Reverse Engineering Challenge Writeup: Tiempo Oculto

Authors: Abinash Bastola | Hakimuddin Lokhandwala

September 14, 2025

1 Introduction

The challenge consists of a Linux executable that prompts the user to enter a key. If the input matches the expected key, the program accepts the solution and indicates success. The goal of this write-up is to analyze the binary and recover the correct key. The analysis was performed exclusively using Ghidra.

The original CrackMe challenge can be found here: Tiempo Oculto

2 Challenge Description

"This CrackMe is a small program written in C that tests basic analysis and reverse engineering skills. The challenge consists of discovering or deducing the randomly generated secret key that activates the program. Unlike many static challenges, this CrackMe generates a unique key on each execution, requiring the analyst to study the code and understand its logic to obtain the correct solution." — snugpadd

3 Analysis

3.1 Decompile Binary

The compiled file is called, `crackme`.

```
undefined8 main(void)
{
  bool bVar1;
  int iVar2;
  time_t tVar3;
  char *pcVar4;
  size_t sVar5;
  undefined8 uVar6;
  long in_FS_OFFSET;
  int local_b4;
  char local_a8 [48];
  char local_78 [104];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_b4 = 0;
  tVar3 = time((time_t *)0x0);
  srand((uint)tVar3);
  iVar2 = rand();
  generar_clave(local_a8,0x20);
  signal(0xe,manejador_tiempo);
  alarm(iVar2 % 5 + 4);
  while( true ) {
    printf(" %s[ INF ]%s   Introduce la clave de activacion: ",&
       DAT_00102066,&DAT_0010200d);
    bVar1 = false;
    pcVar4 = fgets(local_78,100,stdin);
    if (pcVar4 == (char *)0x0) {
      printf("\n %s[ ERR ]%s   Error leyendo entrada\n",&
         DAT_00102012,&DAT_0010200d);
      uVar6 = 1;
      goto LAB_00101656;
    }
    if ((local_78[0] == '\n') || (local_78[0] == '\0')) {
      builtin_strncpy(local_78,"none",5);
      bVar1 = true;
    }
    else {
      sVar5 = strcspn(local_78,"\n");
      local_78[sVar5] = '\0';
    }
    iVar2 = comprueba_clave(local_78,local_a8);
    if (iVar2 != 0) break;
    local_b4 = local_b4 + 1;
    if (local_b4 < 2) {
      if (bVar1) {
        printf(&DAT_00102138,&DAT_00102129,&DAT_0010200d);
      }
      else {
        printf(" %s[ ERR ]%s   Clave incorrecta\n",&DAT_00102012,&
           DAT_0010200d);
      }
    }
    if (1 < local_b4) {
      printf(&DAT_00102188,&DAT_00102012,&DAT_0010200d);
      uVar6 = 1;
LAB_00101656:
      if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return uVar6;
    }
  }
  printf(" %s[ OKK ]%s   Producto activado correctamente\n",&
     DAT_001020cf,&DAT_0010200d);
  puts("ASOC{l4_cl4v3_3st4_3n_3l_c0d1g0}");
  uVar6 = 0;
  goto LAB_00101656;
}
```

The code above is the Ghidra-decompiled version of the compiled executable. Throughout this analysis, we will ignore the variable `local_10` and the function `__stack_chk_fail()` since they are only used to prevent stack smashing and buffer overflow attacks.

3.2 Identify Key Variables and Function Calls

- `local_a8[48]`: A character array used to hold the generated key.
- `local_78[104]`: A character array used to hold the input provided by the user.
- `ivar2 = rand()`: This call discards the first random value, ensuring that only subsequent values are used to generate the key.
- `tVar3 = time((time_t *)0x0)`: Equivalent to `time(NULL)` and returns the current time. It is often used with `srand()` to seed the random number generator.
- `srand((unsigned int)tVar3)`: Seeds the pseudo-random number generator using the current time as the seed. Since the current time constantly changes, this ensures different random sequences on each program run.
- `generar_clave(local_a8, 0x20)`: Generates a key that is accepted by the program. The first argument is the array that will hold the key, and the second argument specifies the key length, which is 32 bytes.
- `comprueba_clave(local_78, local_a8)`: Compares the user-provided input key with the valid key generated by the program.

3.3 Step Into Key Generator Function

```
void generar_clave(char *param_1,int param_2) {
  int iVar1;
  long in_FS_OFFSET;
  int local_60;
  char local_58 [72];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  builtin_strncpy(local_58,"
     ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
     ,0x3f) ;
  for (local_60 = 0; local_60 < param_2; local_60 = local_60 + 1) {
    iVar1 = rand();
    param_1[local_60] = local_58[iVar1 % 0x3e];
  }
  param_1[param_2] = '\0';
15    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
16                      /* WARNING: Subroutine does not return */
17      __stack_chk_fail();
18    }
19    return;
20  }
```

At line 9, `local_58` will hold the alphanumeric string that contains all uppercase and lowercase letters, as well as the digits 0-9, and it ends with a null terminator `\0` because `strncpy` automatically appends it.

At line 10, we loop from 0 to `param_2`, which is the length of the key that the function will generate. In this case, the function call is `generar_clave(local_a8, 0x20)`.

At line 11, we create a random number using `rand()`, which was seeded with the current time so that the generated sequence changes each second.

At line 12, each index of the key, starting from 0, is assigned an alphanumeric character from the range 0–61. The modulus operator (%) ensures that the random index stays within this range.

3.4 Step Into Key Compare Function

```
bool comprueba_clave(char *param_1,char *param_2)
{
  int iVar1;

  iVar1 = strcmp(param_1,param_2);
  return iVar1 == 0;
}
```

The `comprueba_clave` function takes two `char` pointers. In C, arrays decay to the address of their first element. This function compares the random key generated by the program with the user-provided key.

