Hijacking Your Routers via Control-Hijacking URLs in Embedded Devices with Web Interfaces

Ming Yuan, Ye Li, and Zhoujun Li(✉)

Beihang University, Beijing, China
yuanmingbuaa@gmail.com, {li_ye,lizj}@buaa.edu.cn

Abstract. Embedded devices start to get into the lives of ordinary people, such as SOHO routers and IP camera. However, studies have shown that the safety consideration of these devices is not enough, which has led to a growing number of security researchers focusing on the exploit of embedded devices. A majority of embedded devices run a web service to facilitate user management, which provides a potential attack interface. But what needs to be pointed out is that unfortunately most vulnerabilities of web service need attackers to provide login credentials to access and exploit, which makes attacking much less practical. This paper presents an automated vulnerability detecting and exploiting model DAEWC (Detect and Exploit without Credentials). Firstly, the DAEWC uses the symbol execution method to find URLs that are not protected by authentication mechanism. Secondly, DAEWC aims at these URLs using fuzzing method, combined with a lightweight dynamic data flow tracking technology to analyze the web server, which can quickly and accurately find easy-to-exploit vulnerabilities. Last but not least, DAEWC implements an automatic vulnerability exploit model, which generates executable custom shellcode, for example, executing system ("/bin/sh") or read/write arbitrary memory. Using these vulnerabilities, we can attack embedded devices with web services even without the access to the web interface. For example, attackers can control a Wi-Fi router at the airport without login credentials by sending a specially constructed URL request. We applied the DAEWC to the firmware of two embedded device vendors, found 9 unreported 0-day vulnerabilities in four of them and generated highly usable exploit script.

Keywords: Firmware · Authentication-bypassing URLs · Symbolic execution
Lightweight dynamic data tracker · Automatic exploit generation

1 Introduction

Along with the development of the Internet of Things, more and more embedded devices connect to the Internet. For the convenience of user management, these devices usually run a web service, so that users can remotely operate or set these devices. Users need to have a web authentication before the management operation, and only after the authentication (such as the username/password check or the cookies check) they can have access to other pages. If authentication fails, the server will return a 401 HTTP code (unauthorized response), or a 302 HTTP code that redirect user to the login page. However, in practice, some device manufacturers do not complete the verification of all user URL requests, which leads to unauthorized access to some URLs. Once the function to handle this type of URL request in the web sever is vulnerable, attackers can control the firmware program flow by directly requesting these URLs and sending malicious payload even without the login credentials. We call this type of URLs control-hijacking URLs.

Web services on embedded devices might have vulnerabilities such as SQL injection, XSS, CSRF and so on, but these web application layer problems often fail to meet the requirements of fully controlling the devices. This article aims at the web server binaries, and focuses on the detection of web services memory corruption vulnerabilities (such as buffer overflow, command execution, format string vulnerability and so on). Once attacker control the web server program flow, they can operate with web server's privileges on the operation system level. The web server of embedded devices is usually run as root, so that attacker can control the device with the root privilege of the operation system. It is obvious that in embedded devices, compared to the vulnerabilities of the web application (i.e. CSRF or XSS), the binary's memory corruption or command execution vulnerability are more threatening. And this kind of vulnerability not only exists in theory, in fact, in 2017 the Axis camera is reported to have a vulnerability that affects millions of IoT devices and it can be exploited by sending a POST package to control the equipment.[1]

But finding control-hijacking URLs is not easy. Firstly, we can't get the firmware source code - this is a problem for all binary analysis method. Secondly, firmware is run on the device, and deploying device for each firmware will undoubtedly cost a lot. Andrei Costin [1] simulate the firmware's web interface, perform static and dynamic analysis on it. But this analysis is dependent on the existing static (RIPS) or dynamic (metesploit or exp from exploit database) analysis tools, and as the manufacturer's security consciousness improves, these vulnerabilities can be easily found by manufacturers themselves and fix, and thus the number of them has greatly reduced. Also some of the vulnerabilities they found require web login credentials (such as cookies) to be triggered, thus these vulnerability do not pose an effective threat to devices in the real world. In addition, they need to use a real Linux kernel to simulate the firmware, which is not efficient enough.

