Exploiting Memory Corruption Vulnerabilities in Connman for IoT Devices

K. Virgil English, Islam Obaidat, and Meera Sridhar

Department of Software and Information Systems, UNC Charlotte, Charlotte, NC, USA

{kenglis8,iobaidat,msridhar}@uncc.edu

Abstract—In the recent past, there has been a rapid increase in attacks on consumer Internet-of-Things (IoT) devices. Several attacks currently focus on easy targets for exploitation, such as weak configurations (weak default passwords). However, with governments, industries, and organizations proposing new laws and regulations to reduce and prevent such easy targets in the IoT space, attackers will move to more subtle exploits in these devices. Memory corruption vulnerabilities are a significant class of vulnerabilities in software security through which attackers can gain control of the entire system. Numerous memory corruption vulnerabilities have been found in IoT firmware already deployed in the consumer market.

This paper presents an approach for exploiting stack-based buffer-overflow attacks in IoT firmware, to hijack the device remotely. To show the feasibility of this approach, we demonstrate exploiting a common network software application, Connman, used widely in IoT firmware such as Samsung smart TVs. A series of experiments are reported on, including: crashing and executing arbitrary code in the targeted software application in a controlled environment, adopting the attacks in uncontrolled environments (with standard software defenses such as W⊕X and ASLR enabled), and installing publicly available IoT firmware that uses this software application on a Raspberry Pi. The presented exploits demonstrate the ease in which an adversary can control IoT devices.

I. INTRODUCTION

Vulnerabilities in consumer *Internet-of-Things* (IoT) devices are quickly becoming rampant [1–7]. Operational factors such as low power consumption, business factors such as the need to keep the cost of the device low and the time to market, and many consumer IoT device manufacturers failing to adopt reasonable security practices, lead to the usage of IoT devices without proper security protection [8–12].

While research efforts in IoT security are invested extensively into IoT system and network security solutions [4], [13], [14], a NIST report indicates that 77% of all reported vulnerabilities are in the software applications, not in the operating systems or networks [15].

Recent work shows the increasing prevalence of *memory corruption* vulnerabilities in IoT devices (cf., [16]), largely due to 66% of embedded software development using the C language [17], [18], which is highly susceptible to such vulnerabilities [19]. Generally, *buffer-overflow attacks* are among the most widespread exploitations of memory corruption vulnerabilities [20]. In this class of attacks, stack-based buffer-overflows are among the most dangerous [21].

In this paper, we report on a series of *proof-of-concept* (PoC) exploits for a known stack-based buffer-overflow vulnerability,

published as CVE-2017-12865 [22], in Connman [23], an open source lightweight network connection manager software that is widely used in many IoT firmware such as Nest thermostats [24], NAO robots [25], [26], and most smart devices from Samsung such as smart watches and smart TVs. The buffer-overflow vulnerability exists within the DNS proxy module of Connman. This allows a crafted DNS response to be sent to the DNS proxy module of Connman, which can result in *denial-of-service* (DoS) or *remote code execution* (RCE). To our knowledge, there is no existing PoC available for CVE-2017-12865.

We first report on a series of PoC exploits conducted in a controlled environment. For this purpose, we install Connman in a virtual machine running Ubuntu 16.04 LTS (compatible with x86) and Ubuntu Mate 16.04 LTS (compatible with ARMv7), and use gdb in both OSes to construct attacks that are able to bypass different levels of two standard software memory protection defenses, *Writable XOR Executable* (W⊕X) [27], [28] and *Address Space Layout Randomization* (ASLR) [29]. We begin our experiments with a traditional code-injection attack. Without W⊕X or ASLR enabled, we send an unexpected long string in DNS responses that consist of NOP sled, shellcode, and repeated return address [30]. Then, we assume W⊕X protection is enabled, and construct an attack to bypass W⊕X. Our last experiment assumes the device has both W⊕X and ASLR enabled. To bypass these protection mechanisms, we construct a *code-reuse attack* (cf., [31]). Our proof-of-concept exploits demonstrate a method for achieving arbitrary code execution (e.g., spawning a root shell) through this vulnerability.

Having successful PoC exploits that bypass W⊕X and ASLR in a controlled environment, we conduct the previous series of PoC exploits in a non-controlled environment. To achieve this, we install publicly available IoT firmware that uses Connman on Raspberry Pi. Next, to conduct a realistic man-in-the-middle attack scenario, we use a *Wi-Fi Pineapple* [32], a wireless network tool that is capable of impersonating any Wi-Fi SSID scan request, to hijack the IoT traffic and route it to a fake DNS server that constructs malicious DNS responses to trigger the vulnerability.

Our main contributions include:

- We construct a series of PoC exploits that can execute arbitrary commands for a known stack-based buffer-overflow vulnerability in Connman, a software application that is used widely in IoT devices.
- Our exploits are constructed for Connman running on x86 and ARMv7, under different levels of memory pro-

tection techniques, including W⊕X and ASLR.

- We adopt these exploit scenarios in non-controlled environments, against IoT devices equipped with `Connman`.

Our experiments demonstrate that: (i) it is easy for adversaries to exploit these types of vulnerabilities even if IoT devices use standard countermeasures such as W⊕X and ASLR, and (ii) such vulnerabilities persist, even months after being discovered, especially in up-to-date IoT devices. We hope that this study will bring better awareness of these problems to the software security community and lead to developing more robust defenses.

Buffer-overflow vulnerabilities are extensively targeted in traditional systems, and we expect will become popular for exploits in IoT ecosystems. Such attacks are not yet popular in the IoT ecosystem since attackers are able to conduct very low-cost attacks that exploit weak configurations, such as default passwords. However, we expect that as these basic security holes such as default or weak passwords are fixed, and governments and IoT vendors start providing protection and awareness about the above [33–35], code-injection/code-reuse attacks will provide fertile ground for IoT attackers [36].

