

Sri Lanka Institute of Information Technology

**CVE-2017-16995 Linux Kernel - BPF Sign Extension Local Privilege Escalation**

Individual Assignment

[IE2012](https://courseweb.sliit.lk/enrol/self/unenrolself.php?enrolid=10332) - [Systems and Network Programming](https://courseweb.sliit.lk/course/view.php?id=3166)

Submitted by:

|  |  |
| --- | --- |
| Student Registration Number | Student Name |
| IT19187488 | Y.K.Lumindu Dilumka |

Date of submission

2020/05/12

**Table of Contents**

1.How I found the vulnerability…………………………………………….03

2. Background………………………………………………………………04

3. Vulnerability Overview and Impact……………………………………...04

4. Code Analysis……………………………………………………………06

4.1 eBPF Instruction Set……………………………………………..06

4.2 Source Code Analysis……………………………………………08

4.3 Explanation for the Exploit………………………………………12

5.Conclusion……………………………………………………...…………15

6. References………………………………………………………..………16

**1.How I found the vulnerability**

I found this vulnerability for the assignment by google and this  vulnerability, numbered [CVE-2017-16995](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995), is related to a [set](http://www.openwall.com/lists/oss-security/2017/12/21/2) of security vulnerabilities in the eBPF verifier found by [Jann Horn](https://twitter.com/tehjh) that were fixed in this [commit](https://github.com/torvalds/linux/commit/3db9128fcf02dcaafa3860a69a8a55d5529b6e30). It is operational on Linux kernel versions 4.4 to 4.14.

**2. Background**

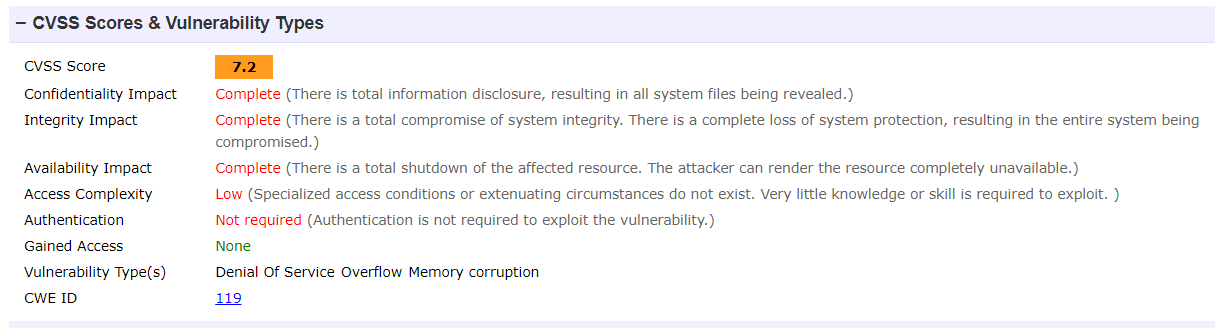
eBPF(Enhanced Berkeley Packet Filter) is a virtual in-kernel machine that is used as an interface to data link layers, enabling the filtering of network packets by rules. A userspace process provides a filter program to specify which packets it wishes to receive, and eBPF returns packets that pass through its filtering.

Every BPF memory instruction consists of 64 bits (eight bytes). 8 opcode bits, 4 source register bits, 4 target register bits, 16 offset bits and 32 bits for the immediate interest.

eBPF is made up of 10 64-bit registers referred as r0-r10. R0 stores the return value, r1 to r5 is allocated for arguments, r6 to r9 is allocated for callee saved registers and r10 stores read-only frame pointers.

In order to preserve the status between eBPF function invocations, to enable data sharing between eBPF kernel programs and even between kernel and user-space applications, eBPF uses various types of maps in the form of a key-value pair. To facilitate the data sharing between programs, two bpf functions, BPF\_MAP LOOKUP\_ELEM and BPF\_MAP\_UPDATE\_ELEM are provided.

**3.Vulnerability Overview and Impact**



The bug is triggered by an expansion of the sign from a signed 32-bit integer to an unsigned 64-bit integer, bypassing the eBPF verifier and contributing to a local privilege escalation.

