eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering
Computer science

End of degree work

Gaining root access in Linux using the CVE-2021-26708 vulnerability

Author

Markel Azpeitia Loiti

2021

eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

Gaining root access in Linux using the CVE-2021-26708 vulnerability

Author

Markel Azpeitia Loiti

Director(s)

Jose A. Pascual

Summary

In this project, a novel Linux kernel vulnerability discovered on February 2021 and present from kernel version 5.5 to 5.10.13 is analyzed and dissected. The vulnerability arises from a race condition residing in the virtual sockets implementation, consequence of an easily avoidable programming mistake, that causes a 4-byte write-after-free at offset 40 on a 64-byte kernel object.

Using a QEMU/KVM virtualized Linux environment alongside the GDB debugger, the inner workings of the vulnerability are analyzed, determining the reason for which it is considered a vulnerability and the conditions that must be met to to trigger it, implementing a reliable proof of concept that can cause the bug using system calls.

Furthermore, publicly available exploitation techniques like heap spraying and control flow hijacking are implemented in the C language that firstly turn the 4-byte memory corruption caused by the vulnerability's race condition into an arbitrary free exploit, which is then used for arbitrarily leaking some kernel object pointers of interest. Eventually, the leaked information is used for briefly hijacking the kernel's control flow and redirecting execution to a ROP gadget consisting of two opcodes that overwrite the kernel object storing the current user's id to zero, ultimately escalating privileges to root for all intents and purposes.

Disclaimer

Due to the nature of the exploit developed in this thesis, its implementation will not be publicly disclosed as to avoid it being used for malicious purposes.

The author of this document assumes no responsibility for possible damages that may be caused by the contents of this thesis; the contents of this document are purely for academic purposes and should not be used with any other intention.

Contents

List of Figures

List of Tables

1. CHAPTER

Introduction

In this project, a Linux kernel vulnerability known as CVE-2021-26708 [NVD, 2021] [Help Net Security, 2021] is analyzed and dissected, looking at what makes it a vulnerability, how it can be triggered and taken advantage of and ultimately demonstrating the security implications of said vulnerability and how it can be fixed.

The risk of said vulnerability is very high, as it can be exploited for local privilege escalation, meaning that an unauthorized user may gain root access to a system locally with it. The exploit has received a CVSS v3 base score of 7.0 (high severity) out of 10.

At the time of writing, there is not much publicly available information regarding the inner workings of the vulnerability or how one could exploit it. However, the patch [Popov, 2021b] regarding is very short (only one source code file changed with 12 line insertions and 5 deletions), meaning that the culprit of the vulnerability should be easy to identify.

The vulnerability results from race conditions implicitly added when virtual socket multi-transport support was featured in Linux kernel version 5.5 in November 2019. Alexander Popov, Linux kernel developer and security researcher, and the man who discovered the bug, described it like this:

1. `vsock_sock.transport` pointer is copied to a local variable.
2. `lock_sock()` is called.
3. The local variable is used.

The value of `vsock_sock.transport` may be modified or freed after its pointer is copied to the local variable, but before its lock is obtained through `lock_sock()`, leading to the following code to possibly read or write values to a modified or freed pointer, resulting in a use after free type of vulnerability.

The patch regarding the vulnerability is straightforward, as the only change that needed to be done to fix the bug is to simply copy the pointer of `vsock_sock.transport` to the local variable after the `lock_sock()` call. With a vulnerability fix this simple, identifying the error and in which scenario it is caused should not pose a problem.

2. CHAPTER

The aims of the project

The main objective of this project is to investigate and understand a recently discovered Linux kernel level vulnerability as well as analyzing its security implications, ultimately proving how a very simple programming mistake that has gone unnoticed for nearly a year and a half may be exploited to gain root-level access and ultimately defeat the security of a mature system like Linux.

The vulnerability code-named CVE-2021-26708 will be identified, examined, and dissected, demonstrating how and why it can be triggered, what consequences that may have and how it can be fixed.

Before deep-diving straight into the vulnerability, two Linux Kernel subsystems will be analyzed to provide some context: Virtual Sockets or VM sockets, and the memory management subsystem, more specifically the Buddy allocator and the Slab allocator, as well as Linux kernel security measures KASLR, SMEP & SMAP, and SELinux, put in place to protect the kernel's integrity.

Moreover, a suitable Linux kernel version and distribution will be installed to be used as the working and testing environment. The vulnerability and the consequent exploit will not work in all Linux kernel versions and distributions, so choosing a proper combination will be critical; with the work environment set, the vulnerability will be analyzed and presented, identifying how to trigger it reliably at will from user space.

Finally, common Linux kernel vulnerability exploiting techniques will be examined and presented, ultimately implementing a proof of concept exploit that takes advantage of

CVE-2021-26708 to gain root-level access on an unauthorized user account of a Linux machine.

In short, the following are the tasks that will be completed throughout the project.

1. **Research the relevant Linux subsystems**: The memory management subsystem, the virtual socket subsystem and Linux kernel security measures KASLR, SMAP & SMEP and SELinux are the most relevant.

2. **Set up the working environment**: Install a suitable Linux kernel version and the necessary development tools.

3. **Identify and analyze the bug**: Identify in which source code file and line the bug is located and what makes it a vulnerability.

4. **Trigger the bug**: Analyze how the bug can be caused through user space and develop a proof of concept to trigger it at will.

5. **Research about vulnerability exploiting**: Investigate about common vulnerability exploits in software, more specifically Linux kernel vulnerability exploits.

6. **Implement an exploit**: Develop a exploit in C that, using the vulnerability discussed in this work as an entry point, elevates privileges to root.

3. CHAPTER

Project management

In order to successfully develop a project of this scale, it is necessary to carefully think and create an accurate planning that identifies both the tasks and risks, to ease the development of the project and avoid any possible drawback or delay. Moreover, the workload and time estimates must be balanced and coherent and match the final objectives of the project.

On the same note, the project has been divided into three main stages in order to classify the basic tasks: project management, where the project's objectives and planning will be done; development, where the kernel vulnerability will be presented and analyzed alongside a proof of concept meeting the objectives of the previous stage; and finally the documentation stage, where all discoveries are presented.

Furthermore, all previously mentioned three main stages have been subdivided into smaller tasks, as can be seen in the following section.

3.1 Description of the phases

In this section the different stages of the project have been gathered to further detail and explain the sub-tasks that compose them, analyzing the usefulness of each of them and how they complement and correlate with each other.

3.1.1 Management

The main objective of the management phase is to estimate the duration and cost of all the stages and the tasks that compose them and at the same time, make up the whole project. Since this is an early stage of the project, estimating potential risks and the time that will go into the tasks accurately is of uttermost importance, as it will eventually be a reference to evaluate and assess the progress of the project, and whether there have been any unexpected deviation from the original plan. The following are the tasks that compose the management stage:

- **Planning:** This first stage aims to identify the tasks and milestones of the project, estimate their duration and the resources that may be needed for their completion. Once the tasks have been identified and estimated, the possible risks and deviations will be identified and a risk management plan done according to those to actively avoid them.
- **Tracking:** In this part, the project's progress is analyzed, checking whether the objectives are being completed in the previously estimated time and if any delays have occurred. By tracking the progress, unnecessary risks may be avoided. In addition, weekly meetings will be done with the project director to further improve the project tracking by keeping them up to date with the project development, analyzing the progress and taking more frequent steps to distribute the workload better, avoiding any unexpected delay.

3.1.2 Development

Most of the time that will be invested in the project will go to the development phase. The development phase has been further divided into three parts:

- **Research**: The field that is being worked on in this project is unknown to the author of the project and as such researching and understanding the underlying concepts is necessary before any further development is done. Several fields will have to be researched:
  - VSOCK, VM Sockets, and the Linux memory subsystem
  - Use After Free vulnerability

  - Linux privilege escalation techniques

- **Preparing the working environment**: The vulnerability in question affected Linux kernel versions 5.5 (including) to 5.10.13 (excluding). However, these were all patched [Popov, 2021b], and therefore, in order to work with an exploitable system, the kernel will have to be recompiled after removing said patches from its source code. Moreover, KGDB [Wessel, 2010], a Linux kernel debugger, will be set up to help in the debugging process.

- **Vulnerability exploiting**: Once the vulnerability has been identified, analyzed and understood, it will be exploited to prove the consequences that a seemingly small bug can have on the integrity of a system. A proof of concept will be developed that exploits it to escalate privileges from a local unauthorized user.

3.1.3 Documentation

The last part of the project consists of gathering all the researched information, progress through development and achieved results in a presentable manner.

LaTeXwill be used to redact the end of degree work in an ordered and elegant manner using the online tool *Overleaf*. This way, the document may be accessed and consulted at any time by the director as well.

3.2 Work Breakdown Structure

The Work Breakdown Structure (WBS) is used below to better present the main stages and sub-tasks that the project will go through:

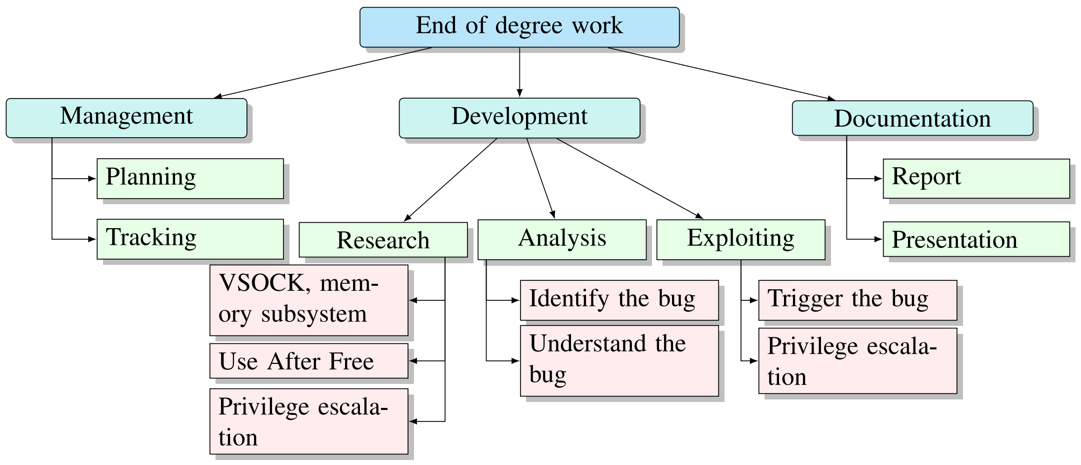


Figure 3.1: WBS diagram

3.3 Time estimates

In the following table (see Table 3.1), the estimated time of completion and the real invested time of each phase and sub-task is presented in hours. The estimated and final invested time hours of sub-tasks are summed in each phase.