4 Solution Code

The C language is used here because of its rapid runtime speed, thanks to optimized assembly, and its fast I/O operations.

4.1 Key Generator (keygen.c)

```
// Key Generator Code | keygen.c | keygen

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

void generar_clave(char *key,int keySize) {
  srand(time(NULL));
  rand();

  FILE *keyFile = fopen("key.txt", "w+");

  int iVar1;
  int loopCounter;
  char allAlphaNum [72];

  strncpy(allAlphaNum,"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789", 63);
  for (loopCounter = 0; loopCounter < keySize; loopCounter = loopCounter + 1) {
    iVar1 = rand();
    key[loopCounter] = allAlphaNum[iVar1 % 62];
  }

  key[keySize] = '\0';

  fprintf(keyFile, "%s", key);

  fclose(keyFile);
}
```

The file is then compiled as `keygen` executable.

The most important part of this code is the first call to rand(). This call must be made; otherwise, the generated key will be incorrect.

4.2 Runner Code (main.c)

```
// Key Runner Code | main.c | runner

#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t keyFork = fork();

    if (keyFork == 0) {
        char *args[] = {"./keygen", NULL};
        execv("./keygen", args);
    }

    pid_t crackmeFork = fork();

    if (crackmeFork == 0) {
        FILE *inputFile = fopen("key.txt", "r");
        dup2(fileno(inputFile), STDIN_FILENO);
        char *args[] = {"./crackme", NULL};
        execv("./crackme", args);
    }

}
```

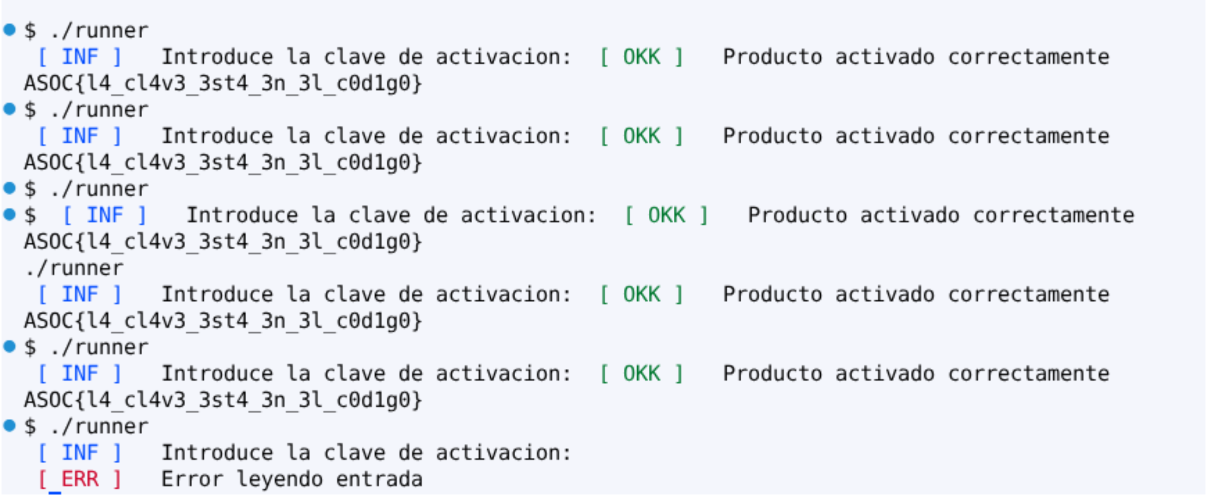
The code above is compiled into the executable `runner`.

The compiled program first forks a process, creating a child. This child process is immediately replaced by the `keygen` executable, which writes a key to a file. The key is only valid for one second. Next, the code forks another child process, which reads from the key file. In this child, the file is treated as stdin; instead of reading from the terminal, input comes from the file. The child then execs itself with the crackme executable, and the exec inherits the redirected stdin. All of this happens very quickly, thanks to the speed of a compiled language.

4.3 Compilation Steps

```
gcc keygen.c -o keygen
gcc main.c -o runner
```

4.4 Test Solution Code



```
● $ ./runner
   [ INF ]   Introduce la clave de activacion:  [ OKK ]   Producto activado correctamente
  ASOC{l4_cl4v3_3st4_3n_3l_c0d1g0}
● $ ./runner
   [ INF ]   Introduce la clave de activacion:  [ OKK ]   Producto activado correctamente
  ASOC{l4_cl4v3_3st4_3n_3l_c0d1g0}
● $ ./runner
● $  [ INF ]   Introduce la clave de activacion:  [ OKK ]   Producto activado correctamente
  ASOC{l4_cl4v3_3st4_3n_3l_c0d1g0}
  ./runner
   [ INF ]   Introduce la clave de activacion:  [ OKK ]   Producto activado correctamente
  ASOC{l4_cl4v3_3st4_3n_3l_c0d1g0}
● $ ./runner
   [ INF ]   Introduce la clave de activacion:  [ OKK ]   Producto activado correctamente
  ASOC{l4_cl4v3_3st4_3n_3l_c0d1g0}
● $ ./runner
   [ INF ]   Introduce la clave de activacion:
   [ ERR ]   Error leyendo entrada
```

Figure 1: The `runner` program produces a key that is accepted, but the authentication can fail if the key expires before it is used.

4.5 Directory Tree

```
.
├── crackme
├── key.txt
├── keygen
├── keygen.c
├── main.c
└── runner
```

5 Solution

The program does not use a fixed key. Instead, it generates a time-dependent key that changes every second. To successfully authenticate, you must determine the correct key for the current second and pass it to the program before it expires. Once the second has passed, the key becomes invalid and a new one is generated.