Static vulnerability detection method, such as Yan Shoshitaishvili's Firmliace, can detect logic flaw (such as back door) effectively, but for the memory corruption and command injection vulnerabilities static detection is not effective enough. However, to perform dynamic firmware testing, one needs to spend a lot on buying equipments or need to get firmware simulated. In fact, the present automated simulation technique works for only a part of the firmware and as we said before, the test result is often not ideal.

In light of these challenges, we purpose a novel firmware vulnerabilities detecting and exploiting tools - DAEWC. It can automatically extract the firmware and look for web server binary. In the firmware binary, through specific strategy it can locate the

[1] http://blog.senr.io/blog/devils-ivy-flaw-in-widely-used-third-party-code-impacts-millions.

credential authentication code in the program and use symbol execution technique to identify URLs accessible by unauthorized users, then use a lightweight dynamic data tracker system to detect potential memory corruption and arbitrary command execution vulnerabilities. In addition, this framework can also detect URLs that can open undesired backdoor services (such as Telnet, SSH), which also do not require credentials.

We tested the DAEWC model on four different routers and found at least one control-hijacking URLs on each of them. These routers have different web services, and their architecture includes ARM and MIPS. Experimental results show that DAEWC is effective and accurate to automatically detect vulnerabilities and generate exploit (exp).

To sum up, we made the following contributions:

- We purpose a novel model that can perform automated detection of Control-hijacking URLs in the embedded device firmware. It can be used to control embedded devices without the web login credentials, so the model can detect more exploitable vulnerabilities than existing firmware vulnerability detection techniques. We can exploit the memory crash and arbitrary code execution vulnerabilities without knowing the details of the firmware implementation.

We implemented a tool DAEWC for automatic vulnerability detection based on the model, which uses the original firmware as input, outputs control-hijacking URLs, and the PoC (Proof of Concept) corresponding to each problematic URL. DAEWC can detect vulnerabilities in multiple architecture firmware and is not related to device hardware platforms.

- We used DAEWC to detect four different real-world firmware samples and successfully found nine 0-day vulnerabilities. They have been submitted to CVEs.
- According to vulnerability detection report, we generated usable exploit successfully.

2 Approach Overview

DAEWC determines control-hijacking URLs in the firmware by following a few steps. First of all firmware is preprocessed. Then it find all URLs that don't need user authentication by using the static method. After that the dynamic test method is used to identify the vulnerabilities that web server may have in handling these URLs and the payload to trigger these vulnerabilities. Using the payload, a exploit program is automatically generated. At last we manually validate the results (Fig. 1).

Pretreatment. Before vulnerability analysis, we need to do some initial work, including selecting firmware to meet the requirements (we will discuss the requirements in Sect. 3), firmware decompression, determination of its root directory, finding the web service application (httpd, lighthttpd, boa, etc.) and its related files, and recording the firmware architecture.

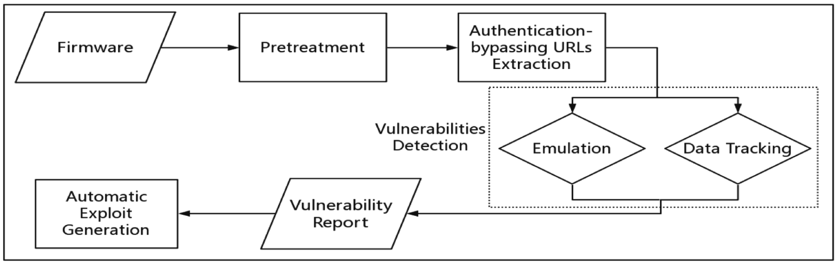
Firmware
Pretreatment
Authentication-bypassing URLs Extraction
Vulnerabilities Detection
Emulation
Data Tracking
Automatic Exploit Generation
Vulnerability Report

Fig. 1. Module Graph of DAEWC

Authentication-bypassing URLs Extraction. The pre-processed firmware is passed to the authentication-bypassing URLs extraction module. We design the URLs authentication security policies, and according to this policies the DAEWC determine the web authentication module in the firmware (we call this the URLs privilege check point). In these module, by using the method of symbolic execution and regular expression match, DAEWC find URLs in the web site that does not need web authentication. We will discuss the Authentication-bypassing URLs Extraction module in Sect. 4.