The rest of this paper is organized as follows: Section II presents an overview of `Connman` and the targeted vulnerability; Section III reports on the construction of the PoCs, including a discussion of the main challenges that were encountered; Section IV describes suggested mitigations, and current work towards them; Section V discusses adapting our approach to exploit other vulnerabilities; Section VI discusses related work, and Section VII presents future work.

II. OVERVIEW

In this section, we present an overview of `Connman` and its buffer-overflow vulnerability. We selected `Connman` from recent work that surveys control-flow hijacking vulnerabilities in IoT devices [16]. We chose `Connman` because: (1) it is a networking application that is used in a large number of IoT devices, (2) it contains a stack-based buffer-overflow vulnerability, and (3) adversaries can compromise and control IoT devices with this vulnerability remotely.

Connman: `Connman` [23] is an open-source network management daemon with a lightweight, low overhead design, ideal for Linux embedded devices. `Connman` can manage connections over Wi-Fi, Ethernet, cellular, and Bluetooth, using plug-ins such as `oFono` [37] and `Bluez` [38]. This plug-in based architecture provides customizable implementations, and allows network functions such as DHCP and DNS to be handled by one daemon instead of multiple. `Connman`'s versatility comes from its ability to be used as a whole or in parts, allowing `Connman` to work side-by-side with other network management software to form a comprehensive suite. The lightweight and customizable design of `Connman` make it ideal for use in IoT devices.

Several smart devices and OSes currently use `Connman`, such as Nest thermostats [24], Mer [39], Yocto [40], Jolla OS [41], Ostro [42], Sailfish OS [43], Tizen OS [44], [45],
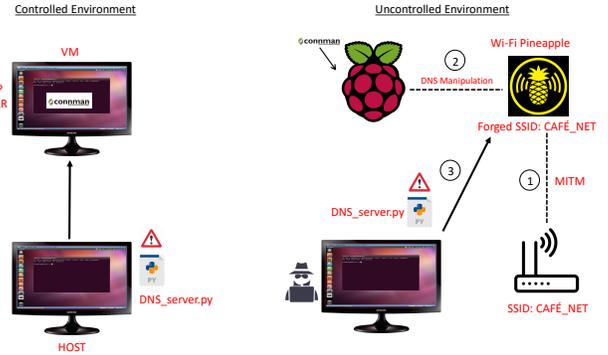

Fig. 1. Experimental Setup for PoCs

used in most Samsung smart devices including tablets, smartphones, netbooks, smart TVs, and other smart home devices. `Connman` is also the default network manager for NAO OS [26], used in NAO robots, and many IoT development platforms, such as Intel Galileo [46].

Vulnerability: CVE-2017-12865 [22] (henceforth referenced as "the vulnerability") is a stack-based buffer-overflow vulnerability in the DNS-proxy feature of `Connman` 1.34 and earlier. DNS-proxy sends DNS queries from `localhost` to external DNS servers. The received DNS response, if malformed, can cause a DoS or RCE. This vulnerability is located in the `dnsproxy.c` file of `Connman`.

The `parse_response` function of `dnsproxy.c` has a buffer, `name`, which is used when `Connman` expands a compressed DNS name to cache DNS responses of `type A`, which is a 32-bit IPv4 lookup response, or `type AAAA`, a 128-bit IPv6 lookup response [47]. For this purpose, the `parse_response` function calls `get_name` function. Listing 1 shows the call of `memcpy` libc function in `get_name` function with no proper length checking. The pre-defined limit of the buffer `name` size is 1024. A DNS response crafted with a length over this limit overwrites adjacent memory, allowing DoS and RCE attacks to be executed.

```
  name[(*name_len)++] = label_len;
1 memcpy(name + *name_len, p + 1, label_len + 1);
2 *name_len += label_len;
```

Listing 1. Unchecked `memcpy` used in `get_name` function

The vulnerability has been patched in subsequent versions of `Connman` as of August 2017 [48]. Size checks have been added to the data to be copied into the `name` buffer, returning out if the size is larger than the buffer size. The fixed version, version 1.35, is the latest release. However, all prior versions (1.34 and below) contain this vulnerability [49].

III. EXPERIMENTS

An attacker-controlled DNS server can exploit the vulnerability by sending crafted messages to execute a DoS attack. We construct a PoC to simulate this scenario. We create a simple DNS server to act as a proxy for `Connman`'s DNS request. On receiving a request from `Connman`, our DNS server sends

a `Type A` response with length greater than the `name` buffer size. When `Connman` decompresses and adds the message to the `name` buffer, the application crashes.

For this experiment, we use a 32-bit operating system, as it more accurately portrays the limited resource environment of most IoT devices. In this section, we report on six usable attacks we construct that exploit this vulnerability. The six attacks are conducted using three protection levels: (1) no protections enabled, (2) (W⊕X) enabled, and (3) W⊕X with ASLR enabled. We bypass each protection level on two different architectures: Intel x86 architecture running Ubuntu 16.04, and on a `Raspberry Pi` 3 Model B v1.2 with ARMv7 architecture running Ubuntu Mate, a popular embedded OS. Each exploit construction presents its own set of requirements and challenges, which we discuss in the following sections. Our experiments demonstrate that unpatched versions of the vulnerability are susceptible to attack regardless of what OS-level protections are present. Specifically, we found three major embedded operating systems that still contain vulnerable versions of `Connman`: the Yocto project, a popular embedded OS development platform, compiles distributions with `Connman` 1.31; OpenELEC, a popular media streaming OS, comes with `Connman` 1.34, the last vulnerable version of `Connman`; Tizen OS, a bedrock for Samsung devices, utilizes a vulnerable version of `Connman` up until version 4.0. Following our successful exploitation of the vulnerability in a controlled environment, we plan to target all three OSes for simulated attacks.

Experimental Setup: The goal of all these experiments is to interrupt the flow of `Connman` and spawn a root shell. `Connman` natively runs with root permissions, so no permission change is required for this to occur. All code presented is written in Python.