Before each of the BPF programs runs, two verification passes are carried out to ensure that it is correct. The first move check\_cfg() guarantees that the code is loop-free using depth-first scan. The second pass do\_check() will run a static review to simulate the execution of all potential paths obtained from the first instruction. The software would be stopped if some incorrect instruction or memory infringement has been detected.

In the exploit, a series of BPF instructions was deliberately designed to circumvent this filtering mechanism by unintended sign extension from 32 bits to 64 bits. As a consequence, a few lines of malicious code added managed to run in kernel space, culminating in a privilege escalation.

The eBPF opcodes can be referenced from

<https://github.com/iovisor/bpf-docs/blob/master/eBPF.md>

This vulnerability enables the attacker to have complete control of the root access system. The low severity of the assault and the limited rights needed to execute this procedure make it a high priority to repair it.

**3. Code Analysis**

**3.1 eBPF Instruction Set**

User-provided eBPF programs are written in a special machine language that operates on a computer eBPF system. The VM follows the design of the Reduced Instruction Set Computer(RISC) and has 10 general purpose registers and several named registers..

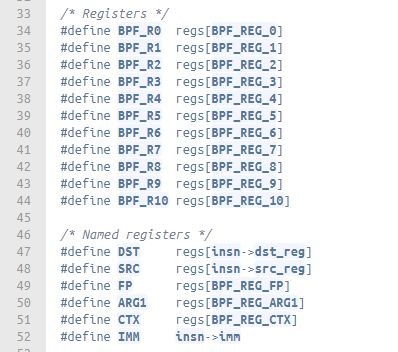
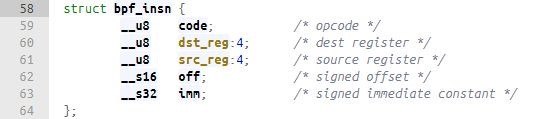
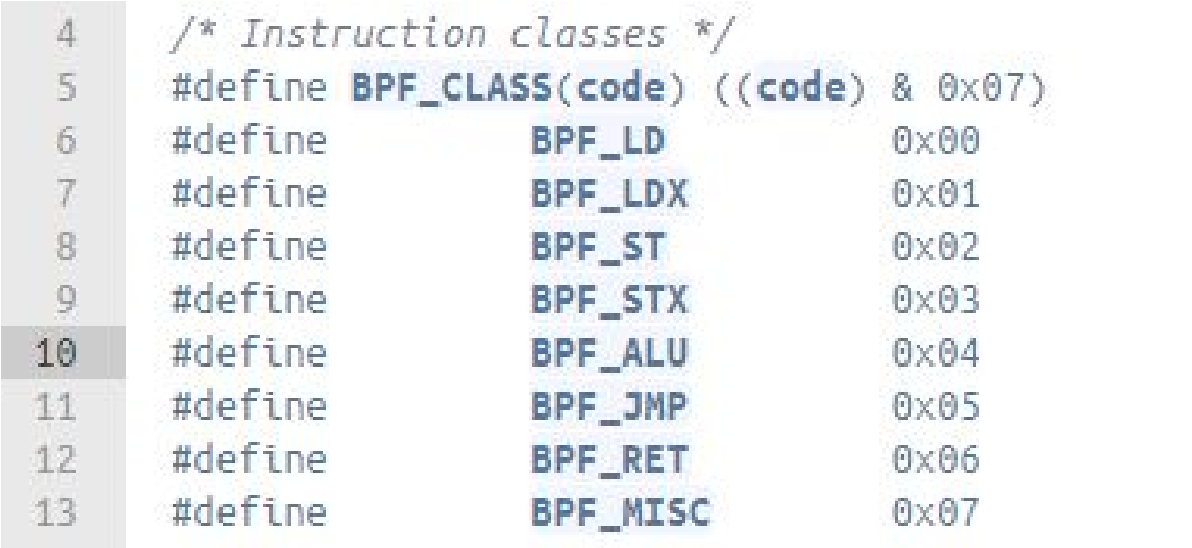