Vulnerability Detection. After extraction, DAEWC dynamically detect the URLs that are output from the previous module. Then we simulate exclusively the web server, hook the sensitive library functions in web server. According to the special URLs we construct tagged input. After the tagged input gets into our hook function, we can check the context, and perform a series of comparison operation, to determine whether there is buffer overflow or arbitrary command execution vulnerability. Finally, we export the vulnerability detection report, including the location of the bug in the assemble instructions, the payload to trigger crash, and the type of vulnerability. We will discuss the Vulnerability Detection module in Sect. 5.

Automatic Exploit Generation. In this process, we analyzed the report generated in the previous stage and established different exploit generation models for different types of vulnerability. Our ultimate goal is to run our own shellcode by hijacking control flow, to achieve arbitrary read/write, or even to bounce a shell directly to our computer. We will discuss the Automatic Exploit Generation module in Sect. 6.

Let's illustrate this method with a simplified example. For simplicity, we define the security policy for the URLs authentication as "cookies verification for the requested URL (the fixed strings 'COOKIES')". DAEWC first decompresses the firmware, extracts the web server binary, library files, and configuration files. The results are then passed to the authentication-bypassing URLs extraction module, it generates program control flow graph (CFG) and data dependency graph (DDG). Then it finds the function to verify the cookies, which has the "cookies" fixed string as the URL privilege check point of the program. Then using the angr [2] it generates to control flow diagram the data dependency graph (DPG), and makes a back slice from the URLs privilege check point, and finally uses the constraint solver to find all the URLs that do not require authentication. In the code example below, it is "goform/ping".

```
int RequestHander(char *url)
{
        if (strcmp(url,"goform/ping") == 0)
                return 1;
        else
        {
               char *request_cookies = han-
dle_header(url,'COOKIES')
               if (check_cookies(request_cookies))
               {
                        int code = 200;
                        pointless();
                        response(code,TEMPLATE);
               }
               else
               {
                        int code = 300;
                        pointless();
                        response(code,"This document has
moved to a new....");
               }
        }
}
int main()
{
        char *url = getrequest();
        char cmd[20];
        if (RequestHander(url))
        {
                snprintf(cmd,20,"ping
%s",getValue(url,"target"));
                system(cmd)
        }
}
```

These URLs are then passed into the vulnerability detection module. The DAEWC will simulate the web services of the firmware and then, using a lightweight data tracker, identify possible vulnerabilities. Specifically, we will get the parameters of the URLs of the previous output to construct special payload.

And then we hook and examine the context of sensitive system function. If the input we construct gets into the parameters of the sensitive function or overrides the key memory, you can assume there is a vulnerability here. In the above code example, when program is processing “goform/ping” request, we will find that the value of parameter “target” becomes a parameter of the sensitive function system, which is equivalent to triggering a command execution vulnerability. DAEWC will output a vulnerability report then. The automatic exp generation module, based on the report, determines that a command execution vulnerability exists, and then sets target’s parameter values to a command we expect to execute, such as ‘/bin/sh’ or ‘cat/etc./passwd’. It is important to note that in order to ensure that the command is executed successfully, we add a semicolon before the command to ensure that the preceding instructions, such as the ping command in the program, are closed. Finally, the full exploit is generated according to the payload in the leak report, so as long as the exploit program is directly run, it can obtain a shell or steal the login credentials of the user.