Due to the nature of the vulnerability in `Connman`, all exploits have to be sent over DNS responses, specifically `Type A` or `Type AAAA` responses. We select `Type A` for its universality. A basic DNS server is created to respond to requests. The DNS responses must appear legitimate, otherwise `Connman` dumps the packet as a bad response and never enters the vulnerable portion of code. Our DNS server must first craft a legitimate response header to each DNS query, then place the exploit code into the returned record itself. A simple Python DNS server is created to perform this function, which copies the relevant portions of the query from the target machine's packet, inserts the proper flags, and encodes the malicious code into the record response.

A. No Protections Enabled

1) Intel x86: Our first experiment constitutes a code-injection exploit on the x86 architecture. W⊕X, ASLR, and stack protectors (canaries) are all disabled through compilation options. With no system protections enabled, we utilize a standard buffer-overflow attack [30]. Upon confirmation that `Connman` does crash as expected with the oversized DNS `Type A` response, we begin examining `Connman`'s runtime activities using a debugging tool. The tool we select for

use is the GNU Project debugger, `gdb`, in Ubuntu. Using `gdb`, we are able to isolate the sections of memory occupied by the stack of the `parse_response` function. After discovering these memory locations, we construct a payload to exploit the vulnerability. This payload consists of a NOP sled [30], consisting of repeated `\x90` (`xchg eax, eax`) instructions, followed by assembly instructions to execute `execve("/bin/sh", ["/bin/sh"], NULL)`, and finishes with repeating new return addresses pointing at the NOP sled. We pass this payload to `Connman` via our man-in-the-middle DNS server, and successfully spawn a root shell on the targeted machine. Subsequently, the same attack is successfully carried out against `Connman` on the x86 architecture without the aid of the `gdb` debugging tool.

2) ARMv7: : In the next exploit, we recreate the Intel x86 buffer-overflow attack on ARMv7 architecture. We make the following small changes to adapt the x86 attack to ARMv7 (`Raspberry Pi` v3b+). Unlike x86, ARMv7 does not have a single-byte NOP command. Instead, we use a 4-byte code string, `\x01\x10\xa0\xe1`, which on ARMv7 translates to `mov r1, r1`, an effect-free operation. We modify the shellcode to utilize ARMv7 assembly instructions instead of x86. On the ARMv7 platform, a few memory locations that are overwritten which `Connman` expects to be `NULL` in a check prior to the `pop {pc}` command in `parse_response`. We must place `NULL` values in these locations for the ARMv7 exploit. This exploit is successful on a default configuration of `Connman`, installed with developer provided `CFLAGS` and ASLR disabled.

B. W⊕X Enabled

1) Intel x86: Our next experiment targets a successful exploit with stack execution protections enabled. We use a *return-to-libc* [50] attack to create the attack on x86. A return-to-libc attack leverages the fact that, without ASLR enabled, the location of `libc` in system memory is static. In this manner, functions can be accessed through the `libc` library that the target executable does not call. In our case, it allows us to utilize `system()`, which is not referenced within the `Connman` program. With access to the victim machine, we again utilize `gdb` to examine the memory layout of `Connman` as it executes. Using `gdb`, we find the location of the `libc` functions `system()` and `exit()`, allowing us to construct the payload. This payload utilizes explicit calls to the `__libc_system()` function to execute `"/bin/sh"` This attack is successful against x86 architecture.

2) ARMv7: Our next experiment is against the ARMv7 architecture with W⊕X protection. Unlike Intel x86, ARMv7 does not pass arguments directly from the stack. Therefore, a traditional return-to-libc attack is not possible, as the attack does not have the ability to manipulate registers. Instead, we utilize a *gadget*-based approach for this exploit. *Gadgets* [50] are small blocks of assembly code, ending in `ret` commands (Intel x86), or a branch or `pop pc` command (ARMv7), that chain several instruction sequences together.

Our approach can be delineated into three steps: (1) we locate a gadget that can be used to load arguments into registers; (2) we use this gadget to load the `r0` and `r1` registers with the appropriate arguments; and (3) we load the `pc` register with the memory address of `execlp@plt`, the function we utilize to switch `Connman`'s execution and spawn a root shell.

To find an appropriate gadget to load the needed registers, we utilize the open-source program `ropper` [51]. Among other functionality, this program allows an easy way to display information about a binary, including it's compiled assembly instructions (gadgets). With `ropper`, a suitable gadget is found, in our case `pop {r0, r1, r2, r3, r5, r6, r7, pc}`.

Next, we use this gadget to load the `r0` and `r1` registers with the appropriate arguments. For our exploit, these arguments are `"/bin/sh"` and `NULL`, respectively. The gadget in step (1) is used to pop these values from the stack into the correct registers.

Finally, we utilize the gadget to invoke `execlp@plt`, a *Procedure Linkage Table* (PLT) reference. The PLT allows the program to make calls from its shared text section to external functions by loading the memory location of libraries at run-time and linking to specific functions using their offset from the start of the memory block [52]. This allows the program to make external function calls without knowing the memory address of the program, a requirement for an executable to function with ASLR enabled. While ASLR is not enabled for this exploit, making a call in this manner allows us to utilize the same code in our following ASLR exploit. This call is made by using the gadget in step (1) to pop the value into the `pc` register.

```
        . . .
1       +'\xb1\x12\x01\x00' #Pop r0−r7, pc
2       + '\xe4\x53\xd8\x76' 1 r0, static /bin/sh
3       +'\x00\x00\x00\x00' # r1, NULL to terminate
            execlp argument array
4       +'\x88\xe9\xff\x7e' # r2
5       +'\x97\xff\xff\xff' # r3
6       +'\xc4\xd2\xff\x7e' # r5
7       +'\x59\x58\xf0\x76' # r6
8       +'\x00\x00\x00\x00' #r7 placeholders
9       +'\xd0\xb2\x01\x00') #pc to execlp@plt
```

Listing 2. `execlp` ROP chain for ARMv7

Listing 2 shows the code for this exploit. Line 1 shows the explicit call to the gadget to load registers `r0`, `r1`, and `pc`. This gadget pops the next 32 bytes of memory, separating each 4 bytes and placing them into separate registers. The gadget loads many more registers than are needed simply to call `execlp@plt`. However, utilizing a gadget with fewer registers results in a `SIGSEV` in the `parse_rr` function, as the locations occupied by line 7 and line 8 overwrite portions of memory required for a `mvn.w` call in `parse_rr`. This gadget is selected to overcome that obstacle.

