UNIVERSITÄT BONN

Automated Recognition of the Vulnerability in "NetUSB"

Modeling Vulnerability as Classification Problem

Lab Report

by

Rabie Alawad
3342263

submitted to
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

in degree course
Informatik (B.Sc.) & Informatik (M.Sc.)

First Supervisor:	Prof. Dr. Michael Meier University of Bonn
Second Supervisor:	Dr. Matthias Frank University of Bonn
Sponsor:	Jörg Stücker University of Bonn

Bonn, April 21, 2022

Acknowledgement

Contents

1 Main

1.1 Intruduction

NetUSB is a kernel module and was developed by Kcodes which is a“ Corporation was founded in June 2001, specialized in USB over IP software development..........” .And it was licenced to many prominent Companies in Router Production like TP-Link, NetGear, and DLink.It allows a remote device in the network to connect to a usb device pluged in the Router as if it's pluged in the remote device. lately a vulnerability was discovered in this kernel module that could enable a buffer overflow attack . A month after that Kcodes released a patch to fix the vulnerability . the purpose of this paper is to present a suggested approach to automate recognizing whether a NetUSB kernel module is patched . The assessment is based on relatively small sample size of ten NetUSB kernel modules which were first decompiled then a Bit Patterns had to be found that marks the vulnerable Kernel modules and another that marks the patched kernel modules.Yara rules were used to express the derived bit pattern and were Incorporated into a Fact plugin.

1.2 The vulnerbility

the vulnerability is a result of a very common poor coding practice of not carefully managing memory in C and C++ especially in critical memory sections in the kernel .more specifically not checking whether the supposed size of the received or to be received data and the a actual size In this case an attacker could rewrite the function stack to execute a code in the kernel. In order to understand the vulnerability the function SoftwareBus_fillBuf needs to be introduced. it takes a Buffer and its size as input and writes data in that buffer from an external source The command reaches the following code in the function SoftwareBus_dispatchNormalEPMsgOut:

```
//4 bytes are received and stored in the buffer supplied_size
Var17 = SoftwareBus_fillBuf(sbus_info, (char*)&supplied_size
    ,4);

f ((int)uVar17 == 0) {
eturn;

 //the buffer is treated as an Integer and added to 0x11
    assuming a malicious intent Supplied_size would be equal
    to 0xffffffff which result in allocating the overflown
    value which is equal to 0x10*/
llocated_region = (char *) kmalloc(supplied_size + 0x11
                                                      ,0xd0);
f ((soft_bus_info *)allocated_region ==
                              (soft_bus_info *)0x0) {
og_msg = "INFO%04X: Out of memory in USBSoftwareBus";
og_line = (char *)0x1156; goto LAB_0001a8fc;
```

Listing 1.1: *the code section where the vulnerability is located.*

The value of supplied_size is add to 0x11 and then used as a size of a buffer to be allocated in the kernel by the function kmalloc. Since this supplied size isn't validated, the addition of the 0x11 can result in an integer overflow. For example, a size of 0xffffffff would result in 0x10 after 0x11 has been added to it. after that the supplied_size is used as the size of the buffer not the overflown value:

```
// assuming a malicious intent a code would be sent to be
    executed in the kernel
 uVar14 = SoftwareBus_fillBuf(subs_info,allocated_region->
    data, supplied_size)
```

Listing 1.2: *Out of bound writes*

the vulnerability was solved in various ways
one of the solutions is as following:

```
if(supplied_size  < 0x1000000) {
allocated_region = (char *) kmalloc(supplied_size + 0x11
                                                       ,0xd0);
}
```

Listing 1.3: *Patch 1.*

another solution :

```
supplied_size = ((( supplied_size >> 0x10) << 0x18 | (
    supplied_size >> 0x18) << 0x10) >> 0x10) + ((
    supplied_size \& 0xff) << 8 | supplied_size >> 8 \& 0xff)
     * 0x10000;
```