3 Pretreatment

We download the latest firmware from the vendor’s website. We select the firmware that was easy to decompress, with a file system, and with Unix-like operating system. We then use binwalk, firmware-mod-kit, or other tools to extract the firmware’s instruction set type based on /bin/busybox or /bin/sh. Finally, the DAEWC finds the web server (boa, HTTPD, or lighthttpd) in the decompressed directory, along with the configuration file and the document root (WWW directory).

4 Authentication-bypassing URLs Extraction

There are many ways to handle URL requests in the firmware, such as binaries (HTTPD, boa), scripting languages (PHP, LUA). We will use two methods to find control-hijacking URLs. For binary files, DAEWC use symbolic execution methods. For scripting languages, we will make targeted regular expression matches.

4.1 Regular Expression Match

Our study of some script-type CGI has found that its user authentication mechanism follows certain rule. Web server based on Lua language is very common. In our tests, we found that most of the Lua script will use entry function to determine whether you need web authentication. For example if the last parameter is set as true, it means that the interface can be accessed without the user login authentication. For this authentication method, we use a custom regular expression match to find.

In addition, some dynamic request mechanism in javascript server often ignores the authority certification. For example, in some HTML files in the router firmware we can find URLs of the ajax synchronization mechanisms to be lack of cookies validation. This type of URLs can also be found effectively through regular expression matching.

4.2 Symbolic Execution Methods

Security Policies. For binary programs, unlike other vulnerability detection system like Firmalice [3], KLEE [4], AEG [5], and Mayhem [6], DAEWC is more complex. It first finds URLs that do not require web credentials verification and then exploits them. It is actually using the firmware's logical bug in the design process to find these URLs. After connecting to the Wi-Fi, hackers can reset the router web interface's user name and password without credential validation. That is because that the firmware does not have a credentials checking when processing the URL request to modify the username.

In the absence of development documentation, to infer automatically complete web server's request processing logic is very complicated, so we need to manually specify which operation is related to checking web access and providing specific access policy. This requires the analyst to have a good understanding of the firmware internal program, and to know which sensitive programs or memory are involved in authorization authentication. We developed the following strategies for the DAEWC.

- *Fixed strings.* The security policy can be specified as several fixed strings. An example of such a policy is that the web server directly extracts the value of the "COOKIES" field in the request for inspection. We can use the data dependency graph to identify where it is in the program.
- *Behavioral patterns.* Another policy for DAEWC is to determine the location based on what the authentication program might do. For example, a server may return a 302 redirect code or 400 code in the HTTP response headers when the authentication check returns false. We can determine the location of the credentials verification code according to this behavior pattern. In addition, web server may access specific memory (a data segment that holds a user's password) after authentication or validation, which can also be treated as a behavioral pattern.
- *Manually specified.* If we already see the identification checking function when manually reverse engineering, we can specify this code as our policy.

Symbol Execution. We'll perform some static analysis before symbol execution, because instead of analyzing the entire web server, we can just focus on the credentials verification section. The DAEWC generates control flow graph (CFG) by using the static analysis module of angr, and then generates control dependency graph (CDG) on this basis, combining with data dependency graph (DDG) to generate the program dependency graph (PDG). We can then start slicing back from the credentials verification point. When back slicing, irrelevant functions and instructions can be ignored, which greatly reduces the complexity of the analyzer.

Finally, we use the claripy constraint solver of angr to get all the URLs that do not need to pass the authorization process, and analyze the parameters passed when the request is sent to these URLs.

5 Vulnerability Detection

In the vulnerability detection phase, we aim to find opening undesired services (SSH, Telnet), memory corruption and arbitrary command execution vulnerabilities.

To detect undesired services open, we can directly match strings such as *ssh*, *telnet*, *ftp* in the URL generated in the previous module. Once these strings which suggest that a special service open exist, a warning will be reported for further verification. If the service is actually turned on, we can use tool like Hydra to brute force SSH/TELNET/FTP user name and password. Because embedded devices generally run these services as root, this means that a back door exists.