Line 2 and Line 3 are the 4 byte values assigned to `r0` and `r1` respectively. Line 2 is the static memory location of the full string `/bin/sh`, loaded into `r0`. This string is located in the `libc` portion of memory, and is not randomized with ASLR disabled. Line 3 is a 4 byte `NULL` sequence, loaded into

`r1`. These two arguments, `/bin/sh` and `NULL`, are loaded into `r0` and `r1` to be passed to `execlp@plt`.

The next 20 bytes (Lines 4-8) are placeholder values. Starting at line 4, the gadget pops these values into `r2`, `r3`, `r5`, `r6`, and `r7` respectively. We select the values for these placeholders after examining the `parse_rr` function during run-time, and seeing the expected values for those positions.

Finally, the gadget loads the `pc` with `Connman`'s PLT reference to `execlp`, which is the 4 bytes located on Line 9. This executes the `execlp@plt` function call. `execlp` is a member of the `exec` family, similar to `execve`. The main differences are `execlp` allows the use of relative rather than explicit addresses for the file to be executed, and allows a variable number of arguments to be passed. For this reason, the final argument must be a `NULL` to indicate the end of the passed arguments [53], which is the reason for line 3.

With this exploit, we successfully spawn a root shell in `Connman` utilizing `gdb`. The exploit is then successful on a default installation of `Connman` without the aid of `gdb` and with the developer default `CFLAGS` enabled. ASLR is not enabled for this exploit.

C. W⊕X and ASLR Enabled

1) Intel x86: Our third experiment involves constructing an exploit with both W⊕X and ASLR enabled. Enabling ASLR prevents our previous attack strategy, `ret-to-libc`, from working, as ASLR randomizes the location of `libc` at run-time. Without a memory leak exposing the current position of the library, it becomes difficult for an attacker to reliably guess the position of needed functions within the library to make explicit calls.

To circumvent this protection, we employ a *return-oriented programming* (ROP) attack [50]. ROP attacks work by chaining gadget calls to redirect the execution of a program to cause arbitrary actions. Each of these gadgets ends in a `ret` instruction, allowing these gadgets to be chained together to perform complicated, multi-step instructions similar to a code-injection.

Our ROP attack is conducted in three steps: (1) we locate necessary gadgets and the characters `"/bin/sh"` in `Connman` memory; (2) using these gadgets and `memcpy@plt`, we copy the characters into `.bss` memory block to form the string `/bin/sh`; and (3) we invoke `execlp@plt` using the crafted string to spawn a root shell.

On ARMv7, we utilize the program `ropper` to find the required gadgets, as discussed in §III-B2. On x86, we use an open-source program called `ROPgadget` [54]. `ROPgadget` provides similar functionality to `ropper`, displaying compiled binary information. To successfully craft this exploit, a gadget in the form of `pop pop pop ret` is found. This gadget is selected for its ability to remove the next 16 bytes from the stack. The gadget reference comes directly after the `memcpy@plt` reference, allowing the instruction pointer to remove the arguments (Listing 3 Lines 3-6) before proceeding on to the next call. The first three `pop` commands remove the arguments from the stack, and the final `pop` removes four

```
          ...
1        + '\xf0\x29\x05\x08'  #execlp@plt
2        + '\x14\x14\x14\x14'  #random bytes
3        + '\x01\x02\x12\x08'  #bss string (\bin\sh)
4        + '\x00\x00\x00\x00'  #arg array (null)
```

Listing 4.   memcpy chain ROP exploit

bytes of random values following the memcpy arguments, required for the add esp, 0xc; pop ebp; at the end of the memcpy function.

Next, we combine this gadget with memcpy@plt calls to craft the string /bin/sh in the .bss portion of memory. Common practice is to use strcpy to accomplish this goal. However, Connman contains no references to strcpy, instead replacing them on compilation with _strpy_chk. Fortunately, since Connman contains references to memcpy, we still can craft the needed string. The memcpy function takes three arguments: src, dest, and length. With the x86 architecture, these variables can be passed via the stack.

Utilizing memcpy, our goal is to put the string "/bin/sh" somewhere in Connman's memory. The .bss section is selected, as it is uninitialized memory and therefore not randomized with ASLR. Single character references are found in Connman using the -memstr argument in ROPGadget. Using these locations as the source, offsets from the beginning of the .bss section of memory as the destination, and a length argument of 1, we successfully craft the "/bin/sh" string in the .bss.

```
          ...
1        + memcpy #memcpy@plt reference
2        + ppppr #pop pop pop pop ret
3        + '\x01\x02\x12\x08'  #bss + 1
4        + '\x54\x81\x04\x08'  # '/'
5        + int_val # int = 1
6        + garbage # garbage for add esp, 0xC; pop ebp
             ; @ memcpy end
```

Listing 3.   memcpy chain ROP exploit

Listing 3 shows the memcpy portion of the ROP chain.

The +memcpy and +pppr references are to .text memory locations of these gadgets, +int_val is a variable with a hex value of one (0x00000001), and +garbage is 4 bytes of random values, specifically \xAA\xAA\xAA\xAA. These random values are required for the aforementioned code at the end of memcpy. The code snippet in Listing 3 shows the portion of code used to copy the "/" character into the .bss portion of memory. This code block is repeated for each character of the string /bin/sh, increasing the .bss offset by one each time until the entire string is copied into attacker-controlled memory.

Having successfully placed the required string into an accessible portion of memory, the final step is to craft the call that utilizes this string to spawn a root shell. To do this, a execlp@plt call is used, as in §III-B2. Listing 4 shows the execlp portion of the exploit.