fig 3.1 Register Definitions in the eBPF VM, from /kernel/bpf/core.c 2

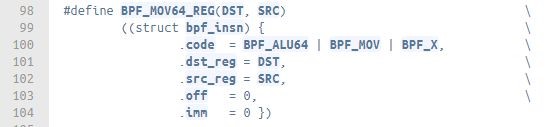
Each BPF instruction on the x64 platform is 64 bit long. They are internally defined by the bpf insn struct comprising the following sectors (fig 3.2). According to the small size of the opcode sector, the instructions are grouped into 8 groups (fig 3.3). For example, BPF MOV shares the same opcode with BPF ALU64 and BPF X by definition (Fig 3.4).



*fig 3.2 Structure of a BPF instruction, from /include/uapi/linux/bpf.h*



*fig 3.3 BPF instruction classes, from /include/uapi/linux/bpf\_common.h*



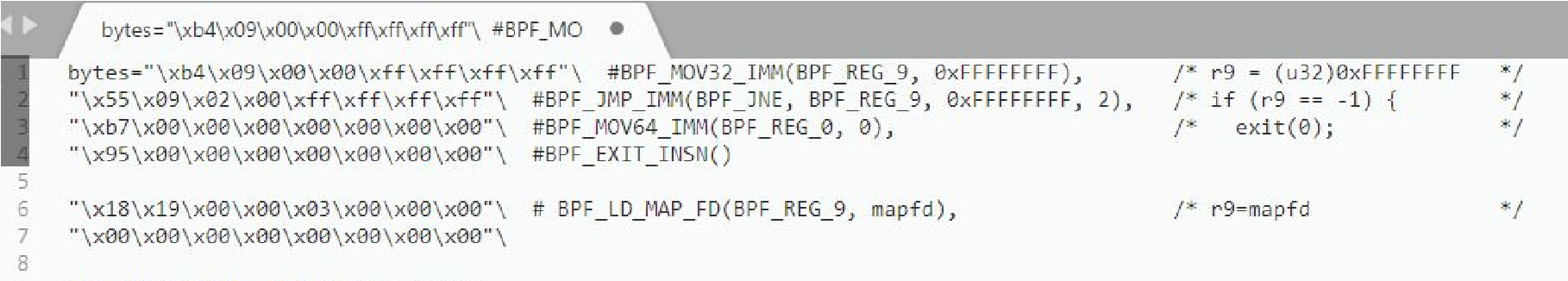
*fig 3.4 Definition of BPF\_MOV64\_REG, from /include/linux/filter.h*

Source code excerpts in this document are based on kernel version v4.4.116.

## 

## 3.2Source Code Analysis

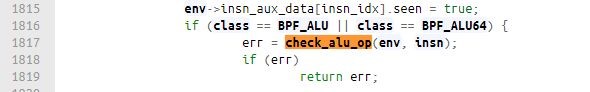
The usage of CVE-2017-16995 is restricted to a mere 40 eBPF instructions. We should concentrate on the first two instructions as they are primarily used to circumvent the eBPF verification process.



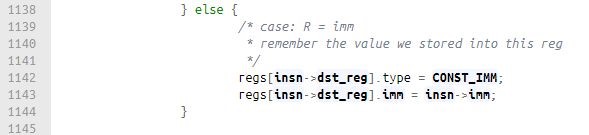
*fig 3.5 eBPF code in the exploit of CVE-2017-16995 with annotation*

As described above, eBPF conducts two round verifications before finally running the user-supplied code. For this CVE we are only interested in the second round check that is done in the do \_check() function. When the first BEF MOV32 IMM instruction is executed, it is transferred to check alu \_op() to process as BEF MOV32 IMM belongs to the BPF ALU community (Fig. 3.6). The immediate value (0xFFFFFFFF) of the first instruction is then placed in the BPF REG 9 register (Fig. 3.7).



