Debugging the Linux Madwifi Driver

Tung M. Le[1], Lachlan L. H. Andrew and Hai L. Vu
Centre for Advanced Internet Architectures. Technical Report 100820B
Swinburne University of Technology
Melbourne, Australia
6216005@swin.edu.au

***Abstract*-This report describes a bug in the fragmentation code of the Madwifi driver for the Atheros chipset under the Linux kernel. The bug is due to incorrect locking of a data structure. Techniques for Linux kernel debugging are described, and a fix for the bug is presented.**

I. Introduction

Wireless local area networks (WLANs) are increasingly popular, leading to increasing demands on wireless resources. CAIA's MAGIC project is investigating modifications to the medium access control (MAC) protocol used by IEEE 802.11 (WiFi) WLANs [1], to improve its performance. A useful tool for this is the Atheros wireless chipset. Unlike most chipsets, the Atheros chipset implements most MAC functionality in software on the host computer rather than in firmware on the wireless card. This makes the chipset very flexible and suitable for experimentation.

This report describes the outcome of an internship by Tung Le, supervised by Hai Vu and Lachlan Andrew, which originally aimed to investigate variants of the ready-to-send/clear-to-send (RTS/CTS) mechanism [1] which WiFi uses to avoid combat "hidden terminals" [2]. This mechanism was to be emulated by using explicit packet fragmentation, which involves the host computer passing a large packet to the MAC and the MAC dividing the packet into smaller frames before transmission onto the wireless medium.

However, a bug was discovered in the implementation of fragmentation in the MadWiFi driver [3], and the project changed to fixing that bug. MadWiFi is one of three open source drivers for this chipset. The ath9k driver was written by Athero[1]s and the ath5k driver was, like MadWiFi, written by reverse engineering the chipset behavior. Although it would have been possible to continue the original project by using a different driver, it was decided that contributing a fix to the bug would be a useful contribution.

Although the MadWiFi driver is popular, the bug has not previously been detected because fragmentation is not common. The kernel normally produces packets no larger than 1500 bytes, and the WiFi interface allows packets of that size.

The outline of this paper is as follows. Section II describes the conditions under which the bug occurs, and its symptoms. Section III describes how it was found that the bug was in the driver rather than another component. Sections IV-VI present the debugging techniques used: remote access to the system console, detailed code inspection and the "bisection" method for locating bugs. The final cause of the bug and the solution are presented in Section VII.

II. Bug description

This section covers the detailed hardware, software description and the scenario where the bug occurs.

We consider an infrastructure wireless LAN with one Acer Aspire One net-book as a mobile device and an access point. The Madwifi driver has been installed on the net-book running Linux operating system. The following version of Madwifi and Ubuntu is used:

```
Linux version: 2.6.32-22-generic
Wireless Network Adapter: AR928X
Madwifi version: svn r4132 (trunk)
```

Tests are carried out by sending ping packets from the net-book to the access point. Without fragmentation, the transmission occurs as expected. When fragmentation option is turned on and the net-book sends packets exceed fragment threshold, the system crashes. Below is the screenshot showing command sequence and output when the crash occurs.

Figure 1: The network setup

```
root@wlan-proj-5:~# iwconfig ath0 frag 512
root@wlan-proj-5:~# iwconfig
lo        no wireless extensions.

wifi0     no wireless extensions.

ath0      IEEE 802.11g  ESSID:"AP605"
          Mode:Managed  Frequency:2.412 GHz  Access
Point: 00:0F:66:90:AD:0D
```

[1] Author is currently an engineering student at Swinburne University of Technology. This report was written during the author's winter internship at CAIA in 2010

```
        Bit Rate:11 Mb/s   Tx-Power:13 dBm
Sensitivity=1/1
        Retry:off   RTS thr:off   Fragment thr=512 B
        Encryption key:1111-1111-11   Security
mode:restricted
        Power Management:off
        Link Quality=33/70  Signal level=-60 dBm  Noise
level=-93 dBm
        Rx invalid nwid:69797  Rx invalid crypt:0  Rx
invalid frag:0
        Tx excessive retries:0  Invalid misc:0   Missed
beacon:0

eth5      no wireless extensions.

root@wlan-proj-5:~# ping 192.168.0.1 -c 1 -s 1000
*****system crashes here*********
```

Figure 2: Screen commands and output

As shown in Fig. 2, the fragmentation threshold has been set to 512 bytes and the net-book was sending 1000 bytes ping packet.