Listing 1.4: *Patch 2*

1.3 Task and Tools

The task is to create automated scanning tool that detects the above described vulnerability based on the following samples :

- Netgear R6300v2 - 1.0.4.6
- Netgear R6100 - 1.0.1.10
- Netgear PR2000 - 1.0.0.15
- Netgear R6220 - 1.1.0.34
- Netgear R7000 Nighthawk - 1.0.7.6_1.1.99
- TP-Link Archer C2600 - 160902
- TP-Link Archer C1200 - 160918
- TP-Link Archer C59 - 160621
- TP-Link TL-WR1043NDv4 - 160607
- zyxel NBG4615 v2 - V1.00(AAFI.3)C0

before starting with the task the following tools used in this project need to be introduced :

- Ghidra. : Reverse Engineering tool that provides various reverse engineering functionalities among which is decompiling and providing a Symbol tree of the decompiled code.
- Yara Rules: used to express bit pattern that are used to detect malware that regularly shows the expressed bit pattern.
- Fact Core: Firmware analysis tool that can take a yara Plugin that can be used to detect malware and vulnerabilities via Fact core.

to achieve that Ghidra was used to decompile the binary files then bit patterns had to be found that mark the vulnerability and expressed via Yara Rules . in the end yara rule was incorporated in a Fact core Plugin ,which enables Fact core to detect these Bit patterns based on the Yara rules that were designed.

1.4 The Approach

The first step of the solution is to decompile the samples using Ghidra to the high-level language from which they were compiled. Ghidra provides multipule additional functionalities to decompiling , most important of all for this assessment is the symbol tree, which contains symbols of imports ,functions,labels .....and so on. under the "function tree" in tree symbol section the label of the function"SoftwareBus_dispatchNormalEPMsgOut" can be found where the vulnerability is located. The decompiled code of this function and the corresponding machine code can be accessed by clicking on that label.

its possible to to know which samples are patched and which are not by inspecting the code section where the vulnerable implementation of kernel memory allocation is and seeing if the vulnerability was fixed.

the following samples were not patched since there is no checking if the supplied_size can be overflown:

- Netgear R6300v2 - 1.0.4.6
- Netgear PR2000 - 1.0.0.15
- Netgear R6220 - 1.1.0.34
- Netgear R7000 Nighthawk - 1.0.7.6_1.1.99
- TP-Link Archer C2600 - 160902
- TP-Link Archer C1200 - 160918
- zyxel NBG4615 v2 - V1.00(AAFI.3)C0

And the folowing are Patched :

- TP-Link Archer C59 - 160621
- TP-Link TL-WR1043NDv4 - 160607
- Netgear R6100 - 1.0.1.10

The vulnerability in the presented samples is fixed by the following line code which insures that the variable does not take a value that can potentially be overflown by adding it to 0x11 :

```
local\_48[0] = (((local\_48[0] >> 0x10) << 0x18 | (local\_48
    [0] >> 0x18) << 0x10) >> 0x10) + ((local\_48[0] \& 0xff)
    << 8 | local\_48[0] >> 8 \& 0xff) * 0x10000;\\
```

Listing 1.5: *Patch fix*

this line of code corresponds to a unique binary bit-String. Which means a static Yara rule can classify a kernel module as being vulnerable or not only by looking for this bit string given the binary file being scanned by this rule is a NetUSB kernel module . Other patches have implemented other lines of codes to fix the vulnerability that means the Yara rule can still be extended by adding the bit strings that correspond to these lines of codes in order to detect other variants of the Patch, but no kernel with a different patch was found among the considered samples .

```
rule NetUSB
  {
   strings:
       $a = { 7c 04 20 a0 00 24 24 02}
   condition:
       $a
  }
```

Listing 1.6: *Negative Implementation*

but this Yara rule is not suitable for general use , because most software recognition tools don't only look for one vulnerability in one single product and it's not assumed that the input is a NetUSB kernel module ,in this scenario kernel modules with vulnerability would not be detected but only fixed ones. A Yara rule for general use have to detect the vulnerability itself not the vulnerability fix.