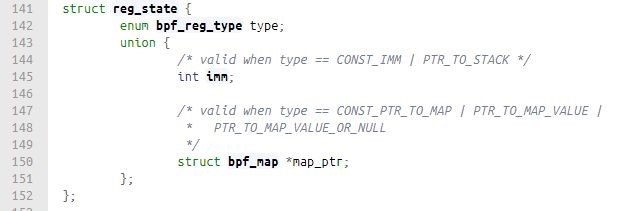


*fig 3.6 do\_check(), from /kernel/bpf/verifier.c*



*fig 3.7 check\_alu\_op() from /kernel/bpf/verifier.c*

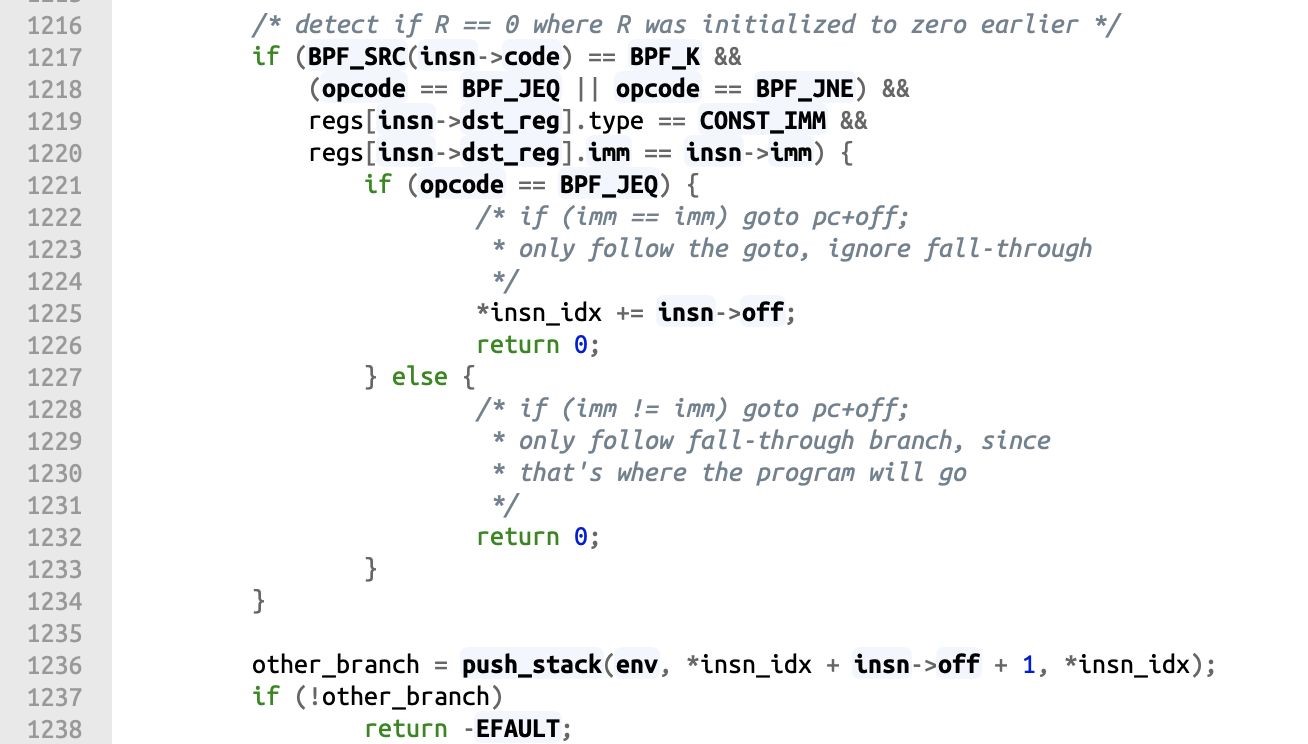
To make it simpler, we should have a peek at how the eBPF registers are represented. The registered ones are contained in a struct list called reg state. The immediate value 0xFFFFFFFF is encoded in a 64-bit int imm, which is 0x00000FFFFFFFFFF in memory.



*fig 3.8 struct reg\_state, from /kernel/bpf/verifier.c*

The second instruction BPF JMP IMM(BPF JNE, BPF REG 9,0xFFFFFFFF, 2) is now evaluated. The order compares the immediate value 0xFFFFFFFF with the content within BPF REG 9, and hops to the position that is 2 instructions away if the two values are not identical.