For memory corruption or arbitrary command execution vulnerability, we adopt a dynamic lightweight data tracker detection method.

First, we need to solve the problem of simulating the web server. Chen et al.'s [7] method is to simulating the whole firmware, Costin [1] focuses on getting the firmware web interface into the simulation environment to run. These two solutions both need Linux kernel, and with efficiency and success rate under satisfaction. We are using the User Mode of the qemu emulator directly (such as qemu-arm) to run the web server program. At the beginning, the web server program needs to be patched in the binary level. (1) The IP of the BSS section is changed to a fixed value of 0.0.0.0 in data section, so we can make local network testing. (2) We ignore the external library functions that some vendors customize that has no influence on web services; (3) The functions in libnvram, such as nvram_get, are hooked to make it return fixed number directly, and so on. After chrooting the decompressed the firmware, modifying the web configuration file, running init program (such as rcS), we finally start the web server.

Next we come up with a solution that can quickly find a web sever memory corruption vulnerability and command execution vulnerability. Through hooking sensitive library function, we can monitor the register value when the sensitive function is called and the memory of the address in the register. We summarized more than 40 library functions involving memory operations and command execution (strcpy, system and popen, etc.). We compile the hooking program to generate a dynamic library file, and use preloading to hook library function call. The hooking function will call the original library function after processing.

After hook, we will conduct a series of tests to determine if there are any vulnerabilities. First, we send special payload, such as "AAAAAA" strings, in a URL request, and then perform the following detection:

A. For the command execution vulnerability and the format string vulnerability, we check the parameter registers to determine whether the input data is part of the function's parameters. For example, our data goes to the first parameter register r0 of the printf function, which can tell that there is a format string vulnerability. When our data is the first parameter register r0 or the second parameter register r1 of execve, you can judge that there is a command execution vulnerability here.
B. We check some specific registers or memory data for memory corruption vulnerabilities. For example, in a stack overflow vulnerability, before overwriting the return address on the stack, payload will first overwrite the fp address, that is, the address pointed by the r11 register in ARM architecture. So each time after

hooking memory copy functions (memcpy/strcpy/sprintf), we read the parameter register to check if they contain any constructed payload. If so, the hooking function will call the original library function itself, and then check the r11 pointing memory. It should be pointed out that the stack frame base address is a linked list, so we can record the whole list in the first place, and read each address in the list to check if it is polluted.

We use our own fuzzing script to generate payload as the URL parameters output by the previous module, and record the payload to compare to the data in hook operation to export a vulnerability report. Our report is in json format, which contains the URLs and parameters of the vulnerability, the addresses in the binary, the types of vulnerabilities, and the payload needed to trigger the crash.

6 Automatic Exploit Generation

As early as 2011, Avgerinos [5] put forward a model that use symbolic execution to automate the generation of vulnerability utilization programs-AEG. However, this model has three problems to consider. Firstly, it takes a long time to determine the location of the vulnerability by using the simple symbol execution, and the effect is not ideal. Secondly its exploit generation model is basically aimed at overflow vulnerability, and there is no vulnerability modeling for the format string vulnerability and arbitrary command execution. Thirdly, AEG emphasize particularly on exploiting on x86 architecture. However, embedded devices are mostly based on reduced instruction set such as MIPS and ARM. Their function call convention, parameters passing convention and stack frame structure are very different from x86, so we can't simply transplanted AEG model.

DAEWC proposes a new automatic exploit generation model for embedded devices. We consider the ARM and MIPS instruction set to generate arbitrary read/write, rebound shell and a series of other shellcode. And the final purpose is to hijack the web sever program control flow to make it execute our shellcode.