The execlp@plt is the .text memory location of execlp@plt, the random bytes on line 2 is utilized as a spacer, since x86 architecture skips 4 bytes when looking for arguments on the stack, followed by the memory location of the string we copied into .bss and a memory equivalent

NULL argument. The full ROP exploit code is successful in spawning a root shell on the target machine.

2) ARMv7: As with x86, we utilize ROP for our ASLR bypass on the Raspberry Pi. The general structure of the exploit is the same as in §III-C1 (we locate the required gadgets, copy the needed string into the .bss portion of Connman's memory using memcpy, and call execlp@plt). However, there are three major differences.

First, ARMv7 arguments must be loaded into registers. This is addressed in the same manner as in §III-C1. As previously stated, memcpy takes three arguments: dest, src, and length. Therefore, the same gadget as §III-C1 is utilized to load the proper arguments into registers r0, r1, and r2, respectively. The source of these arguments is the same as the x86 exploit, although on ARMv7 .bss+4 is used.

Second, the length of the gadget required makes copying the full /bin/sh string impossible. The length of the gadget prevents more than three calls from executing, as after the third call in the ROP chain the exploit is overwritten by data from a subsequent legitimate function reference. If attempting to copy the full /bin/sh string, the exploit terminates after copying /bi and a SIGSEV occurs. However, as mentioned previously, execlp@plt has the ability to use relative file addresses. This allows us to copy only sh into the .bss portion of memory, leaving one function call available to reference execlp@plt.

The final and most challenging difference is the lack of a ret; function in ARMv7 assembly. In §III-B2, we use a pop pc command to invoke execlp@plt. A pop pc command will not by itself return to the previous location. Instead, ARMv7 utilizes branch-link calls, such as bl (*branch-link*) or blx (*branch-link-exchange*) [55]. Branch-link stores the address of the next instruction from the call in r14, also known as the lr or *link register*, and changes program execution to the passed memory location. The called function can then return to the previous location using a branch command (b or bx), or if necessary push the value of r14 to the stack, and return using a pop pc command. A suitable gadget has to be found to facilitate this behavior in our exploit. We find a blx r3 gadget in Connman that branch-links to the value stored in r3. We load r3 with the memory location of execlp@plt and the pc register with the blx r3 gadget.

```
          ...
1        +'\xb1\x12\x01\x00'  #Pop r0-r7, pc
2        +'\xc4\x9d\x0b\x00'  #1 r0, bss+4
3        + '\x68\x01\x01\x00'  #2 r1, 's'
4        +'\x01\x00\x00\x00'  #3 r2 int=1
5        +'\x98\xb7\x01\x00'  #4 r3 = memcpy@plt
6        +'\xc4\xd2\xff\x7e'  #5 r4
7        +'\x59\x58\xf0\x76'  #6 r5
8        +'\x00\x00\x00\x00'  #7 r6 - all placeholders
             to prevent sigsev in parse_rr
9        +'\x1d\xc3\x01\x00'  #8 pc to blx r3
10       + '\x00\x00\x00\x00'  #offset characters for
             blx
```

Listing 5.   memcpy chain ROP exploit

The code to copy the string into `.bss` is a repeated occurrence of the contents of Listing 5, as described below.

On Line 1, the same gadget for loading the registers is used as in §III-B2. The gadget loads register `r0` (Line 2) with the destination argument for `memcpy`, in our case `.bss+4`. We then load `r1` (Line 3) with the source argument for `memcpy`, in this snippet the location of an `'s'` character in the `.text` portion of `Connman`'s memory. Next, the gadget loads Register `r2` (Line 4) with the `length` argument for `memcpy`, an integer of value 1. Lastly, we load `r3` (Line 5) with the memory address of `memcpy@plt`. We again load `r5`, `r6`, and `r7` with placeholders to prevent `SIGSEV` in `parse_rr`.

On Line 9, the gadget loads the `pc` with the memory location of the gadget `blx {r3}` which itself branches to the location stored in `r3`, `memcpy@plt`. The final `NULL` bytes on line 10 are an offset for `blx`, which attempts to return to 4 bytes after its calling in our scenario.

Following a string of these calls to put `"sh"` into the `.bss`, we make a call to `execlp@plt` in the same manner as in Listing 2, with the exception of placing the address of `bss+4` in Line 2.

This exploit is successful in spawning a root shell under multiple circumstances, with or without the aid of `gdb`. We are able to exploit `Connman` with ASLR enabled as per system default settings after boot, and with the developer default `CFLAGS` enabled. This exploit is successful with no changes to the compilation of `Connman`, the system ASLR or other settings, or the `Connman` code.

D. Wi-Fi Pineapple

Having successfully bypassed the targeted protection mechanisms, our next experiments focus on conducting the attack remotely with a man-in-the-middle DNS server. To accomplish this, we use a Wi-Fi Pineapple [32] to mimic a malicious access point. The Wi-Fi Pineapple is a mobile access point designed for network reconnaissance and penetration testing. With the Pineapple, we simulate a specific class of attacks that can trigger this vulnerability, namely using a rogue access point or hijacking device traffic. However, this vulnerability can be triggered by other classes of attacks. For instance, an attacker can use a malicious domain and lure a target user to their site, then use the domain's DNS server to respond to queries with the exploit code. A cache poisoning attack could be used to force traffic to a domain, at which point exploit code designed to create a botnet could be sent to visitors, allowing a recreation of the Mirai attack from 2016 [10].

With the Wi-Fi Pineapple, we first remotely exploit `Connman` on the x86 architecture. The goal of this experiment is to perform the exploit in conditions similar to those a malicious adversary might use. We set the Pineapple to broadcast a trusted network SSID, and configure it to utilize DHCP to assign our malicious DNS server to clients. The Wi-Fi Pineapple is able to broadcast a stronger signal than the legitimate access point, causing our targeted machine to switch its connection. Once this had been achieved, the malicious DNS server is able to intercept all DNS requests being sent

from the target machine, and provides responses containing the exploit code. On the x86 architecture, the only attack we attempt is the basic stack smash, as a proof of feasibility for the man-in-the-middle setup.