Following messages are what appears in screen before the machine crashes:

```
[ 4656.967047]  [<c03532bd>] ? copy_from_user+0x3d/0x130
[ 4656.967047]  [<c04bc8da>] ? verify_iovec+0x5a/0xa0
[ 4656.967047]  [<c04b41fd>] sys_sendmsg+0x15d/0x290
[ 4656.967047]  [<c058a479>] ? mutex_lock+0x19/0x40
[ 4656.967047]  [<c0353189>] ? copy_to_user+0x39/0x130
[ 4656.967047]  [<c03b87f5>] ? copy_termios+0x35/0x50
[ 4656.967047]  [<c01c9ca2>] ? find_get_page+0x22/0xa0
[ 4656.967047]  [<c01304cc>] ? kmap_atomic_prot+0x4c/0xf0
[ 4656.967047]  [<c01ca036>] ? unlock_page+0x46/0x50
[ 4656.967047]  [<c01e49b8>] ? __do_fault+0x3a8/0x490
[ 4656.967047]  [<c01e6259>] ? handle_mm_fault+0x139/0x390
[ 4656.967047]  [<c04b481c>] sys_socketcall+0xcc/0x280
[ 4656.967047]  [<c058db30>] ? do_page_fault+0x160/0x3a0
[ 4656.967047]  [<c01033ec>] syscall_call+0x7/0xb
[ 4656.967047] NOHZ: local_softirq_pending 2c2
```

Figure 3: Screenshot of crash

The following sections describe the debugging process to identify why the system crashes and how to fix it.

III. Bug Source Identification

The first step of the debugging process is to identify the place that causes the crash. There are several possible error sources such as hardware faulty or incompatibility, software error (bug) in either operating system or the Madwifi driver. Each possibility must be carefully verified because it significantly affects the later debugging steps.

The first possibility is because of hardware faulty. The test has been carried out in the same experiment and configurations but using different net-book. The same crash is observed. So the bug is not likely due to hardware failure.

We then replace the Ubuntu net-book with a Unix operating computer and repeat the test. Again the same crash is observed. So the bug is not likely due to operating system failure.

As the Madwifi driver has been developed through various versions, the easiest way to check is to reinstall the newest Madwifi driver in the net-book. Fragment bug might have been discovered before and is fixed in a newer version. However, after installing the new driver, the system still crashes.

The last possible error source is the implementation of the Madwifi driver. To verify it, the Madwifi driver is replaced by an ath5k driver which is a default Linux wireless driver. We then observe that the system is now working properly with fragmentation.

At this point, we conclude that the fragment bug comes from the Madwifi driver implementation. In the next section, we describe the network setup to gather more information in order to find the place in the Madwifi implementation that causes the crash.

IV. Collecting Information

Collecting information is essential in the debugging process. The more you know about the bug, the easier to fix. The most obvious source of information is the Internet, as chance is someone already experienced this bug and posted the solution for it. After some searching effort, no relevant information can be found that is related to this problem. Below we will focus on how to collect useful information regarding a so-called "oops messages".

Oops messages are error messages printed out when the system experiences problem. When Linux kernel detects problems, it prints out oops messages and terminates any offending process. Those messages are used to debug and fix software problems.

Fig. 3 shows messages that appear in the screen before the machine crashes. The problem is, however, a large part of these oops messages are missing. It is because the net-book crashes after printing all of those messages and we cannot go back and see do all the error messages except the last part appeared in the screen. One way to overcome this problem is to use a network console. Network console has been introduced from Linux 2.6 series. It allows sending oops messages to an external terminal using UDP. So the entire oops messages can be retrieved in the remote machine later on. To setup the network console, a separate Ethernet interface is created between the mobile device and a desktop computer.

Fig. 4 shows the network console and the commands to set it up with the net-book.