	Estimated time (h)	**Final time (h)**
Management phase	20	20
Planning	10	12
Tracking	10	8
Development phase	220	233
Research	100	85
Analysis	50	43
Exploiting	70	105
Documentation phase	60	67
Report	45	55
Presentation	15	12
Total time sum	300	320

Table 3.1: Project time estimates and final times

3.4 Risk management

In projects of this scale, unforeseen delays or problems that affect its proper development are expected regardless of the topic. That is why it is essential to identify the potential risks and prepare a risk management plan to avoid or reduce the impact of any possible drawback.

Usually, the more extensive the workload of a stage, the more bound it is to suffer from delays. Therefore, the estimated times of those stages are increased to account for delays, while the time estimates of the tasks with smaller workloads are expected to be more accurate and less extra time is assigned to them.

3.5 Deviation analysis

Although some deviations have occurred with the estimated times, the initial planning has mostly been kept during development as no considerable delay or problem affected it.

The major deviation corresponds to the exploit implementation phase. In the first stages of this phase, a custom kernel was compiled for Ubuntu 20.04 to be used as the working environment. However, this distribution resulted inadequate for the exploit and had to be scrapped in favor of Fedora 33 Server, resulting in having to recompile the kernel.

Moreover, kernel crashes were a very common occurrence during the development and debugging of the exploit, resulting in having to reboot the target machine and set up the working environment several times throughout each work session, hindering development greatly.

3.6 Work methodology

In this section, several choices and decisions made about the employed work methodology are presented and explained.

3.6.1 Meetings

During the whole project life cycle, the end of degree work author and the project director will not hold face-to-face meetings due to geographical constraints. Instead, weekly online meetings will be used to discuss, share and keep track of the project through an online meetings platform. Moreover, email communication will always be open to answering any question deemed too simple for having a meeting.

3.6.2 Work schedule

In order to comply with the set time estimates, it was decided that the student will work on the project every week from February to May, while having a regular working schedule each week.

4. CHAPTER

Linux subsystems

Before examining the vulnerability at hand, it is necessary to understand the inner workings of the affected system; in this chapter, the Linux kernel subsystems most relevant to the vulnerability are examined and presented: memory management, virtual sockets and Linux kernel security mechanisms.

4.1 Memory management subsystem

The Linux memory management subsystem is responsible, as the name implies, for managing the memory in the system, which includes implementation of virtual memory and demand paging, memory allocation both for internal kernel structures and user space programs, mapping of files into processes' address space, etc.

The most relevant memory management concepts to the topic, the Buddy allocator and the Slab allocator, are presented below to provide some context and better understand the forthcoming vulnerability analysis and exploit.

4.1.1 Buddy allocator

The Buddy allocator is the main algorithm used in Linux for physical page management. The basic principle behind it is the following: physical memory is broken up into large chunks of memory where each chunk is of size $2^n \times PAGE\_SIZE$ ($PAGE\_SIZE$ being a constant in the kernel, usually 4096 bytes). Whenever a block of memory has to be allocated, and the requested size is not available, one of the aforementioned big chunks is halved continuously until a block that wastes the minimum amount of memory and meets the size requirement is found. Every two broken blocks of the same size are *buddies* to each other; one half is used for allocation, and the other is free. When a block is later freed, its *buddy* is examined, and if free they are both joined again.

One fundamental property of the Buddy allocator is that it allocates physically contiguous memory blocks, a property that will prove helpful later on for exploiting. However, this memory management scheme has a big problem: internal fragmentation.

If a process requires to allocate 33 pages, then the minimum quantity of pages that the Buddy allocator can hand to the process will be $2^6 = 64$ because $2^5 = 32 < 33$, meaning that 31 pages are wasted out of the 64 that were allocated, as can be seen in Figure 4.1.

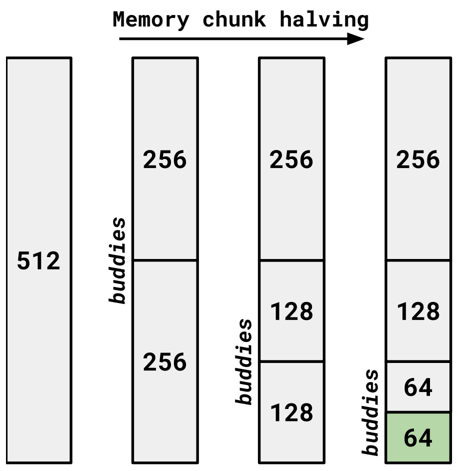


Figure 4.1: Buddy allocator, 33 page allocation request

The internal fragmentation problem is addressed in Linux using the Slab allocator, which slices memory pages into smaller chunks of memory (slabs) for further allocation. With this combination of allocators, the kernel ensures that wasted space due to internal allocation is kept to a minimum.

4.1.2 Slab allocator

The main idea of the Slab allocator [The kernel development community, 2008b] [Dio, 2020] is to keep caches of commonly used objects in an initialized state available for allocation in the kernel. Without an object-based allocator like this the kernel would spend much of its time allocating, initializing and freeing the same objects over and over again. The Slab allocator aims to cache freed kernel objects so that their basic structures are preserved between uses.

The Slab allocator provides two main classes of caches:

- **Dedicated**. These are caches for objects that are commonly used in the kernel, such as task_struct, mm_struct, cred_jar and more.
- **Generic**. These are general purpose caches, usually of sizes corresponding to powers of two.

Different information about the available slab caches can be consulted in the dedicated file */proc/slabinfo* as can be seen in Figure 4.2 (several rows and columns have been removed for brevity):

```
slabinfo - version: 2.1
# name  <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
# Dedicated Caches
inode_cache        37895  37895    608   53    8
dentry            110993 111720    192   42    2
mm_struct            210    210   1088   30    8
files_cache          322    322    704   46    8
cred_jar            1218   1218    192   42    2
pid                 1600   1600    128   64    2
radix_tree_node    12712  12712    584   56    8
# General purpose caches
dma-kmalloc-8k         0      0   8192    4    8
dma-kmalloc-96         0      0     96   42    1
kmalloc-8k           132    136   8192    4    8
kmalloc-64          9920   9920     64   64    1
kmalloc-32         22144  22144     32  128    1
kmalloc-16         16896  16896     16  256    1
kmalloc-8          12288  12288      8  512    1
kmem_cache_node      256    256     64   64    1
kmem_cache           256    256    256   64    4
```

Figure 4.2: Output of slab cache info file */proc/slabinfo*

In Figure 4.3 an example of three pages of `PAGE_SIZE` bytes reserved for three different generic slab caches, `kmalloc-1024`, `kmalloc-512`, and `kmalloc-64` are shown.

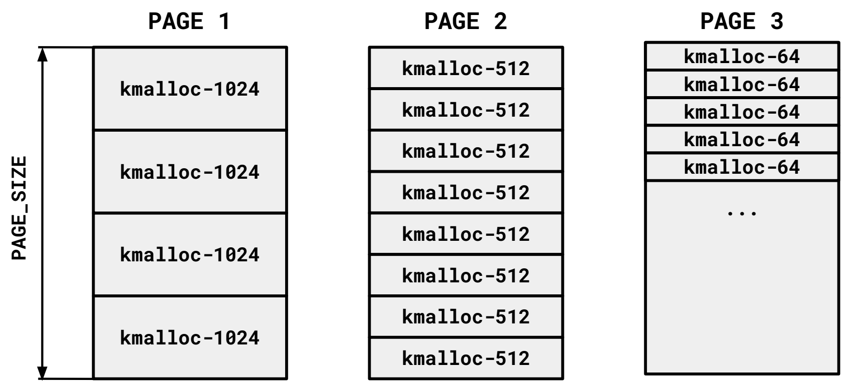


Figure 4.3: Slab allocator, example of generic slab caches

In order to do allocations of generic slab caches the `kmalloc` interface is used, defined in Linux kernel source code file *include/linux/slab.h* as follows in Listing 4.1:

```
// allocates memory generic slab allocator cache.
static __always_inline void *kmalloc(size_t size, gfp_t flags);

// frees previously allocated generic slab cache.
void kfree(const void *);
```

Listing 4.1: Extract of Linux kernel source code file *include/linux/slab.h*

Consequently, similar to how memory allocation is done from user space, `kmalloc()` is invoked for allocating a generic slab cache and `kfree()` for freeing it.

4.2 Linux virtual socket subsytem

Linux Virtual Sockets (VSOCK) or VM sockets [King, 2013] allow communication between virtual machines and their hypervisor (the host machine). User-level applications in both the host and the guest virtual machine can use the VM socket API to quickly and efficiently communicate. The use cases include clipboard sharing, mouse integration, automatic adjustment of video resolution, guest control and remote console, to name a few.

For identification purposes, port numbers which are represented using 32 bits and most importantly, *CID*s (Context IDentifiers, 32 bits) are used in the context of VM sockets, the most important ones being [Garzarella, 2020]:

- `VMADDR_CID_ANY` (`0xFFFFFFFF`): Use any address for binding.
- `VMADDR_CID_LOCAL` (`1`): Address for local communication (loopback).
- `VMADDR_CID_HOST` (2): Address of the host.