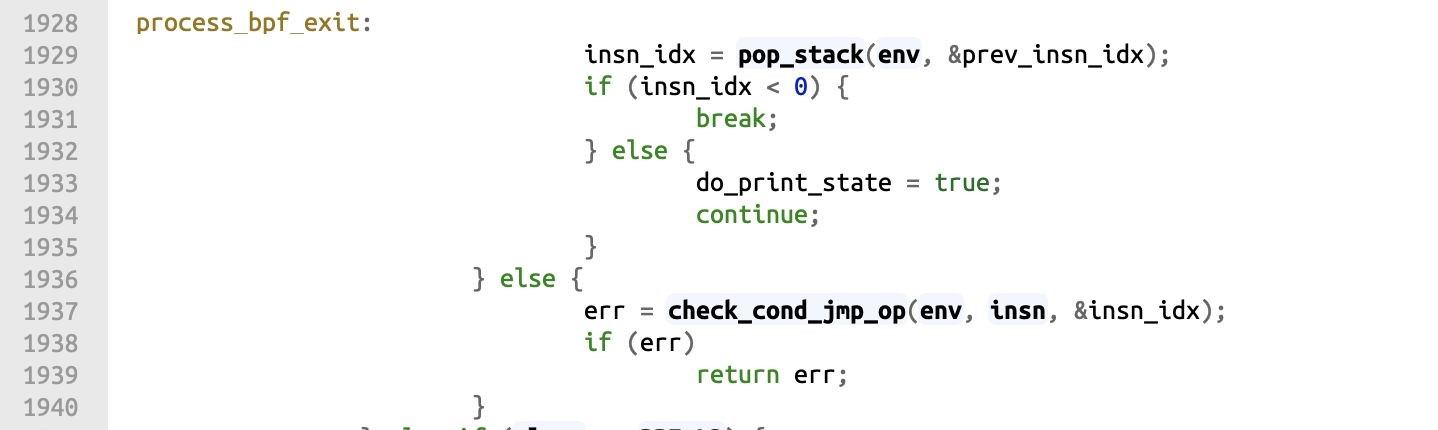
This time do check() calls search cond jmp op() to test both form and meaning in dst reg, which is BPF REG 9 in this case (fig. 3.9). Definitely (int)0x00000FFFFFFFF = (s32)0xFFFFFFFF and opcode! = JEQ, comes under the category of imm! = imm and the leap is not made. The software must proceed until it reaches BPF EXIT\_INST() on line 4 and exits.

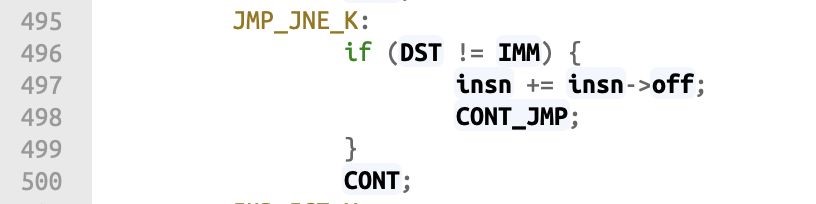
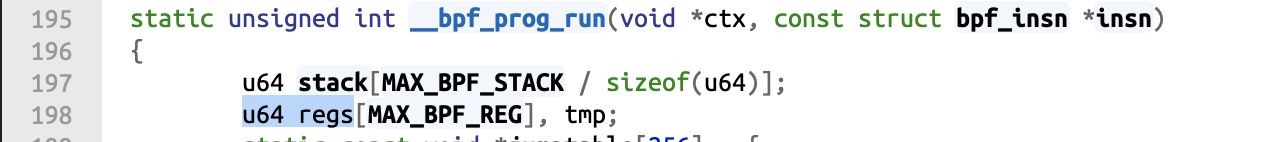


*fig 3.9 check\_cond\_jmp\_op(), from /kernel/bpf/verifier.c*

Currently, the eBPF verifier uses a stack to maintain track of divisions that have not been checked and to update them later (Fig. 3.9, line 1236). However, as the integer relation on line 1220 is still the same, the code starts from line 1232 and the other branch is never added to the stack.