Eth0: the interface used to send oops messages

12345: port number to send oops messages

10.0.0.1: remote host's IP address

00:E0:81:2B:0C:C1: MAC address of the remote host

The following important part of the oops messages is captured by the above network console:

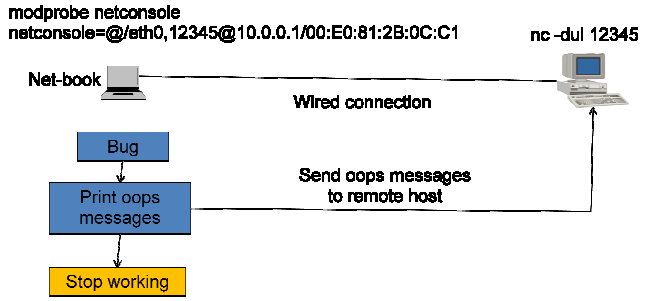

Figure 4: Network console

```
root@wlan-proj-6:~# nc -dul 12345
[ 4656.358987] ------------[ cut here ]------------
[4656.359188]kernel BUG at /home/tung/madwifi/ath/if_ath.c:3345!
[ 4656.359438] invalid opcode: 0000 [#1] SMP
[4656.359649] last sysfs file: /sys/devices/virtual/sound/timer/uevent
```

Figure 5: Network console captured oops messgaes

Fig. 5 clearly identifies where the crash occurs. Note that although in this case the oops messages was able to point the line of the code where the error lies, in general the oops messages do not always state exactly that. In the following sections V and VI, we assume that we have no knowledge about these oops message and will attempt to locate the bug in a different way.

V. SOURCE CODE

In order to fix something, it is important to understanding how it works. Since the bug causes system crashes whenever the net-book sends fragmented packets, we focus on the code to fragment and transmit packets.

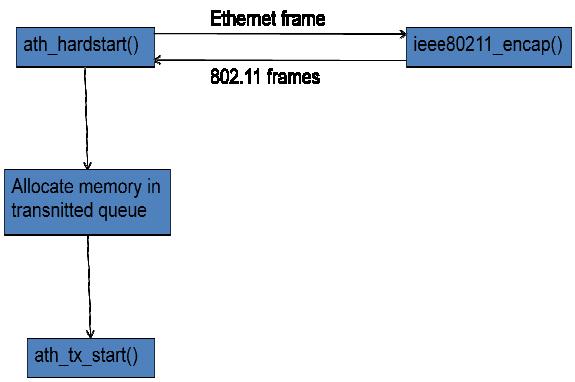

Figure 6: Madwifi transmission process

Figure 3 represents the simplified Madwifi transmission process. The function ath_hardstart() is called whenever kernel wants to send a packet. Then function ath_hardstart() calls function ieee80211_encap() to set the correct 802.11 encapsulation and fragment packet if necessary. After that, ath_hardstart allocates memory for each fragment. Finally, each fragment is sent by calling function ath_xt start().

Experiments to find the bug focused on those sections of code.

VI. DEBUGGING TECHNIQUES

Simply looking at the code rarely shows the cause of a bug. It is often necessary to modify the code and run it to obtain information about its dynamic behavior. This section describes the two techniques used to find the bug.

When debugging user-space programs, the simplest and most useful technique is to add diagnostic output. This technique inserts a of lots of output statements to track the control flow and data values in the execution of a piece of code. In the kernel, the function printk() behaves similarly to the printf() function in the standard C library. However, the differences make it less useful for debugging. The printk() function sends messages to a logging system, which saves the output in a file and can also send it to the console, depending on the settings of the system logger daemon. After the printk() calls are added in different positions of the transmission code, the Madwifi driver can be reinstalled, and the system made to crash. The hope is that the output message on screen or in the log files will indicate the last executed printk() before the crash. It would then be possible to narrow down the error source.

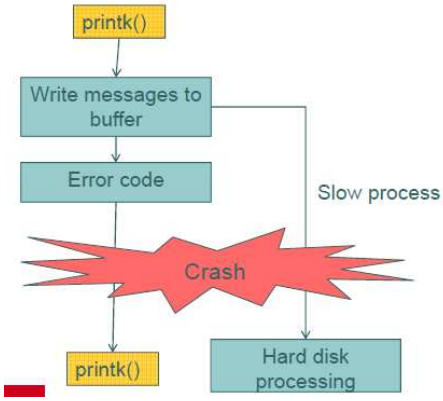

Figure 7: Weakness of printk()

However, when the system crashes, the printk() functions print nothing. The reason is that printk()7does not save messages directly to the hard disk. It just writes messages to a circular buffer then the next instruction is executed. It takes long time for Linux to copy messages from the buffer to the hard disk. In our case, the system crashes before the printk() messages are written to the hard disk, as illustrated in Figure 7. This means that printk() is of limited use in this case.