Moreover, VSOCK supports the standard Linux socket API, meaning that all Linux socket functions are supported: `socket()`, `bind()`, `listen()`, `connect()`, `send()`, `recv()`, etc.

4.3 Linux kernel security measures

The following are some of the most critical Linux security measures intended to keep the integrity of the kernel intact and avoid active attacks against it.

4.3.1 SMEP & SMAP

Supervisor Mode Execution Prevention (SMEP) [Intel Corporation, 2018] and Supervisor Mode Access Prevention (SMAP) [Wikipedia, 2021] [Intel Corporation, 2015] are CPU based security mechanisms that provide user space address-space protection. SMEP prevents unauthorized supervisor mode **execution** from user pages, while SMAP prevents unintended supervisor mode **access** (read & write) to data in user pages.

Without SMEP and SMAP, kernel space code has full read & write access to the whole memory address layout, including user space memory addresses. Several kernel exploits, including privilege escalation exploits, rely on this to gain kernel privileges from user space. SMEP and SMAP aim to prevent this type of exploits and many more from succeeding.

SMEP and SMAP are enabled when memory paging is active and the SMEP and SMAP bits of the CR4 CPU control register (bit 20 and 21, respectively) are set, as shown in Figure 4.4.

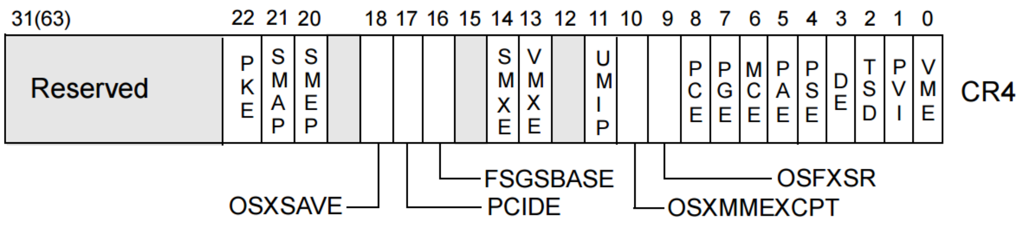


Figure 4.4: CR4 register control bits

SMEP and SMAP are supported since mainline Linux Kernel v3.7 and are enabled by default on all CPUs that support these features (all modern Intel and AMD x86-64 CPUs do).

4.3.2 Kernel Address Space Layout Randomization

Kernel Address Space Layout Randomization (KASLR) [Edge, 2013], or ASLR [Mordechai Guri, 2015] applied to a kernel, is a security mechanism that aims to prevent exploitation of memory-corruption vulnerabilities by randomly arranging the address space positions of key data areas of a process or kernel, such as the base address of an executable and the position of libraries, heap, and stack, instead of placing them in fixed addresses.

ASLR makes redirection of program/kernel execution flow more difficult as the addresses of functions are randomized on each execution.

The Linux kernel has KASLR enabled by default since version 4.12; when activated, a constant is randomly calculated on boot, which serves as the KASLR offset. In the following Figure 4.5, the memory layout of a program with ASLR through different executions is shown.

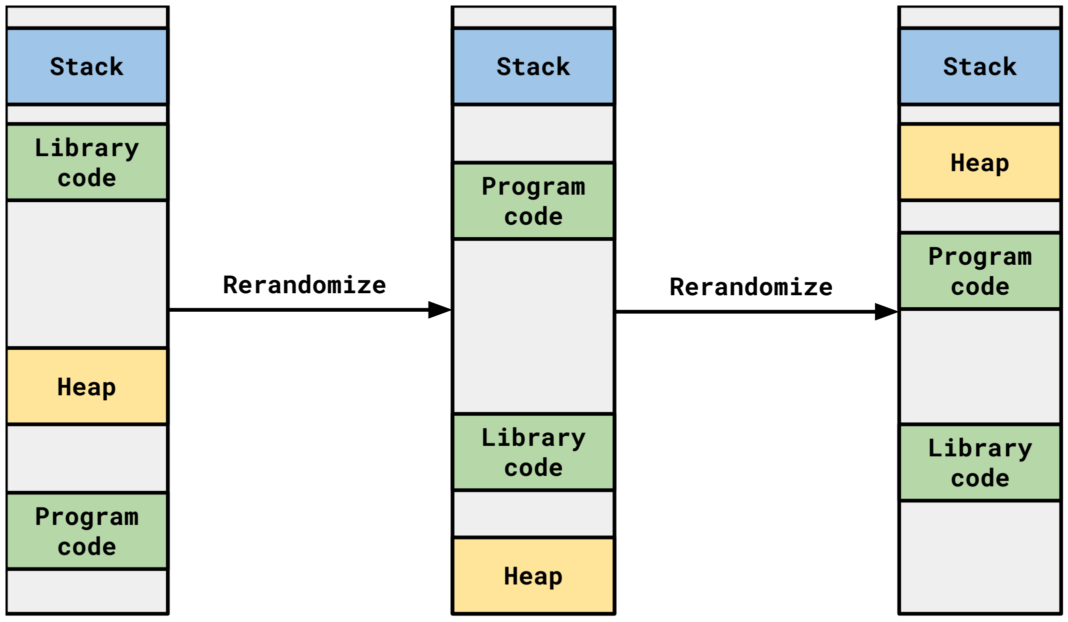


Figure 4.5: Memory layout of ASLR enabled program through different executions

The aforementioned security measures, SMEP & SMAP, and KASLR, keep the most basic attacks against the kernel at bay, but many techniques render them useless and allow for successful attacks

4.3.3 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) is a security architecture integrated into Linux systems that allows administrators to have more control over who can access the system's resources through mandatory access control (MAC).

SELinux will not be a security measure that will play an active role in preventing the vulnerability or the exploit to succeed. Instead, its implementation will be taken advantage of to develop the exploit.

5. CHAPTER

Common software vulnerabilities

A software vulnerability is a weakness or flaw in a system that an attacker can exploit to perform unauthorized actions such as reading or writing to restricted areas or execute limited operations. Vulnerabilities are common in all kinds of software, from web browsers to operating systems, mainly caused by design flaws and their ever-increasing complexity.

Many different kinds of software vulnerabilities exist, and depending on the platform in use, some types of software may be more prone to have some sort of vulnerabilities than others.

The following are some of the most usual vulnerabilities that can be found in software [CWE, 2020b]:

- **Cross-site scripting (XSS)**: typically found in web applications, allows attackers to inject malicious client-side scripts in websites which get executed by other users visiting the website.
- **Improper input validation (e.g. SQL Injection)**: When software does not validate input properly, an attacker is able to craft the input in a way that is not expected by the application.
- **Buffer overflow**: a vulnerability in which a program overruns the boundaries of the buffer it was writing to, overwriting adjacent memory.
- **Use After Free**: the use of a previously freed memory.

The vulnerability analyzed in this work is a Use After Free (UAF) one. A UAF vulnerability (CWE-416) [CWE, 2020a] refers to the referencing of a memory location after this has been freed, which can cause a program to crash, use unexpected values, corrupt previously valid data or execute arbitrary code.

The code in Listing 5.1 illustrates the simplest pattern of a Use After Free error. The reason this is a bug is that after the `free(ptr)` call, the content of memory being referenced by `ptr` is unknown, and any operation with it will result in undefined behavior. Pointer `ptr` is now a dangling pointer, a pointer that references an already freed memory location.

```c
int *ptr = (int*) malloc(sizeof(int));
*ptr = 23;
free(ptr);
*ptr = 42; // Use After Free
```

Listing 5.1: Example of a UAF vulnerability pattern

5.1 Exploiting

UAF exploiting techniques will differ depending on the system they are running on and the implementation of the executable in question. The following is a basic UAF exploiting scenario that leads to execution flow redirection:

1. **Allocate memory *A***: An allocation is made where we provide a struct containing a function pointer allocated in the heap.

2. **Free memory *A***: The previously allocated memory is freed.

3. **Allocate memory *B***: Allocate a memory chunk containing malicious code, aiming to replace the chunk that the first allocation A had.

4. **Reference allocation *A***: When the function member of the struct we created is called, we expect it to call the function from allocation A. However it will call the malicious function from allocation B.

5.2 Proof of concept

The following is a proof of concept that demonstrates how the aforementioned use-after-free vulnerability may be exploited on a Linux machine [Jimenez, 2017]. As we can see in Listing 5.2, the code starts off with a struct definition, `struct vuln_t`, containing a function pointer which is called `vulnfunc()`:

```c
typedef struct vuln_t {
    void (*vulnfunc)();
} vuln_t;
```

Listing 5.2: Vulnerable `struct vuln_t` declaration

Afterwards we have two global function definitions as shown in Listing 5.3. The aim of the exploit is to call function `bad()` without explicitly referencing it from the pointer `vulnfunc()` in `struct vuln_t`.

```c
void good(){
    printf("I AM GOOD :)\n");
}

void bad(){
    printf("I AM BAD >:|\n");
}
```

Listing 5.3: Function declarations for UAF

In addition, the `main()` function is as follows in Listing 5.4:

```c
vuln_t *malloc1 = malloc(sizeof(vuln_t));
malloc1->vulnfunc = good;
malloc1->vulnfunc();
free(malloc1);
long *malloc2 = malloc(0);
*malloc2 = (long)bad;
malloc1->vulnfunc();
```

```
return 0;
```

Listing 5.4: `main()` function declaration for UAF

On lines (1),(2), and (3) of Listing 5.4 a `vuln_t struct` is allocated, function pointer "good" assigned to its member `vulnfunc()` and then called, resulting in output "I AM GOOD :)", as expected. Next `vuln_t struct` is freed (4) and immediately more memory is allocated (5) [1] and function pointer "bad" assigned to it (6), taking the memory location in the program heap that function pointer "good" previously had. Finally, when `vulnfunc()` is called from the now dangling pointer function `bad()` will be called instead, resulting in output "I AM BAD >:|"

It should be noted that the aforementioned situation is very unlikely to happen in a real environment, as one usually has to deal with security measures and more complex data types. However, the main concept behind UAF exploiting remains.