Once we show the Pineapple is able to be used as a man-in-the-middle DNS server, we perform all three ARMv7 exploits against the `Raspberry Pi` with no configuration changes except connecting to the SSID broadcast by the Pineapple. The only network configuration set in the `Raspberry Pi` under Ubuntu Mate is to utilize DHCP and automatic DNS server via DHCP. All three ARMv7 exploits are successful under these conditions.

IV. SUGGESTED MITIGATIONS

The most immediate mitigation for memory corruption vulnerabilities is patching. However, this puts the onus on the code-producer to remedy the issue and ensure a critical mass of devices are updated by users, an obligation developers and users have struggled with in the past.

Hardware-supported *control flow integrity* (CFI) techniques (e.g., [56]) show promise towards securing IoT firmware against code-reuse attacks, such as the ones we demonstrate. Hardware-supported security has now been widely introduced in many embedded architectures. For example, TrustZone [57], ARM's hardware-based security technology (ARM spans 60% of the current embedded device market [58]), is being adapted into most ARM processor families, and TrustZone-enabled devices are expected to reach 1 Trillion by 2035 [59]. Intel and AMD have also introduced similar technologies [60], [61]. As the next step in our research, we plan to adapt the CFICaRE [56] technique to our IoT devices running `Connman` to gauge the efficacy of protection against code-reuse attacks we created, and extend the approach if necessary.

Artificial software diversity (ASD) (cf., [62]) protects against code-reuse attacks through adding probabilistic protection to a binary by randomizing program implementations (i.e., randomizing program data space or control-flow sequences) [62]. This probabilistic protection implies that a successful attack is not guaranteed to work on multiple systems, preventing mass attacks from occurring.

Amongst artificial software diversity techniques, *compile-time software diversity* (cf., [63]) might be best suited for IoT devices. Compile-time software diversity moves the additional performance overhead away from the IoT device and into the developer's space, limiting the impact on device performance. In particular, *code-sequence randomization* [63], which makes use of standard code-rewriting compilation time techniques such as *call-inlining* and *instruction scheduling*, and modifies them so that the output binary is randomized from one compilation to the next, requires minimal code-producer and code-user cooperation to implement, making it ideal for the "it just works" mindset of IoT.

Equivalent-instruction randomization [62] is another ASD technique that takes advantage of semantically-equivalent instruction sets to randomize binaries. An equivalent-instruction randomization framework for IoT firmware is currently under

development at UNC Charlotte. Specifically, we are using a combination of equivalent-instruction randomization and other randomization techniques to randomize compiled programs into dynamically equivalent binaries.

V. ADAPTING FOR OTHER VULNERABILITIES

Our code can work out-of-the-box (with minimal modification) against DNS-based overflow vulnerabilities such as CVE-2017-14493, CVE-2018-9445 and CVE-2018-19278. CVE-2017-14493 and CVE-2018-9445 are stack-based buffer-overflows in `dnsmasq` and `systemd`, respectively. CVE-2018-19278 is a buffer-overflow vulnerability in DNS handling in the Digium Asterisk service. Minimal modification includes basic changes such as changing variables to memory addresses suitable for the targeted vulnerability.

With moderate modification, our code can be adapted to work against a range of protocol-based vulnerabilities. Our code is designed to deliver a payload from a DNS server. However, by modifying the packet creation algorithm, along with previously discussed modifications, overflows in other protocols can be targeted. For example, CVE-2019-8985, CVE-2019-9125, CVE-2018-6692 (buffer-overflow vulnerabilities exploitable with HTTP packets) and CVE-2018-20410 (buffer-overflow vulnerability triggered by a crafted TCP packet) can be exploited by a modified version of our code. In theory, any protocol-based overflow vulnerability is susceptible, as long as the code is modified to craft the appropriate packet, rather than a DNS packet, and the memory values are changed to the appropriate context for the targeted vulnerability.

Our general exploit generation and delivery approach can be used for a wide variety of exploits. Our exploit generation approach includes methods for examining program memory to determine the standard behavior of the targeted function, ascertaining the behavior of the targeted function after passing it corrupted code, and discovering potential exploit pathways within the program (e.g., discovering gadgets within the program that can be called for a code-reuse attack), as described in §III. Our novel delivery approach of using the Wi-Fi Pineapple (which facilitates the often formidable task of allowing control of a devices communication pathway without compromising the targeted LAN) can be used effectively in other exploits widely. In theory, any memory-corruption vulnerability exploitable from a remote posture would be able to use our approach.

VI. RELATED WORK

Numerous related works address security issues in embedded systems, and many academic papers attempt to enhance the security of these systems (e.g., [64–72]). However, most of these concern cryptographic properties (e.g. confidentiality, integrity, authentication) and hence do not address the problem of buffer overflow attacks specifically.

Several works in the literature address defenses for memory-corruption vulnerabilities (including in traditional software), such as CFI; we discuss these in detail in §IV.

Other works attempt to make C a safer language (e.g., AddressSanitizer [73] and SAFECode [74]). These approaches utilize *array-bound checks* (tests performed at run-time to ensure array accesses are safe) and usually involve two steps. First, they scan the instruction set of the program for vulnerabilities, then insert array-bounds checks at the vulnerable regions. While these approaches are effective at deterring out-of-bound memory access, they impose significant overhead on compiled programs, which is incompatible as-is for IoT. For instance, AddressSanitizer slows programs by more than 70%, and increases memory usage by more than 200% [75].