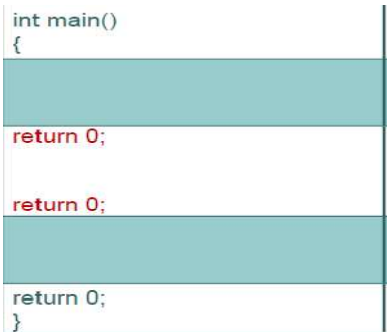

Figure 8: Bisection technique

An alternative technique that does not use printk() is bisection. In this technique, part of the code is disabled, such as by putting a return statement before the end of a function. If the system still crashes, the bug is probably within the section of code which was not disabled; otherwise the bug is most likely in the disabled code. This is repeated with increasingly small sections of code disabled until the location of the crash, or source of the bug, is found. This does not rely on printk()'s logging foibles, but it requires many iterations, in which the code is recompiled, the module reinstalled and (if there is a crash) the system is rebooted. However, this was the method which eventually showed the location of the bug (before the networked console technique was discovered).

VII. The Bug and its Fix

The location of the crash can be found from the oops message in section IV or by the technique described in section VI. The crashed occurred in the highlighted line in the following piece of code from the function ieee80211_encap() in file ath/if_ath.c:

```
for (bfcnt = 1; bfcnt < framecnt; ++bfcnt)
{
tbf = ath_take_txbuf_locked(sc); //crash
if (tbf == NULL)
break;
STAILQ_INSERT_TAIL(&bf_head, tbf,
bf_list);
}
```

This piece of code allocates memory for each fragment. The crash was caused by defensive programming by the module's authors: There is an explicit check for the presence of a "lock". A software lock is a mechanism to prevent conflicting accesses by multiple threads to a single data structure. Before writing to a data structure, a thread obtains a "lock" which it releases after it has restored the structure to a consistent state. Other threads are not able to obtain the lock until the initial thread releases it, which causes them to wait until the initial thread has finished. When a packet is fragmented, the lock should be obtained before extracting the first fragment, and released after the last has been extracted. For this reason, the function ath_take_txbuf_locked(sc) assumes that the buffer which contains packet to be transmitted has already been locked, and explicitly checks for this. However, the commands to obtain the lock were missing. That is why the system crashes when fragmented packets are sent. In order to fix this, the lock and unlock macros should be added, as follows:

```
ATH_TXBUF_LOCK_IRQ(sc);
for(bfcnt = 1; bfcnt < framecnt; ++bfcnt) {
      tbf = ath_take_txbuf_locked(sc);
      if (tbf == NULL)
            break;
      STAILQ_INSERT_TAIL(&bf_head,    tbf,
bf_list);
}

  if (bfcnt != framecnt) {
      ath_return_txbuf_list_locked(sc,
&bf_head);
      STAILQ_INIT(&bf_head);
      ATH_TXBUF_UNLOCK_IRQ(sc);
      goto hardstart_fail;
  }
  ATH_TXBUF_UNLOCK_IRQ(sc);
```

Note that the lock must be released after the failure as well as the successful transmission.

VIII. Conclusion

This report has described the process by which a bug was detected, located and removed from the MadWiFi driver for the Atheros chipset. The symptom of the bug was that the kernel crashed, and the final caused was due to missing locking of a data structure in the function ieee80211_encap() in file ath/if_ath.c.

The general debugging process involved the following steps:

- Bug source identification: identify source of the problem. It could be hardware or software faulty.
- Collecting information: Linux oops messages usually tell exactly where the bug is
- Understanding source code: know which part of the code does what job. So we have clue which part potentially causes error
- Debugging: two most simple methods are diagnostic output and bisection .

Acknowledgments

The authors thank Grenville Armitage for providing TL with the opportunity to undertake this project and CAIA for hospitality throughout the internship.

References

[1] IEEE 802.11-2007 IEEE Standard for Information technology -- Telecommunications and information exchange between systems -- Local and metropolitan area networks-Specific requirements -- Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications

[2] F. Tobagi and L. Kleinrock, "Packet Switching in Radio Channels: Part II--The Hidden Terminal Problem in Carrier Sense Multiple-Access and the Busy-Tone Solution", *IEEE Trans. Commun.*, 23(12):1417-1433, 1975.

[3] "Madwifi project", http://madwifi-project.org/ accessed July 2010.