[1] malloc(0) behavior is implementation dependent, i.e. not defined by any standard.

6. CHAPTER

CVE-2021-26708 Vulnerability

In this chapter the vulnerability under discussion, CVE-2021-26708, is examined, analyzed and discussed in depth, as well as the steps needed to successfully and reliably trigger it with a user space program.

6.1 Vulnerability description

Up until the kernel commit that added the VSOCK multi-transports support, in the context of virtualization each kernel (both host and guest) could only register one VSOCK socket, which proved to be problematic in nested virtual machine environments. For example, if a Linux host had a VMware virtual machine running and this virtual machine had a QEMU/KVM machine running in it at the same time, the VMWare machine could only register one VSOCK, meaning that it could only communicate with its host (the Linux machine) or its guest (QEMU/KVM virtual machine), but never with both at the same time.

This situation was addressed by adding multi-transports support to Linux v5.5, which implicitly added the vulnerability discussed.

The vulnerability in question lies in Linux source file `net/vmw_vsock/af_vsock.c`, and the affected functions as seen in the bug fix [Popov, 2021b] are:

- `vsock_poll()`: Wait for some event on a virtual socket.

- `vsock_dgram_sendmsg()`: Send a message to a connectionless virtual socket.
- `vsock_stream_setsockopt()`: Set options for a connection-oriented stream virtual socket.
- `vsock_stream_sendmsg()`: Send a message to a connection-oriented virtual socket.
- `vsock_stream_recvmsg()`: Receive a message from a connection-oriented virtual socket.

All the changes in the patch and in the mentioned functions involve moving operation `transport = vsk->transport`, so that instead of being executed before calling the function `lock_sock(sk)`, it is executed immediately after it, as shown in Listing 6.1 (where the red background represents the line that was removed in the patch, while the green background shows the lines that were added).

```
1  -   const struct vsock\_transport *transport = vsk->transport;
2  +   const struct vsock_transport *transport;
3      lock_sock(sk);
4  +   transport = vsk->transport;
```

Listing 6.1: CVE-2021-26708 vulnerability's patch extract

In other words, with the fix applied the pointer of the socket transport (`vsk->transport`) is saved into a local variable (`const struct vsock_transport *transport`, line 1 in Listing 6.1) only after its lock has been acquired in function `lock_sock(sk)` (line 3) and never before.

After analyzing the patch, it is clear that the problem lies in that a thread A may save the pointer to the transport into a local variable while its corresponding memory region is freed in a different thread B before thread A acquires the lock, leading to a dangling pointer that may cause memory corruption that could consequently be exploited to be used as a vulnerability to gain privilege escalation; a textbook race condition, as shown in Figure 6.1.

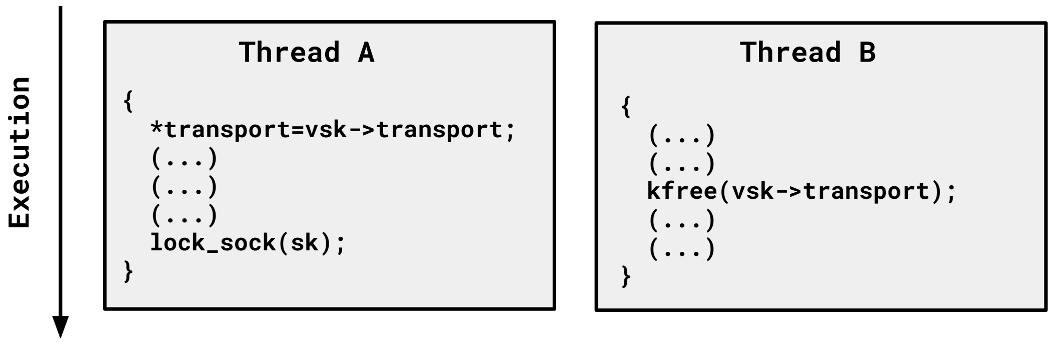


Figure 6.1: CVE-2021-26708 race condition

To sum it up, for the bug to show itself, the following must happen:

1. Pointer to socket transport (`vsk->transport`) is saved into a local variable in thread A.
2. Thread B obtains the socket lock, and the transport object is freed from memory with `kfree()`.
3. Thread A acquires the lock and uses its reference, now a dangling pointer, to execute operations on it.

Once an operation is executed on the dangling pointer, unpredictable behavior may occur during the lifespan of the local variable holding the incorrect pointer, as the content of the memory location that it points to is unknown.

6.2 Reproducing the bug

In this section, the steps that have to be taken to reproduce the bug reliably are presented.

6.2.1 Saving the pointer to the socket transport into a local variable

The first step to reproduce the bug is to have the reference or pointer to a socket transport be saved into a local scope variable before acquiring its lock. This sequence can be identified by looking at the code snippet shown in Listing 6.2

```
transport = vsk->transport;
```

```
lock_sock(sk);
```

Listing 6.2: Code snippet that causes the race condition

The aforementioned sequence in Listing 6.2 is found in the five VSOCK related Linux kernel functions that were patched.

6.2.2 Freeing the socket transport

The next step would be to free the memory region occupied by the socket transport on a different thread before obtaining the lock. But, of course, this can not be forced with a simple free() call from user space, as the memory region holding it is in kernel space, and therefore, must be achieved through kernel system calls.

The value of the socket transport (`vsk->transport`) may be modified in two kernel functions:

- `vsock_deassign_transport()`: where it is destructed, and its value set to `NULL`.
- `vsock_assign_transport()`: where a new transport struct is assigned to it.

Function `vsock_deassign_transport()` (see Listing 6.3) calls socket transport's function `destruct()` in line 6 (defined in `/net/vmw_vsock/virtio_transport_common.c`), which in turn calls `kfree()`, as shown in line 4 of Listing 6.4.

```
1 static void vsock_deassign_transport(struct vsock_sock *vsk)
2 {
3     if (!vsk->transport)
4         return;
5 
6     vsk->transport->destruct(vsk);
7     module_put(vsk->transport->module);
8     vsk->transport = NULL;
9 }
```

Listing 6.3: Function `vsock_deassign_transport()`

```c
void virtio_transport_destruct(struct vsock_sock *vsk)
{
    struct virtio_vsock_sock *vvs = vsk->trans;
    kfree(vvs);
}
```

Listing 6.4: Function `virtio_transport_destruct()`

On the other hand, `vsock_assign_transport()` will assign a new transport to the given socket depending on the socket's current CID as described by a comment in source code shown in Listing 6.5.

```c
/* The vsk->remote_addr is used to decide which transport to use:
 * - remote CID == VMADDR_CID_LOCAL or g2h->local_cid or
   VMADDR_CID_HOST if
 *   g2h is not loaded, will use local transport;
 * - remote CID <= VMADDR_CID_HOST will use guest->host transport;
 * - remote CID > VMADDR_CID_HOST will use host->guest transport;
 */
```

Listing 6.5: Kernel source code comment on transport assignment

If the given transport is the same as the socket already had, no changes are made. However, if the transport is different, then the previous transport is freed by calling kernel function `vsock_deassign_transport()`. Therefore, if a transport is changed, the previously allocated one gets freed from memory. With this information, we can free the memory region occupied by the socket transport at will.

6.2.3 Using the invalid pointer

The last step to reproduce the bug is to determine whether the previously acquired local variable, now with an invalid pointer, is used in any of the functions in which the bug is located.

One of these functions is `vsock_stream_setsockopt()` which when certain circumstances are met calls function `vsock_update_buffer_size()` (see Listing 6.6), which in turn calls socket transport member function `notify_buffer_size()` (line 10 in Listing 6.6). This function will write a 4-byte value given as an argument in offset 40 of the now dangling pointer (see line 8 of Listing 6.7).

```c
1  static void vsock_update_buffer_size(struct vsock_sock *vsk, const
       struct vsock_transport *transport, u64 val)
2  {
3      if (val > vsk->buffer_max_size)
4          val = vsk->buffer_max_size;
5
6      if (val < vsk->buffer_min_size)
7          val = vsk->buffer_min_size;
8
9      if (val != vsk->buffer_size && transport && transport->
     notify_buffer_size)
10         transport->notify_buffer_size(vsk, &val);
11
12     vsk->buffer_size = val;
13 }
```

Listing 6.6: Function `vsock_update_buffer_size()`

```c
1  void virtio_transport_notify_buffer_size(struct vsock_sock *vsk,
     u64 *val)
2  {
3      struct virtio_vsock_sock *vvs = vsk->trans;
4
5      if (*val > VIRTIO_VSOCK_MAX_BUF_SIZE)
6          *val = VIRTIO_VSOCK_MAX_BUF_SIZE;
7
8      vvs->buf_alloc = *val;
9
10     virtio_transport_send_credit_update(vsk,
     VIRTIO_VSOCK_TYPE_STREAM, NULL);
11 }
```

Listing 6.7: Function `virtio_transport_notify_buffer_size()`

6.3 Triggering the bug

All previously mentioned `VSOCK` functions are kernel functions, meaning that they are called by the kernel itself and cannot be used or invoked from a user created program directly. Therefore, if the kernel is to call those functions in a certain order to trigger the bug, a user space program will have to be carefully crafted using system calls, which will call those kernel functions under certain circumstances, and force the kernel to trigger the bug.

In order to trigger the vulnerability we want to create a race condition between kernel functions `vsock_stream_setsockopt()` and `vsock_stream_connect()`. The former will save the transport into a local variable and the latter will cause the free, leaving the dangling pointer in `vsock_stream_setsockopt()` function's local variable and possibly writing on it.

Linux system call `setsockopt()` with `AF_VSOCK` family and `SOCK_STREAM` socket type as arguments will be used to reach `vsock_stream_setsockopt()`.