Other related works demonstrate how attackers used IoT firmware flaws to compromise these devices. Tsoutsos [76] summarizes different types of attacks that compromise memory corruption vulnerabilities in embedded systems. However, no PoCs are provided or discussed for these attacks in his work. Researchers at Google discovered several buffer overflow vulnerabilities [77] in Dnsmasq [78], a lightweight open-source DHCP server and DNS forwarder that is used in IoT devices to manage the DHCP leases and as a caching DNS stub resolver. They provide several PoC exploits for these vulnerabilities [79]. However, these PoCs are limited to causing a DoS attack in the targeted device that contains the vulnerable version of Dnsmasq. Caceres expands upon this by providing a PoC for a return-to-libc exploit against the Dnsmasq vulnerability [80]. However, while he was successful in bypassing W⊕X protection he was unable to craft a PoC to bypass ASLR. Researchers at Senrio security [81] found a stack-based buffer overflow vulnerability in D-link smart routers firmware (v1.12) [82]. For this vulnerability, a PoC that is able to bypass W⊕X and ASLR on MIPS and ARM architectures by brute-force is available online. However, this PoC requires LAN access and cannot be exploited remotely.

VII. CONCLUSION

Memory corruption vulnerabilities are the most popular security vulnerabilities affecting software systems. Stack-based buffer-overflow is one of the main exploits of memory corruption vulnerabilities that can cause a DoS or RCE. These vulnerabilities are also introduced in IoT firmware. In this paper, we demonstrate several attack scenarios to exploit a stack-based buffer-overflow vulnerability in `Connman` for IoT firmware. In these scenarios, we conduct attacks against two different architectures, x86 and ARMv7, bypass different levels of memory protections, W⊕X and ASLR, in a controlled environment. These exploits are then adopted in a non-controlled environment.

In future work, we plan to attack popular IoT OSes TizenOS, OpenELEC, and Yocto builds on the ARMv7 architecture. We also plan to build an automated exploit generator for stack-based buffer-overflow attacks in IoT devices. In addition to shifting to attacking IoT OSes with our current exploits, we plan on developing a light-weight stack memory protection mechanism for IoT devices that address the main challenges in these devices, such as resource constraints.

REFERENCES

[1] M. A. Khan and K. Salah, "IoT security: Review, blockchain solutions, and open challenges," *Future Generation Computer Systems*, vol. 82, pp. 395–411, 2018.

[2] D. Celebucki, M. A. Lin, and S. Graham, "A security evaluation of popular internet of things protocols for manufacturers," in *2018 IEEE International Conference on Consumer Electronics ICCE,*, 2018, pp. 1–6.

[3] C. Bradley, S. El-Tawab, and M. H. Heydari, "Security analysis of an IoT system used for indoor localization in healthcare facilities," in *Systems and Information Engineering Design Symposium SIEDS*, 2018, pp. 147–152.

[4] H. Haddadi, V. Christophides, R. Teixeira, K. Cho, S. Suzuki, and A. Perrig, "SIOTOME: An edge-isp collaborative architecture for IoT security," in *Proceedings of the 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*, 2018.

[5] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, Boston, MA, 2018, pp. 147–158.

[6] R. Sairam, S. S. Bhunia, V. Thangavelu, and M. Gurusamy, "NETRA: Enhancing IoT security using nfv-based edge traffic analysis," *arXiv preprint arXiv:1805.10815*, 2018.

[7] H. Mouratidis and V. Diamantopoulou, "A security analysis method for industrial internet of things," *IEEE Transactions on Industrial Informatics*, 2018.

[8] S. Demetriou, "Analyzing & designing the security of shared resources on smartphone operating systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2018.

[9] E. Bertino and N. Islam, "Botnets and Internet of Things security," *Computer*, no. 2, pp. 76–79, 2017.

[10] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzia, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 1092–1110.

[11] M. Barnes, "Alexa, are you listening?" http://tinyurl.com/y75t2hzh, August 2017.

[12] I. Clinton, L. Clinton, and S. Banik, "A survey of various methods for analyzing the amazon echo," 2016.

[13] J. Mao, Q. Lin, and J. Bian, "Application of learning algorithms in smart home IoT system security," *Mathematical Foundations of Computing*, vol. 1, no. 1, pp. 63–76, 2018.

[14] M. Burhanuddin, A. A.-J. Mohammed, R. Ismail, M. E. Hameed, A. N. Kareem, and H. Basiron, "A review on security challenges and features in wireless sensor networks: IoT perspective," *Journal of Telecommunication, Electronic and Computer Engineering JTEC*, vol. 10, no. 1-7, pp. 17–21, 2018.

[15] P. E. Black, L. Feldman, and G. A. Witte, "Dramatically reducing software vulnerabilities," https://tinyurl.com/ybqtc7fj, May 2017.

[16] A. Mohanty, I. Obaidat, F. Yilmaz, and M. Sridhar, "Control-hijacking vulnerabilities in IoT firmware: A brief survey," in *Proceedings of the 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*, 2018.

[17] S. Cass, "The 2015 top ten programming languages," *IEEE Spectrum, July*, vol. 20, 2015.

[18] U. E. Group, "2015 embedded markets study," http://tinyurl.com/y9wxg3u7.

[19] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits." in *Proceedings of the USENIX Security Symposium*, vol. 12, no. 2, 2003, pp. 291–301.

[20] A. Lautenbach, M. Almgren, and T. Olovsson, "What the stack? on memory exploitation and protection in resource constrained automotive systems," in *Critical Information Infrastructures Security*, Cham, 2018, pp. 185–193.

[21] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao, "To detect stack buffer overflow with polymorphic canaries," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks DSN*, 2018, pp. 243–254.

[22] "CVE-2017-12865," Available from MITRE, CVE-ID CVE-2017-12865, Aug. 29 2017. [Online]. Available: http://tinyurl.com/y2t3hndq

[23] "Connman," https://01.org/connman.

[24] "Nest," https://nest.com/thermostats/.

[25] "NAO robot," https://www.softbankrobotics.com/emea/en/nao.

[26] "OpenNAO - NAO OS," http://tinyurl.com/y5h8tvf8.

[27] P. Team, https://pax.grsecurity.net/.

[28] Microsoft, "A detailed description of the data execution prevention DEP feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003," http://support.microsoft.com/kb/875352/EN-US/.

[29] P. Team, "Address space layout randomization, mar. 2003," http://pax.grsecurity.net/docs/aslr.txt.

[30] A. One, "Smashing the stack for fun and profit," *Phrack*, 1996.