In fact, automatic exploit generation requires a separate modeling of different vulnerability types. Based on vulnerability types and payload to trigger crash detected in previous module, we use the following different ways to generate exploit:

1. For arbitrary command execution vulnerability, the values of the URL parameters become directly a part of the command execution function's parameters, such as the 'system' function. We can replace the parameter value of any command: reboot, cat/etc/passwd and so on. In order to make these commands execute smoothly, we can add a semicolon to avoid interference of other strings in command execution function parameters.
2. For formatting string vulnerability, we first use libformatstr to determine the location of the formatted string parameters in the stack and the length of padding that needs to be fulfill the 32-bit alignment in the stack. Then use %s and %n parameters to achieve arbitrary address read and write. Because the string will have "\x00" truncation, we will put the read-write address at the end of the constructed malicious formatting string.

For overflow vulnerabilities, the following steps need to be taken. Firstly we need to locate the overflow point, which can be obtained according to the report of the vulnerability detection module. Secondly we need to locate the return address. In order to achieve this, we firstly use the qemu to simulate the web sever simulation, and then using ptrace[2] to monitor the running program. After that we changed each byte of the crash trigger payload from the last byte, and resend the payload after every change. After ptrace capture the SIGSEGV signal, we will check PC and stack registers. If value in PC registers has changed, the changed byte's position minus three is the length of needed padding to rewrite the return address in the stack. With padding we can control the PC by adding target address. Since embedded devices rarely open security mechanism like NX, ASLR or Canary of x86 platform, it's easy to take advantage of it. The third step is to determine the layout of the stack. In the second step we already have the padding required to cover the return address and the return address, so we're going to consider shellcode's layout. Because there is no NX and ASLR protection, we can simply put shellcode as part of the input into the stack, and then return to the address in the stack where shellcode located. In the previous step by ptrace, we not only get the position of the return address, but also obtained the stack frame pointer value, so we'll deploy shellcode behind the return address, and then overwrite the return address to be the stack frame pointer value.

7 Evaluation

The Tenda router is popular around the world, and the AC series router has sold hundreds of thousands of units based on conservative estimates. We tested the Tenda AC6 router with the DAEWC, and found that several critical vulnerabilities could be triggered without the need of web credential validation.

Preprocessing. The firmware obtained by preprocessing is based on 32-bit ARM architecture. Web service program is started by HTTPD, web directory is /webroot, and rcS program is found.

Authentication-bypassing URLs Extraction. HTTPD has obvious behavioral pattern when checking the web request credentials. A 302 code response will be returned when authentication failed. We use this as a security policy, and we use the symbol execution module of DAEWC to slice and analyze the HTTPD. We generate 312 slices, and then we get 17 URLs that do not require authorization.

Vulnerability Detection. We found a URL that can open telnet service the regular expression matching process in the previous step and found two command execution vulnerabilities and four overflow vulnerabilities. In the manual vulnerability verification process, we also surprisingly found a password reset vulnerability based on the report of the DAEWC.

[2] http://man7.org/linux/man-pages/man2/ptrace.2.html.

Authentication-bypassing URLs Extraction. DAEWC generated successfully the exploit of the command execution vulnerability and exploit of the stack overflow vulnerability. In view of vendor safety, we have not disclosed these exploit.

Besides AC6 router, we also use DAEWC to analyze other series of Tenda routers and PHICOMM router. We found 9 zero day vulnerabilities that has never been disclosed. we has submitted them to CVE (CVE-2017-9138, CVE-2017-9139, CVE-2017-11495) [8–10] classified by different vendor and vulnerability type.

Acknowledgment. This work is supported in part by National High Technology Research and Development Program of China (No. 2015AA016004), the National Key R&D Program of China (No. 2016QY04W0802).

References

1. Costin, A., Zarras, A., Francillon, A.: Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 437–448. ACM (2016)
2. http://angr.io/
3. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2015)
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI, vol. 8, pp. 209–224 (2008)
5. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: AEG: automatic exploit generation. In: Proceedings of the Network and Distributed System Security Symposium, February 2011
6. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 380–394. IEEE (2012)
7. 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)
8. CVE-2017-9138. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9138
9. CVE-2017-9139. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9139
10. CVE-2017-11495. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11495