Next, we want `vsock_assign_transport()` to call `vsock_deassign_transport()` to free the occupied memory region. System call `connect()` will be used to call function `vsock_stream_connect()`, which, if the socket's state is `SS_CONNECTING`, will call `vsock_assign_transport()` (see line 13 of Listing 6.8). Once this kernel function is called if the socket is of type `SOCK_STREAM` and the transport is changed then the previously referenced transport will be freed, leaving the dangling pointer in function `vsock_stream_setsockopt()`.

```
1  switch (sock->state) {
2      case SS_CONNECTED:
3          err = -EISCONN;
4          goto out;
5      case SS_DISCONNECTING:
6          err = -EINVAL;
7          goto out;
```

```
    case SS_CONNECTING:
        err = -EALREADY;
        break;
    default:
        (...)
        err = vsock_assign_transport(vsk, NULL);
```

Listing 6.8: Extract from `vsock_stream_connect()`

6.4 Proof of concept

The vulnerability depends on a race between obtaining the reference to the transport and obtaining the socket lock. The time that elapses between these operations is very small, a minuscule window in which it can be triggered.

In order to trigger the bug from a user space program, first, a socket stream of type `AF_VSOCK` is created, which will try to connect to a nonexistent local server by using CID `VMADDR_CID_LOCAL` as argument in order to set the socket's transport to the loopback transport, as can be seen in Listing 6.9.

```
vsock = socket(AF_VSOCK, SOCK_STREAM, 0);
addr.svm_cid = VMADDR_CID_LOCAL;
connect(vsock, (struct sockaddr *)&addr, sizeof(struct sockaddr_vm
    ));
```

Listing 6.9: Initializing VSOCK for triggering the vulnerability

Following the first call to `connect()`, two new threads are created as shown in Listing 6.10; one will call `setsockopt()`, which will hold the pointer to the socket transport while the second thread will call `connect()` once again but with a different target CID (`VMADDR_CID_HYPERVISOR`) to force the socket transport to be freed.

```
// Thread A
addr.svm_cid = VMADDR_CID_HYPERVISOR;
connect(vsock, (struct sockaddr *)&addr, sizeof(struct sockaddr_vm
    ));
```

```

// Thread B
setsockopt(vsock, PF_VSOCK, SO_VM_SOCKETS_BUFFER_SIZE, &val,
    sizeof(unsigned long));
```

Listing 6.10: Code to trigger the race condition

This is when the race starts: If the `connect()` thread obtains the lock first, it will end up calling `kfree()` on the transport whose pointer `setsockopt()` holds. When `setsockopt()` acquires the lock next it will call `vvs->buf_alloc = *val;` (as seen in line 8 of Listing 6.6) causing a memory corruption by writing to a freed region. Otherwise, if the `setsockopt()` thread obtains the lock first, then the race is lost, but the process can be repeated safely.

When instruction `vvs->buf_alloc = *val;` is executed, an arbitrary value of 4-bytes specified in the argument `val` of `setsockopt()` will be written in a `kmalloc-64` slab cache at offset 40. The reason behind this is that variable `vvs`, to which `setsockopt()` writes, is of type `struct virtio_vsock_sock` and its member variable `buf_alloc` is of type `u32` (unsigned, 32 bits) and resides at offset 40 (line 9), as seen in Listing 6.11. Moreover, the reason this kernel object is placed at a `kmalloc-64` is that it is 64-bytes in size.

```
1  struct virtio_vsock_sock {       // Offset
2      struct vsock_sock *vsk;      // 0
3      u32 buf_size;                // 8
4      u32 buf_size_min;            // 12
5      u32 buf_size_max;            // 16
6      spinlock_t tx_lock;          // 20
7      spinlock_t rx_lock;          // 28
8      u32 tx_cnt;                  // 36
9      u32 buf_alloc;               // 40
10     ...
11 };
```

Listing 6.11: `struct virtio_vsock_sock`'s definition

6.5 Kernel information leak

Every time the vulnerability is triggered, the kernel will show a warning in the log file `/dev/kmsg`, as can be seen in Figure 6.2:

```
WARNING: CPU: 1 PID: 42067 at net/vmw_vsock/virtio_transport_common.c:34
...
CPU: 1 PID: 42067 Comm: vuln Tainted: G        W         5.10.11+ #2
Hardware name: QEMU Standard PC (Q35 + ICH9, 2009), BIOS 1.13.0-1ubuntu1.1 04/01/2014
RIP: 0010:virtio_transport_send_pkt_info+0x14d/0x180 [vmw_vsock_virtio_transport_common]
...
RSP: 0018:ffffb899c8487e10 EFLAGS: 00010246
RAX: 0000000000000000 RBX: ffff8f9803494e40 RCX: ffff8f980db131c0
RDX: 00000000ffffffff RSI: ffffb899c8487e58 RDI: ffff8f9803494e40
RBP: 0000000000000000 R08: 000000002d6dc425 R09: 0000000000000000
R10: 00000000000003c4 R11: 0000000000000000 R12: 0000000000000008
R13: ffffb899c8487e58 R14: 0000000000000000 R15: ffff8f9803494e40
FS:  00007fbe81f4a640(0000) GS:ffff8f987dd00000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 00007f4a1a07c000 CR3: 00000000120f0001 CR4: 0000000000370ee0
Call Trace:
 virtio_transport_notify_buffer_size+0x60/0x70 [vmw_vsock_virtio_transport_common]
 vsock_update_buffer_size+0x5f/0x70 [vsock]
 vsock_stream_setsockopt+0x128/0x270 [vsock]
...
```

Figure 6.2: Warning in `/dev/kmsg` due to dangling pointer

This warning shows which process has caused the vulnerability to trigger: (`CPU: 1 PID: 1 Comm vuln`), the call trace leading to it (`vsock_stream_setsockopt()`, `vsock_update_buffer_size()`, `virtio_transport_notify_buffer_size`) and the values of several CPU registers at the time. By using GDB it was discovered that the value of register RBX corresponds to the kernel address of the kernel object `vsock_sock` in this context, while register RCX holds the kernel address of the just freed `virtio_vsock_sock`. This information will prove very useful when developing an exploit: The pointer of the freed `virtio_vsock_sock` will be used for arbitrarily freeing another kernel object, which will allow us to leak the contents of `vsock_sock`; this will be used to calculate the KASLR offset and further develop the exploit.

7. CHAPTER

Exploiting the vulnerability

In this chapter, a method of exploiting the vulnerability will be shown and implemented step by step, ultimately escalating privileges to root from an unauthorized user.

The methodology of the exploit was presented by Alexander Popov on his website [Popov, 2021a]. However, all of the programming and implementation of the exploit was done by the author of this investigation work, as the implementation was not disclosed publicly at the time of writing.

7.1 Environment Setup

Before developing the exploit, it is necessary to set up a proper working environment in which the vulnerability can be triggered, and the kernel debugged.

7.1.1 OS & Kernel

As seen in previous sections, the vulnerability was fixed [Popov, 2021b] and backported into all affected stable trees, meaning that the previously affected versions can not be exploited anymore. For this reason, it is necessary to edit and compile a custom Linux kernel with these patches removed in order to trigger the vulnerability.

The exploit that will be presented below relies on several key OS features that must be present on the target machine for successful execution, namely:

- SELinux enabled
- Permission to read the kernel system log as an unprivileged user
- Permission to use `userfaultfd()` as an unprivileged user

If any of the previously mentioned features is not present, the exploit will fail, possibly causing a kernel crash.

The OS of choice for testing and developing the exploit will be Fedora 33 Server for x86_64 as it meets these requirements out of the box. A custom Linux kernel version 5.10.13 will be compiled with the relevant vulnerability related patch reverted in the source code file `net/vmw_vsock/af_vsock.c` in order to be able to trigger the bug successfully. This Fedora machine will be installed on a QEMU/KVM virtual machine to allow for debugging via GDB.

7.1.2 Debugging

In order to ensure the proper development of the kernel exploit, GDB will be used to check Linux kernel variable values, probe memory addresses on run-time, set breakpoints and halt execution at will.

7.2 Exploit step-by-step

A simplified step-by-step procedure, which shows the basics of the exploit, is presented next; more details about the steps are given in the next sections.

1. Get good `msg_msg` addresses
   - Provoke the race condition and heap spray the memory with `msgsnd()`.
   - Read log file `/dev/kmsg` to collect the kernel addresses of the heap sprayed `msg_msg` objects.
2. Arbitrary free
   - Provoke the race condition and heap spray the memory again with `msgsnd()`, this time writing the address of a leaked good `msg_msg` in the UAF.

   - Repeat the previous step until a good `msg_msg` is freed.

3. Arbitrary read

   - Heap spray the arbitrarily freed address with a fake `msg_msg` using `setxattr()` & `userfaultfd()`.
   - Receive the heap sprayed message with `msgrcv()`, which will leak the contents of kernel object `vsock_sock`.
   - `vsock_sock` will contain the address of the credentials object; it will also be used to calculate the KASLR offset and the address of a possible `sk_buff` socket buffer.

4. Arbitrary write

   - Heap spray the memory by sending UDP messages.
   - Arbitrary free one of the previously sprayed UDP messages.
   - Heap spray the arbitrarily freed message with a fake `sk_buff` containing a ROP gadget.
   - Receive the replaced UDP message, which will overwrite the credentials object and escalate privileges.

These are the simplified steps to escalate privileges using CVE-2021-26708. In the following section these steps are developed further.

7.3 Memory Corruption

The vulnerability's race condition may cause a write after free of a 4-byte value on a 64-byte kernel object at offset 40, which may not seem very useful at first glance. However, if a proper kernel object is placed in the memory address where the free was caused, and its value at offset 40 is arbitrarily overwritten, the vulnerability can be turned into a dangerous exploit. A technique known as *heap spraying* will be used to place a desired kernel object in the previously freed address and cause the vulnerability to write on it.

7.3.1 Heap Spraying technique

Heap spraying refers to indirectly causing many sequential memory allocations hoping that one of these allocations is performed on a desired kernel address. Heap spraying is possible thanks to the slab allocator, which always tries to allocate the most recently freed memory addresses. Many heap spraying techniques can be carried out in Linux, one of which is performed by using the system call `msgsnd()`.