When the verifier checks BPF EXIT, it tries to pick all unresolved branches out of the stack (fig 3.10). The testing cycle must end here, because it knows the stack is zero. As a consequence, only the first 4 instructions in the code are tested while the other 36 are unchecked.

*Fig 3.10 Evaluation of instruction BPF\_EXIT, from /kernel/bpf/verifier.c*



*Fig 3.11 regs definition and \_\_bpf\_prog\_run(), from /kernel/bpf/core.c*

After checking, eBPF runs the software via bpf prog run() in core.c, where eBPF instructions are converted into machine instructions using a jump chart. Note that the regs type here is u64. Using the same first two instructions in exploit.c, a sign extension happens while we test the first BPF MOV32 IMM command. More exactly, this occurs when we run DST = (u32)IMM in line 350:

● IMM on the right is identical to insn->imm. Imm is a signed 32-bit integer defined by bpf insn(fig 3.2). Here is IMM = 0xFFFFFF. We set it on the unsigned 32-bit integer, which is always 0xFFFFFFFF.

● On the left side, DST is defined as regs[insn->dst reg], an unsigned 64-bit integer. When we let DST = (u32) IMM, the sign extension is extended and the DST is 0xFFFFFFFFFFFFFF.

Now, if we test the second instruction JMP JNE K, DST would not be equivalent to IMM as 0xFFFFFFFFFFFF! = 0xFFFFFFFFFF. This is distinct from what we saw in the verifier. As a result, the hop is made and the machine proceeds to operate the harmful instructions from line 5.

## 3.3Explanation for the Exploit

## 

## Fig 3.13 Data flow in the exploit

As described above, eBPF uses shared memory for the kernel system to interact with user applications. If we think of this closely, this might be a possible way for us to transfer commands to the kernel and to sneak details to the outside of the kernel. We can soon see how the exploit uses this eBPF map to complete an arbitrary read / write kernel in a short time.

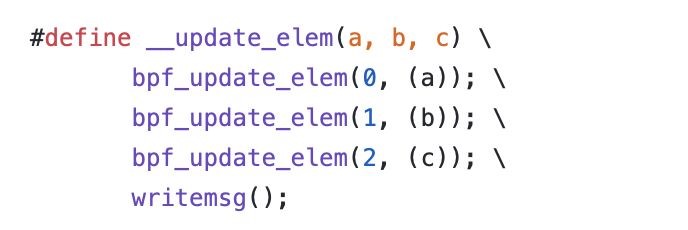
Simply stated, the exploit consists of two parts: an eBPF filter system running in the kernel and a helper system running in the user's space. The assault can be extended to the following steps:

● exploit.c generates an eBPF chart of size 3 using bpf creat map()and loads the eBPF command char\*prog into the kernel using bpf prog load().

● The eBPF instructions act as an agent that takes commands from the chart and reads / writes the kernel space appropriately. The map layout is described as follows:

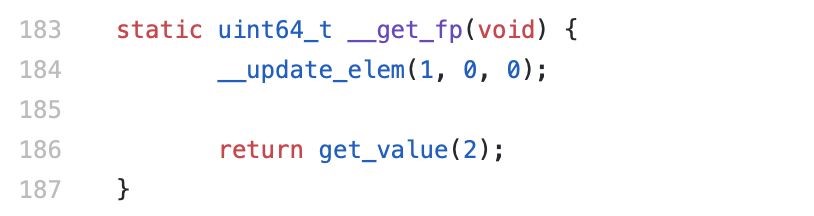
|  |  |  |  |
| --- | --- | --- | --- |
| **Index of eBPF map** | **To read from kernel** | **To get the current frame pointer** | **To write to kernel** |
| 0 (opcode) | 0 | 1 | 2 |
| 1 (address) | Target address | 0 | Target address |
| 2 (value) | (Content at the address) | 0 | 0 |

● To launch a read / write process, exploit.c must first store the parameters in the map using bpf update elem(). This would then call writemsg() that sends a few dummy packets to the socket and cause the eBPF program to run.