[31] P. Larsen and A.-R. Sadeghi, *The Continuing Arms Race: Code-reuse Attacks and Defenses*, 2018.

[32] Hak5, "Wifi pineapple," https://www.wifipineapple.com/, 2018.

[33] "New IoT security rules: Stop using default passwords and allow software updates," https://tinyurl.com/yctkfuuj.

[34] "New IoT legislation bans shared default passwords," http://tinyurl.com/ybb9f6kp.

[35] "California passes law that bans default passwords in connected devices," https://tinyurl.com/ycr9bpof.

[36] A. Designer, "Internet of things security vulnerabilities: All about buffer overflow," https://tinyurl.com/ybfdaob3.

[37] "oFono," https://01.org/ofono.

[38] "BlueZ," http://www.bluez.org/.

[39] "Mer," http://www.merproject.org/.

[40] "yoctoproject," https://www.yoctoproject.org/.

[41] "Jolla OS," https://jolla.com/.

[42] "Configuring an IP address in the ostro os," Avaliable from Ostro Documentation. [Online]. Available: https://ostroproject.org/documentation/howtos/ip-address-config.html

[43] "Sailfish OS," https://sailfishos.org/.

[44] "tizen," https://www.tizen.org/.

[45] S. Saxena, "Tizen architecture," in *Tizen Developer Conference, San Francisco, California*, 2012.

[46] M. D. Sousa, *Internet of Things with Intel Galileo*, 2015.

[47] "CVE-2017-12865 detail," NATIONAL VULNERABILITY DATABASE, 2017. [Online]. Available: https://www.cvedetails.com/cve/CVE-2017-12865/

[48] "Dnsproxy: Fix crash on malformed DNS response," Avaliable from Connman git page, Aug. 09 2017. [Online]. Available: http://tinyurl.com/y6erhvg2

[49] "connman śrc/dnsproxy.c´ stack based buffer overflow vulnerability," Securityfocus, 2017. [Online]. Available: https://www.securityfocus.com/bid/100498

[50] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, Virginia, USA, 2007, pp. 552–561.

[51] S. Schirra, "Ropper," https://github.com/sashs/Ropper, 2018.

[52] Oracle, "Procedure linkage table (processor-specific)," http://tinyurl.com/y2zpweh5, 2018.

[53] "execlp(3): execute file," https://linux.die.net/man/3/execlp, 2018.

[54] J. Salwan, "Ropgadget tool," https://github.com/JonathanSalwan/ROPgadget, 2017.

[55] A. Ltd, "4.8.1. B, BL, BX, BLX, and BXJ," http://tinyurl.com/j7eo7vn, 2010.

[56] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017, pp. 259–284.

[57] A. Ltd, "Arm trustzone technology for armv8-m architecture. version 2.1," http://tinyurl.com/y3yzz2v8, 2017.

[58] S. Pinto and N. Santos, "Demystifying ARM TrustZone: A comprehensive survey," *ACM Computing Surveys CSUR*, vol. 51, no. 6, p. 130, 2019.

[59] P. Sparks, "The route to a trillion devices," *White Paper, ARM*, 2017.

[60] D. Kaplan, T. Woller, and J. Powell, "AMD memory encryption tutorial," *White Paper*, 2016.

[61] V. Costan and S. Devadas, "Intel SGX explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[62] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[63] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*, 2011, pp. 77–98.

[64] D. J. Malan, M. Welsh, and M. D. Smith, "A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography," in *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON 2004*, 2004, pp. 71–80.

[65] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit cpus," in *International workshop on cryptographic hardware and embedded systems*, 2004, pp. 119–132.

[66] S. Sultana, D. Midi, and E. Bertino, "Kinesis: a security incident response and prevention system for wireless sensor networks," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, 2014, pp. 148–162.

[67] R. Watro, D. Kong, S. fen Cuti, C. Gardiner, C. Lynn, and P. Kruus, "TinyPK: securing sensor networks with public key technology," in *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, 2004, pp. 59–64.

[68] S. Zhu, S. Setia, and S. Jajodia, "LEAP: Efficient security mechanisms for large-scale distributed sensor networks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington D.C., USA, 2003, pp. 62–72.

[69] A. L. M. Neto, A. L. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. Loureiro, D. F. Aranha, H. K. Patil *et al.*, "AoT: Authentication and access control for the entire IoT device life-cycle," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, 2016, pp. 1–15.

[70] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead," *Computer networks*, vol. 76, pp. 146–164, 2015.

[71] S. Gisdakis, T. Giannetsos, and P. Papadimitratos, "SHIELD: A data verification framework for participatory sensing systems," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015, p. 16.

[72] T. Markmann, T. C. Schmidt, and M. Wählisch, "Federated end-to-end authentication for the constrained internet of things using ibc and ecc," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 603–604.

[73] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker." in *Proceedings of the USENIX Annual Technical Conference*, 2012, pp. 309–318.

[74] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 144–157.

[75] F. A. Teixeira, F. M. Pereira, H.-C. Wong, J. M. Nogueira, and L. B. Oliveira, "SIoT: Securing internet of things through distributed systems analysis," *Future Generation Computer Systems*, 2017.

[76] N. G. Tsoutsos and M. Maniatakos, "Anatomy of memory corruption attacks and mitigations in embedded systems," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 95–98, 2018.

[77] "Behind the masq: Yet more DNS, and DHCP, vulnerabilities," http://tinyurl.com/y7l44lmw.

[78] "Dnsmasq," http://www.thekelleys.org.uk/dnsmasq/doc.html.

[79] "Google security research PoCs for dnsmasq," http://tinyurl.com/ycj23hm4.

[80] "Local privilege escalation exploit/PoC for dnsmasq <v2.78 on vyos," http://tinyurl.com/y2qujb2n.

[81] "Enterprise security for IoT," http://senr.io/.

[82] "400,000 publicly available IoT devices vulnerable to single flaw," https://tinyurl.com/ycb2p7q4.