In Figure 7.1, the result of a simple memory heap spraying is shown. By provoking the allocation of several memory locations, a target memory region which was recently freed is allocated again alongside other unrelated regions.

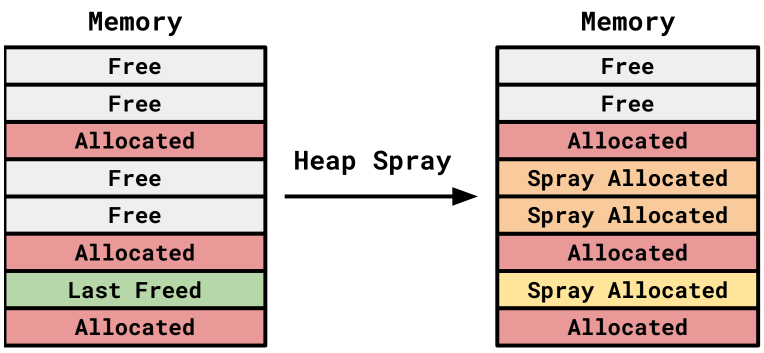


Figure 7.1: Heap spraying

The system call `msgsnd()` is part of the System V message queue operations and is used to send messages to a message queue, allowing for inter-process communication. Each time a 16-byte message is sent with this system call a 44-byte sized kernel object, `struct msg_msg` followed by the message itself, is allocated into the `kmalloc-64` slab cache, as both together take up 60 bytes in space.

Therefore, if `msgsnd()` is called right after a 64-byte kernel object is freed, the resulting `struct msg_msg` may be placed on the same memory address. Furthermore, to counter the constant frees and allocations of the memory by the kernel, the system call may be called several times at once to improve the chance of a successful heap spray.

7.3.2 `struct msg_msg` implementation

When the kernel allocates memory for a sent message, it will write the values of a `struct msg_msg` immediately followed by the message in memory as long as the message is equal or smaller in size than constant DATALEN_MSG defined in source code file ipc/msgutil.c:42[1]. When a message exceeds DATALEN_MSG bytes in size the remaining message chunks are stored in a list of message segments whose starting address is saved in the struct msg_msgseg *next field of the message object, while field size_t m_ts stores the entire message length.

Figure 7.2 demonstrates the structure of the msg_msg struct kernel object:

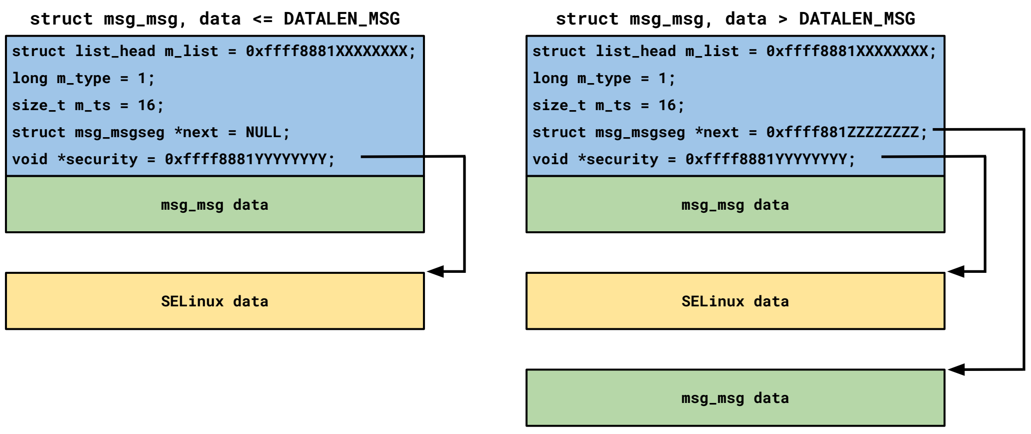

struct msg_msg, data <= DATALEN_MSG
struct list_head m_list = 0xffff8881XXXXXXXX;
long m_type = 1;
size_t m_ts = 16;
struct msg_msgseg *next = NULL;
void *security = 0xffff8881YYYYYYYY;
msg_msg data
SELinux data
struct msg_msg, data > DATALEN_MSG
struct list_head m_list = 0xffff8881XXXXXXXX;
long m_type = 1;
size_t m_ts = 16;
struct msg_msgseg *next = 0xffff881ZZZZZZZZ;
void *security = 0xffff8881YYYYYYYY;
msg_msg data
SELinux data
msg_msg data

Figure 7.2: struct msg_msg implementation

To check whether the heap spraying is successful, GDB may be used to probe memory: if the heap spraying is performed by calling msgsnd() with a message string composed of 15 consecutive 'C' characters (16 bytes with the NULL terminator) then when probing the memory address where the freed object was (which was leaked in log file /dev/kmsg, register RCX) the last 16 values should correspond to 15 consecutive character 'C's ASCII values, which is 0x47, followed by the NULL terminator (0x00).

The following (see Figure 7.3) is the output of kernel memory probing in GDB, where msgsnd() heap spraying has been successful, as can be seen in the last two rows.

[1]https://elixir.bootlin.com/linux/v5.10.11/source/ipc/msgutil.c#L42

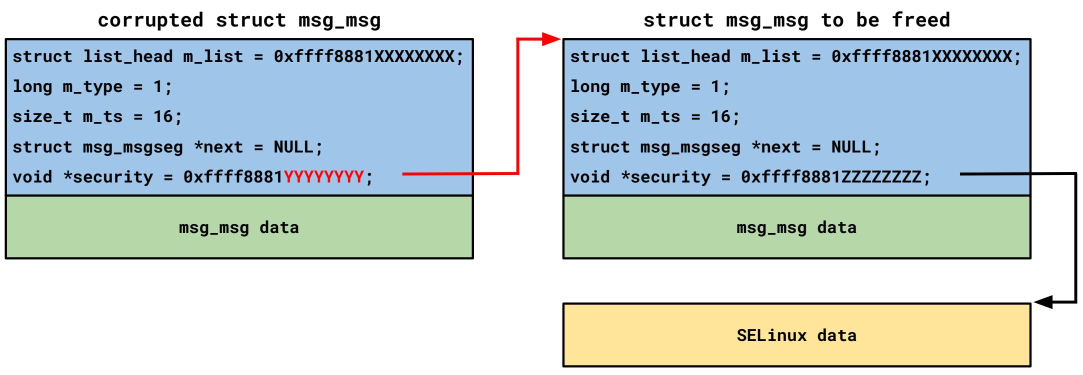
```
(gdb) x/64bx 0xffff97d18dec3340
0xffff97d18dec3340: 0xc0 0xc2 0x97 0x8d 0xd1 0x97 0xff 0xff
0xffff97d18dec3348: 0xc0 0xc2 0x97 0x8d 0xd1 0x97 0xff 0xff
0xffff97d18dec3350: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffff97d18dec3358: 0x10 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffff97d18dec3360: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffff97d18dec3368: 0x80 0xec 0xc9 0x86 0xd1 0x97 0xff 0xff
0xffff97d18dec3370: 0x47 0x47 0x47 0x47 0x47 0x47 0x47 0x47
0xffff97d18dec3378: 0x47 0x47 0x47 0x47 0x47 0x47 0x47 0x00
```

Figure 7.3: Probing memory with GDB after successful `msgsnd()` heap spraying

7.4 Arbitrary Free

An arbitrary free exploit refers to freeing a chosen kernel space memory address without restrictions, meaning that any address, from user space or not, can be freed with it.

If heap spraying is performed with `msgsnd()` after the vulnerability is triggered and a `struct msg_msg` is successfully placed where `struct virtio_vsock_sock` was previously allocated, then the first 4 bytes of the `security` field of the struct, which in a little-endian system correspond to the 32 least-significant-bits of its value, will be overwritten (see Figure 7.4). If the corruption of `msg_msg.security` field happens during `msgsnd()` handling, a SELinux security check will fail, resulting in the freeing of the memory address pointed by `msg_msg.security`. Therefore, an arbitrary free may be achieved when overwriting the 32 least significant bits of the security field.



corrupted struct msg_msg
struct list_head m_list = 0xffff8881XXXXXXXX;
long m_type = 1;
size_t m_ts = 16;
struct msg_msgseg *next = NULL;
void *security = 0xffff8881YYYYYYYY;
msg_msg data
struct msg_msg to be freed
struct list_head m_list = 0xffff8881XXXXXXXX;
long m_type = 1;
size_t m_ts = 16;
struct msg_msgseg *next = NULL;
void *security = 0xffff8881ZZZZZZZZ;
msg_msg data
SELinux data

Figure 7.4: Arbitrary free by corrupting `msg_msg` security field

7.5 Arbitrary Read

An arbitrary read exploit refers to reading a chosen memory address without any restriction, meaning that any memory location from kernel space or user space may be read with it without any user authentication or permission check.

The previously presented arbitrary free can be escalated into an arbitrary read exploit by using the same system call used for the heap spraying: `msgsnd()`. If a kernel allocated `struct msg_msg` is arbitrarily freed and replaced with a user-controlled `struct msg_msg` as shown in Figure 7.5, then an arbitrary read can be achieved by writing the address to be read in field `next` and the amount of bytes to be read + `DATALEN_MSG` in `m_ts`.