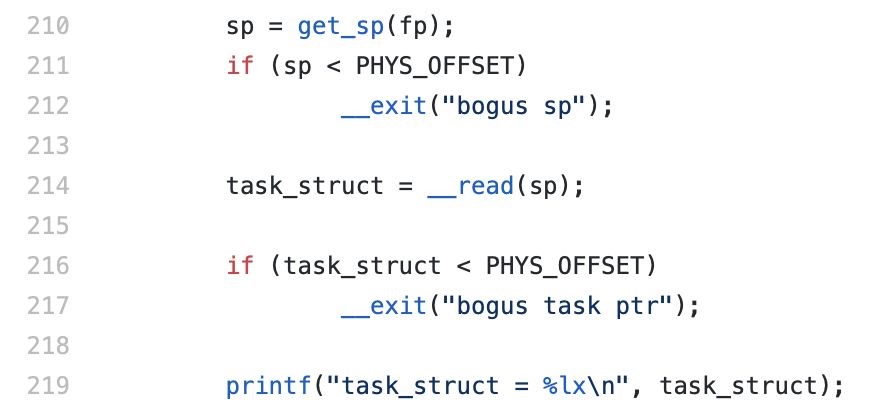
*Fig 3.14 \_\_update\_elem() from exploit.c*

● Given the aid resources above, we can now get the address of the current frame pointer by instructing the BPF software to execute opcode 1. The return value is located in index 2 on the diagram.

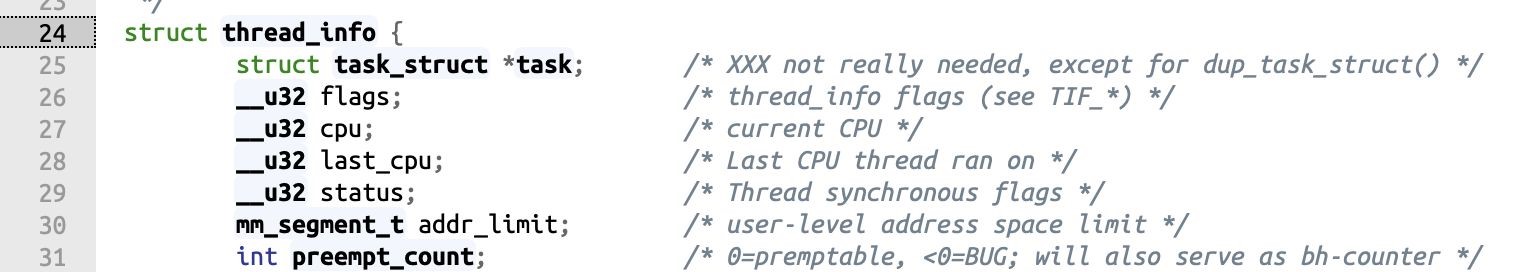


*Fig 3.15 \_\_get\_fp() from exploit.c*

* After the frame pointer has been retrieved, it can be used to locate the task struct pointer in the kernel stack (fig 3.16), which is within the thread info struct. Because the stack size is 8 KB, masking the least significant 13 bits will send the thread info key. The value read from the thread info address will therefore be the address of the task struct \* task.

