator frame-work. In: BlackHat (2015)
17. Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L.: Rfc 2617: Http authentication: basic and digest access authentication. Internet RFCs (1999)
18. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-architecture bug search in binary executables (2015)
19. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: Byteweight: learning to recognize functions in binary code. In: USENIX Security Symposium (2014)
20. Chipounov, V., Candea, G.: Reverse engineering of binary device drivers with revnic. In: Proceedings of the 5th European Conference on Computer Systems, pp. 167–180. ACM (2010)
21. Kuznetsov, V., Chipounov, V., Candea, G.: Testing closed-source binary device drivers with ddt. In: USENIX Annual Technical Conference, no. EPFL-CONF- 147243 (2010)
22. Schlich, B.: Model checking of software for microcontrollers. ACM Trans. Embed. Comput. Syst. (TECS) **9**(4), 36 (2010)
23. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D.: Avatar: a framework to support dynamic security analysis of embedded systems firmwares. In: Symposium on Network and Distributed System Security (NDSS) (2014)
24. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (2016)
25. Chen, D.D., Egele, M., Woo, M., Brumley, D.: Towards automated dynamic analysis for linux-based embedded firmware. In: ISOC Network and Distributed System Security Symposium (NDSS) (2016)
26. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice- automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS (2015)
27. Davidson, D., Moench, B., Ristenpart, T., Jha, S.: Fie on firmware: finding vulnerabilities in embedded systems using symbolic execution. Presented as part of the 22nd USENIX Security Symposium (USENIX Security 2013), pp. 463–478 (2013)