it was clear based on the samples that there is no single bit pattern in the code section where the vulnerability is located that marks the class of kernel modules with the vulnerability but dividing the samples into sub classes made it possible to find tow bit patterns that mark tow classes of vulnerable kernel module. samples of a subclass show the same machine code instructions in the code section where vulnerability is located. the first subclass includes:

- Netgear R6300v2 - 1.0.4.6
- Netgear R7000 Nighthawk - 1.0.7.6_1.1.99
- TP-Link Archer C2600 - 160902
- TP-Link Archer C1200 - 160918

the first subclass includes:

- Netgear PR2000 - 1.0.0.15
- Netgear R6220 - 1.1.0.34

all samples of the first[] class show identical machine code at the vulnerable section as the following code demonstrates:

```
\\the double question mark denotes that the byte can vary
    between the samples of the first class
        00 00 50 e3     cmp         r0,#0x0
        ?? ?? ?? 0a     beq         LAB_0001ac44
        ?? ?? ?? e5     ldr         r0,[sp,#local_2c]
        d0 10 a0 e3     mov         r1,#0xd0
        11 00 80 e2     add         r0,r0,#0x11
        ?? ?? 00 eb     bl          __kmalloc
```

Listing 1.7: *vulnerable section of the first class*

and the second class:

```
\\the double question mark denotes that the byte can vary
    between the samples of the first class
        ?? ?? a4 8f     _lw         a0,local_48(sp)
        04 00 02 3c     lui         v0,0x4
        d0 00 05 24     li          a1,0xd0
        ?? ?? ?? 24     addiu       v0,v0,-0x2e4c
        09 f8 40 00     jalr        v0=>__kmalloc
```

Listing 1.8: *vulnerable section of the second class*

A yara rule can be derived from the tow machine codes above that can detect the vulnerability itself :

```
rule Positive
{
    meta:
        description = "This is just an example"
        threat_level = 3
        in_the_wild = true
     strings:
        $a = { e5 d0 10 a0 e3 11 00 80 e2  }
        $b = {09 f8 40 00}

    condition:
        $a or $b
}
```

Listing 1.9: *Yara rule to detects the vulnerability*

these yara rules can be incorporated into FACT_core as a plugin . the prosses of creating a FACT_core plugin is straight forward as documented.

the last sample 'zyxel NBG4615 v2 - V1.00(AAFI.3)C0' could not be attributed to any of the tow classes , so it's assumed that it belongs to another sub class of which enough samples were not included in this assessment .

1.5 Results

The method used to derive the solution makes assumptions that must be fulfilled otherwise the solution would not be practical. the most important assumption one is that a class would be divided into a reasonable number of subclasses that do not lead to an over-subclassification, the make the solution unusable, since all 3 to 6 kernel modules a new subclass. this is called overfitting, which is well studied in the field classification. concepts in machine learning were applied throughout this approach or a similar approach. By excluding zyxel NBG4615 v2 - V1.00(AAFI.3) the 2 classifiers do not have any false positive or false negative when used on the available samples in this report.

2 The Conclusion

The possibility of framing vulnerability detection as a classification problem has been investigated throughout this paper by going over an concrete implementation example. it's easy to see the papers' shortcoming due to the small sample size investigated . Dividing the class of the vulnerable software seemed practical on a smaller scale ,but the approach might lead to over-classification or so called overfitting, so it might not be practical in some use cases. further studies investigating this method with sufficiently larger sample size are required to test its efficiency

Listings

Statement of Authorship

I hereby confirm that the work presented in this thesis has been performed and interpreted solely by myself except where explicitly identified to the contrary. I declare that I have used no other sources and aids other than those indicated. This work has not been submitted elsewhere in any other form for the fulfilment of any other degree or qualification.

RABIE

Bonn, April 21, 2022

Rabie Alawad