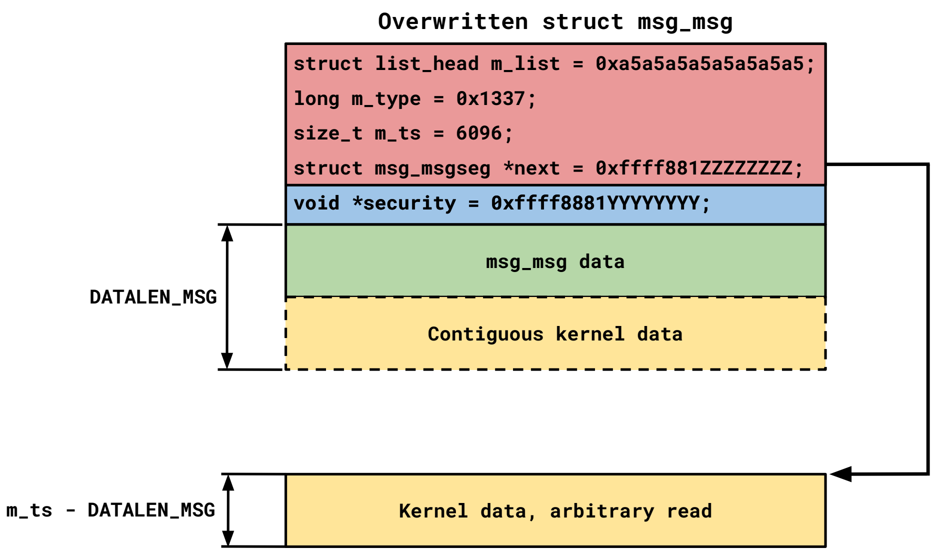

Overwritten struct msg_msg
struct list_head m_list = 0xa5a5a5a5a5a5a5a5;
long m_type = 0x1337;
size_t m_ts = 6096;
struct msg_msgseg *next = 0xffff881ZZZZZZZZ;
void *security = 0xffff8881YYYYYYYY;
msg_msg data
Contiguous kernel data
DATALEN_MSG
m_ts - DATALEN_MSG
Kernel data, arbitrary read

Figure 7.5: Overwritten `msg_msg` kernel object

In order to get the address of a `struct msg_msg` to be freed, the previously explained race condition followed by `msgsnd()` heap spraying will be used. If the spraying is successful, the address of the newly allocated `msg_msg` can be parsed from log file `/dev/kmsg` in the CPU's RCX register.

Once this address is freed, its contents will be replaced with a user-crafted `msg_msg` using a different heap spraying technique that combines the `setxattr()` and `userfaultfd()` system calls.

7.5.1 Heap spraying: `setxattr()` & `userfaultfd()`

This heap spraying technique was first shown by security researcher Vitaly Nikolenko [Nikolenko, 2018], who described it as a universal heap spraying technique. This technique allows an attacker to allocate object contents controlled by the user without any header whatsoever, unlike `msgsnd()` heap spraying, which always allocates a `struct msg_msg` before the message itself.

Kernel function `setxattr()` (see its definition in Listing 7.1) is used to place user-controlled data into kernel memory. However, on a normal execution flow that user data is allocated (lines 9 & 13) and freed (line 19) on the same call.

```
1  static long setxattr(struct dentry *d, const char __user *name,
       const void __user *value, size_t size, int flags)
2  {
3      int error; void *kvalue = NULL;
4      char kname[XATTR_NAME_MAX + 1];
5      // (...)
6      if (size) {
7          if (size > XATTR_SIZE_MAX)
8              return -E2BIG;
9          kvalue = kvmalloc(size, GFP_KERNEL);
10         if (!kvalue)
11             return -ENOMEM;
12         if (copy_from_user(kvalue, value, size)) {
13             error = -EFAULT;
14             goto out;
15         }
16         // (...)
17     }
18     // (...)
19     kvfree(kvalue);
20     return error;
21 }
```

Listing 7.1: Kernel function `setxattr()`'s definition

System call `userfaultfd()` is used to keep that data in kernel memory as long as needed to achieve successful heap spraying. `userfaultfd()` gives the possibility of handling memory page faults in user space at will, meaning that when a read or write is performed on a given page, kernel execution on its calling thread halts until the fault is handled.

The first step into this technique is to allocate two adjacent pages in user space by calling `mmap()` and place the data to be sprayed on the end of the first page. Then, `userfaultfd()` is configured on the second page, so that when `setxattr()` is called with the starting address of the controlled data or payload and a size argument that overflows into the second page, only the data placed in the first page will be copied to the kernel and a page fault will be caused when reaching the second page, effectively halting kernel execution on `setxattr()` and keeping the data from the first page on kernel memory. Figure 7.6 shows the basics of this heap spraying technique.

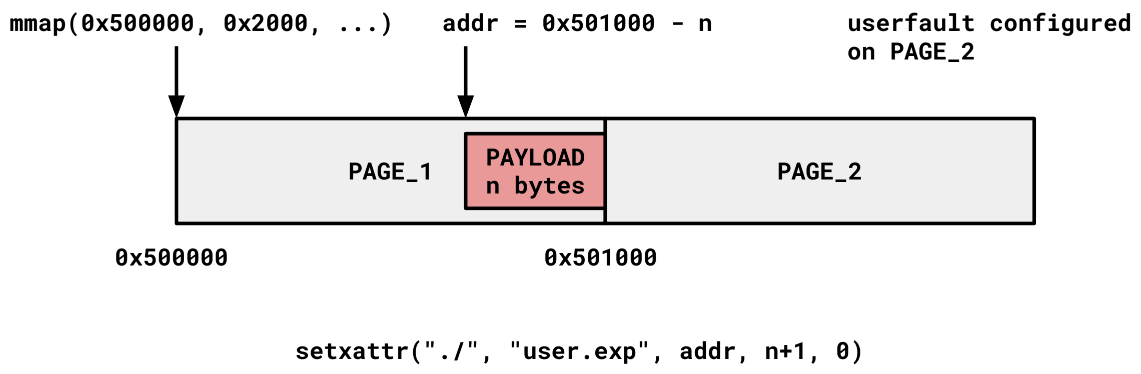

mmap(0x500000, 0x2000, ...)
addr = 0x501000 - n
userfault configured on PAGE_2
PAGE_1
PAYLOAD n bytes
PAGE_2
0x500000
0x501000
setxattr("./", "user.exp", addr, n+1, 0)

Figure 7.6: `setxattr() + userfaultfd()` heap spraying

7.5.2 Receiving the replaced `msg_msg`

Once the original `struct msg_msg` has been freed and replaced with an overwritten one by using `setxattr()` & `userfaultfd()` spraying, the last step into the arbitrary read is to simply receive that message using system call `msgrcv()`, the counterpart to `msgsnd()`. The return value will be `DATALEN_MSG` bytes contiguous to the message's `struct msg_msg` immediately followed by `m_ts - DATALEN_MSG` bytes located in address `next`, as seen in Figure 7.5.

For this privilege escalation exploit, Linux kernel structure `vsock_sock` will be the objective of the arbitrary read. This structure contains three kernel addresses of other kernel structures that will be needed later on, namely:

- struct mem_cgroup *sk_memcg, points to a structure residing in kmalloc-4k slab cache which will be used to calculate the possible address of a struct sk_buff, a socket buffer object.
- const struct cred *owner, stores the credentials or access privileges of the socket owner that will be overwritten to gain root access.
- void (*sk_write_space)(struct sock *), function pointer that is set to the address of static kernel function sock_def_write_space() which will be used to calculate the KASLR offset.

The kernel address of struct vsock_sock is leaked in log file /dev/kmsg in CPU register RBX every time the vsock race condition is caused.

7.6 Privilege escalation

The last step of the exploit is to overwrite the uid and gid fields of the previously leaked const struct cred *owner to effectively escalate privileges. An instance of struct sk_buff, which represents a network buffer, will be replaced and used with the previous arbitrary free and heap spraying techniques.

7.6.1 Control flow hijacking

A network related buffer is represented with struct sk_buff. This object contains an instance of struct skb_shared_info which contains field void* destructor_arg within; this address points to an instance of struct ubuf_info that holds a function pointer, void* callback, which will be replaced and forced to be called to hijack the kernel's control flow.

In order to generate a suitable struct sk_buff in kernel UDP packets of 2800 bytes each have to be sent to local sockets. Sending these UDP packets will generate struct sk_buff kernel objects in the kmalloc-4k slab cache which may end up placing one of them immediately after the leaked struct mem_cgroup *sk_memcg, that is, in address sk_memcg + 4096 due to the fact that struct mem_cgroup *sk_memcg is also placed in the kmalloc-4k slab cache.

By making use of the previously presented arbitrary free exploit followed by `setxattr()` + `userfaultfd()` heap spraying, the `struct sk_buff` immediately after the leaked `sk_memcg` can be overwritten with a payload (see Figure 7.7); when receiving the UDP packet corresponding to that `struct sk_buff` with `recv()` the kernel will execute the function contained in `struct ubuf_info`'s field `callback`, effectively allowing for redirection of the kernel's execution to an arbitrary location.

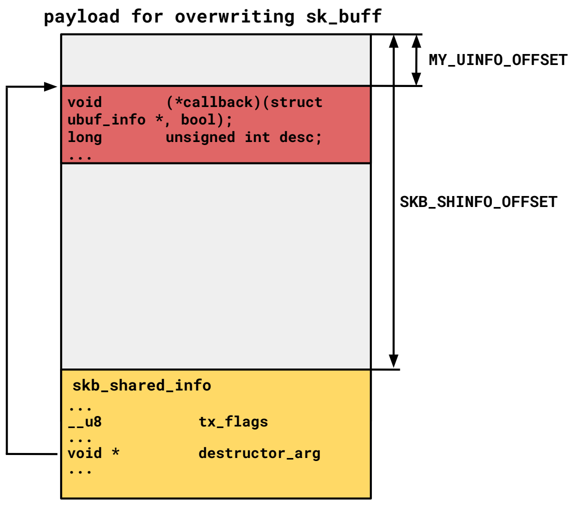


Figure 7.7: Payload for overwriting `sk_buff` object

7.6.2 Arbitrary write

One of the easiest ways to get root privileges at this point would be to redirect the execution flow to a user space function that executed the following two functions (see Listing 7.2):

```
commit_creds(prepare_creds());
```

Listing 7.2: Privilege escalation without SMAP & SMEP

This would immediately elevate privileges to root when executed in kernel space, which was a very common technique for privilege escalation. However, when SMAP and SMEP are enabled, as is the case, this is not possible as execution is halted immediately when such access to user space is detected.