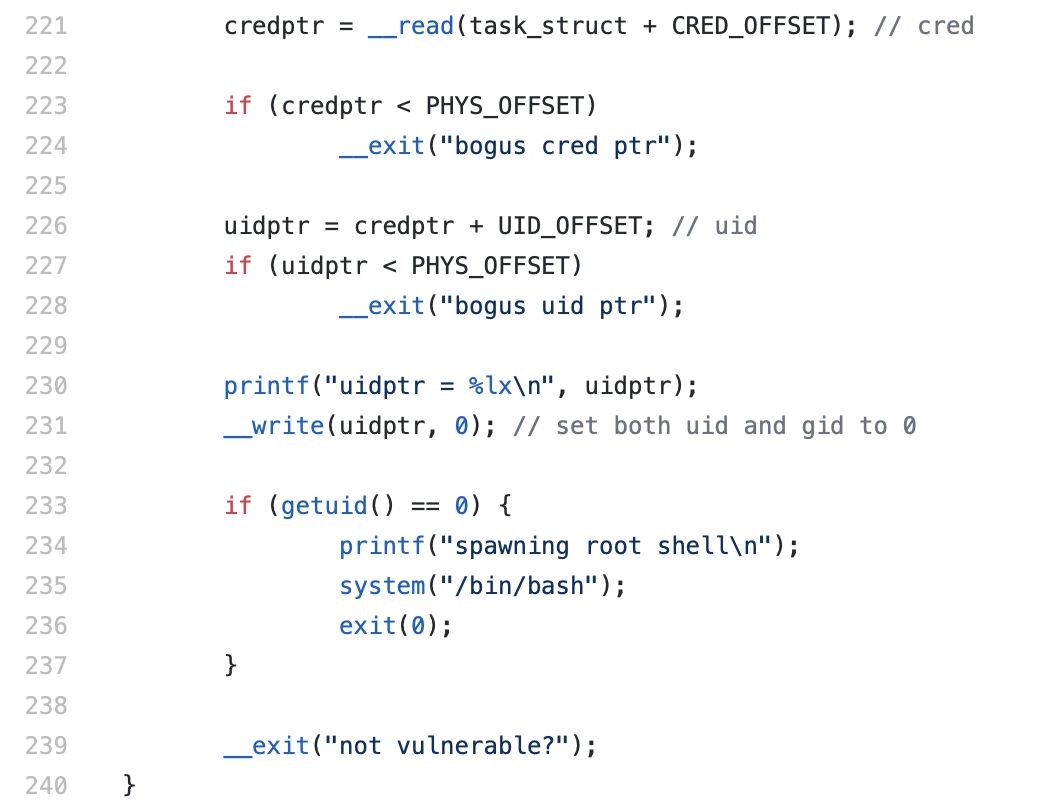


*Fig. 3.16 pwn() from exploit.c*



*Fig 3.17 struct thread\_info, from /arch/ia64/*[​*alpha*​](https://elixir.bootlin.com/linux/latest/source/arch/alpha/include/asm/latest/source/arch/alpha)*/*[​*include*​](https://elixir.bootlin.com/linux/latest/source/arch/alpha/include/asm/latest/source/arch/alpha/include)*/asm/thread\_info.h*

● Using the address of task\_struct, we would be able to access the address of the struct cred base as part of the task\_struct. There would be an uid\_t uid in the struct cred that can be set to 0 base at offset from the address of struct cred. When this uid is set to 0, the method should be allowed to operate the root rights of the system.



*Fig 3.18 pwn() from exploit.c*

**5.Conclusion**

CVE-2017-16995 is a severe Linux vulnerability which, for some reason, has received little attention. It’s a particularly nasty one because it stems from the eBPF virtual machine that’s supposed to make Linux *more*secure. It highlights again the need to ensure minimal privileges to users, and to disable syscalls where  they are not needed.

It also highlights that the use of containers can make systems more secure -- but only if they are configured properly.

Finally, this is yet more proof that no matter how much you adhere to best practices, you may still be vulnerable. Therefore, monitoring applications in runtime is not a luxury but a necessity, to allow you to detect anomalies and prevent (or at least limit) attacks.

**6.References**

1. CVE Details:<https://www.cvedetails.com/cve/CVE-2017-16995/>
2. Man page of bpf() <http://man7.org/linux/man-pages/man2/bpf.2.html>
3. Unofficial eBPF Specification <https://github.com/iovisor/bpf-docs/blob/master/eBPF.md>
4. BFP Source Code in Linux Kernel <https://elixir.bootlin.com/linux/v4.4.31/source/kernel/bpf>
5. CVE-2017-16995 Patch Status on Ubuntu <https://people.canonical.com/~ubuntu-security/cve/2017/CVE-2017-16995.html>
6. Exploit of CVE-2017-16995 <https://github.com/iBearcat/CVE-2017-16995/blob/master/exploit.c>
7. Building Ubuntu Kernel <https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel>
8. Analysis Report of CVE-2017-16995 <https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/>

<https://xz.aliyun.com/t/2212>