Instead, a ROP gadget that will overwrite the corresponding credentials kernel object will be used for this exploit; a Return-Oriented-Programming (ROP) gadget is a small sequence of CPU instructions present on a binary file that perform several instructions like modifying register values or reading/writing to memory before ending with a '`ret`' operation, a 'return from procedure' instruction to return the control to the calling function.

In order to overwrite the `uid` and `gid` fields of the previously leaked `struct cred *owner` the `callback` function pointer will store the address of a ROP gadget present on the kernel executable (vmlinux or vmlinuz if compressed).

The following Listing 7.3 shows the ROP gadget[2] that will be used to escalate privileges:

```
mov rdx, qword ptr [rdi + 8]
mov qword ptr [rdx + rcx*8], rsi
ret
```

Listing 7.3: ROP gadget for arbitrary write

The first instruction will store in register RDX the 8 bytes (QWORD) of data starting at the address stored in RDI at offset 8 (rdi + 8) as a pointer (ptr), then the second opcode will copy the 8 bytes of value of register RSI to the address stored in register RDX at offset RCX*8 and finally return to the previous execution flow.

When the ROP gadget in `callback` is called the RDI CPU register holds the first argument of the callback function which is the `struct ubuf_info` itself, so the address [rdi + 8] will point to `ubuf_info.desc`, which we set it to be the address of the `uid` field of the leaked credentials object minus 1 byte in the payload (see Figure 7.7); the gadget stores this address in register RDX. The value of register RSI is 1, and RCX = 0, so when opcode `mov qword ptr [rdx + rcx*8], rsi` is executed fields `uid` and `gid` of the leaked `cred` object are overwritten with 0's, as shown in Figure 7.8, which correspond to the `root` user account, effectively converting an unprivileged user into root and at which point the root shell is invoked.

[2]Search ROP gadgets automatically with https://github.com/JonathanSalwan/ROPgadget

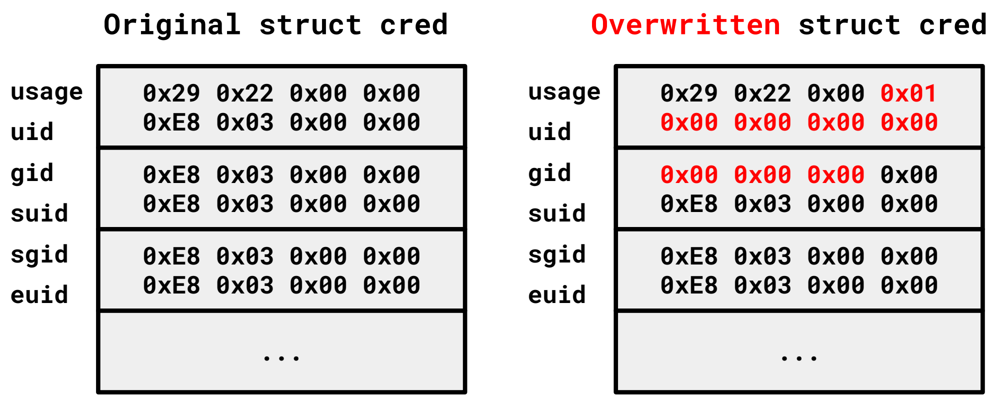


Figure 7.8: Arbitrary write on `struct cred`

By redirecting the execution flow to a kernel space address and not to an user space one SMAP and SMEP are avoided and do not get triggered.

8. CHAPTER

Conclusions

In this project, Linux kernel race condition CVE-2021-26708 was dissected and presented, demonstrating what causes the bug, how to trigger it, and ultimately proving the disastrous consequences it can cause in the wrong hands.

A proof of concept exploit that turned a very limited 4-byte write-after-free into an arbitrary free, read and write of kernel memory, and consequently privilege escalation was implemented using a publicly available technique, bypassing security measures KASLR, SMEP, and SMAP.

The exploit is very target-specific, mostly due to the KASLR calculating scheme and the use of a ROP gadget. Therefore, it requires previous research on the target machine to find the addresses of the ROP gadget and the static kernel function `sock_def_write_space()` (used to calculate the KASLR offset), and to adapt the exploit to it. Moreover, for the exploit to be successful it is necessary that the target machine meets certain requirements that not all popular Linux distributions do; Fedora 33 Server is one of the most popular that meets them all.

Implementing the exploit was quite a challenge, as even though some source codes of different Linux exploits are available online, each exploit relies on different techniques, and they all have their quirks, hardly making any difference in easing the implementation of this exploit. Moreover, kernel crashes were very frequent during development which forced to reboot the virtual machine and set up the working environment over and over again, which slowed down progress greatly.

Furthermore, it was also very frequent to see the System V message queues fill when heap spraying the memory with `msgsnd()`, which also forced to reboot the system, as further heap spraying with `msgsnd()` was not possible and receiving the messages or deleting the queues would make the kernel crash.

Lastly, the exploit may cause several situations beyond the user's control that may cause the exploit to fail, and the kernel to completely freeze, namely:

- **Failed msgsnd() heap spray:** After a `virtio_vsock_sock` object is freed due to the vulnerability's race condition, its memory location is heap sprayed with `msgsnd()`. If this spray fails to replace the freed `virtio_vsock_sock` object, the next step in which this message is arbitrarily freed may free an unrelated kernel object, causing kernel instability.
- **Failed setxattr() + userfaultfd() msg_msg heap spray:** After a real `msg_msg` memory location is arbitrarily freed, its contents are replaced with a fake `msg_msg` using heap spraying. If this heap spraying is unsuccessful, system call `msgrcv()` will fail, causing the exploit to hang or freeze the kernel.
- **Unexpected sk_buff memory location**: After heap spraying with UDP packets, the possible memory address of a `sk_buff` kernel object is calculated as `sk_memcg` location + 4096 bytes. However, it is possible that now `sk_buff` it will be placed on that address, meaning that no control flow hijacking is performed, and kernel instability may be caused.

This investigation work has made me understand better the inner workings of some of the subsystems that compose the Linux kernel, as well as acquiring the necessary skills to develop a kernel exploit from the ground up.

9. CHAPTER

Future work

The exploit developed on this project grants root access to an unprivileged user as long as certain conditions are met. Several situations beyond the user's control may cause the exploit to fail and possibly cause the kernel to become unstable or even completely crash or freeze.

These situations occur commonly, and therefore the exploit must be executed, and the kernel rebooted several times for it to be successful. Further research is needed to determine the exact reasons behind these problems in order to improve the exploit's reliability and success rate.

Bibliography

[CWE, 2020a] CWE (2020a). CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.

[CWE, 2020b] CWE (2020b). Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/data/definitions/1350.html.

[Dio, 2020] Dio, A. D. (2020). The slab allocator in the linux kernel. https://hammertux.github.io/slab-allocator.

[Edge, 2013] Edge, J. (2013). Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[Fabretti, 2018] Fabretti, N. (2018). Cve-2017-11176: A step-by-step linux kernel exploitation. https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html.

[Garzarella, 2020] Garzarella, S. (2020). VSOCK: VM $\leftrightarrow$ host socket with minimal configuration. https://static.sched.com/hosted_files/devconfcz2020a/b1/DevConf.CZ_2020_vsock_v1.1.pdf.

[Help Net Security, 2021] Help Net Security (2021). Now-fixed Linux kernel vulnerabilities enabled local privilege escalation (CVE-2021-26708). https://www.helpnetsecurity.com/2021/03/03/cve-2021-26708/.

[Intel Corporation, 2015] Intel Corporation (2015). Intel® xeon® processor d product family technical overview: Supervisor mode access protection (smap). https://web.archive.org/web/20160411033318/https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview.

[Intel Corporation, 2018] Intel Corporation (2018). Related intel security features & technologies. https://software.intel.com/security-software-guidance/best-practices/related-intel-security-features-technologies.

[Jimenez, 2017] Jimenez, J. (2017). Linux heap exploitation intro series: Used and abused – use after free. https://sensepost.com/blog/2017/linux-heap-exploitation-intro-series-used-and-abused-use-after-free/.

[King, 2013] King, A. (2013). Vsock: Introduce vm sockets. https://github.com/torvalds/linux/commit/d021c344051af91f42c5ba9fdedc176740cbd238.

[Mordechai Guri, 2015] Mordechai Guri, P. (2015). Aslr - what it is, and what it isn't. https://blog.morphisec.com/aslr-what-it-is-and-what-it-isnt/.

[Nikolenko, 2018] Nikolenko, V. (2018). Linux kernel universal heap spray. https://duasynt.com/blog/linux-kernel-heap-spray.

[NVD, 2021] NVD (2021). Cve-2021-26708 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-26708.

[Popov, 2021a] Popov, A. (2021a). Four Bytes of Power: exploiting CVE-2021-26708 in the Linux kernel. https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.

[Popov, 2021b] Popov, A. (2021b). vsock: fix the race conditions in multi-transport support. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c518adafa39f37858697ac9309c6cf1805581446.

[Red Hat, Inc., 2021] Red Hat, Inc. (2021). What is SELinux? https://www.redhat.com/en/topics/linux/what-is-selinux.

[The kernel development community, 2008a] The kernel development community (2008a). Physical page allocation. https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[The kernel development community, 2008b] The kernel development community (2008b). Slab allocator. https://www.kernel.org/doc/gorman/html/understand/understand011.html.

[The kernel development community, 2018] The kernel development community (2018). The linux kernel user's and administrator's guide, memory management. https://www.kernel.org/doc/html/latest/admin-guide/mm/index.html.

[Wessel, 2010] Wessel, J. (2010). Using kgdb, kdb and the kernel debugger internals. https://www.kernel.org/doc/html/v4.15/dev-tools/kgdb.html.

[Wikipedia, 2021] Wikipedia (2021). Supervisor mode access prevention. https://en.wikipedia.org/wiki/Supervisor_Mode_Access_Prevention.