Guía de Seguridad de las TIC
CCN-STIC 483

Seguridad en Binarios GNU/Linux



Noviembre 2026

MINISTERIO
DE DEFENSA

cpage.mpr.gob.es

Catálogo de Publicaciones de la Administración General del Estado
cpage.mpr.gob.es

Edita:

MINISTERIO DE DEFENSA
SUBSECRETARÍA DE DEFENSA
SECRETARÍA GENERAL TÉCNICA

CCN
centro criptológico nacional

Pº de la Castellana 109, 28046 Madrid
© Centro Criptológico Nacional, 2026
NIPO (edición en línea)

CCN
c=ES, l=MADRID, o=CCN,
ou=SELLO ELECTRONICO,
2.5.4.97=VATES-S2800155J,
serialNumber=S2800155J,
cn=CCN
2026.11.23 11:35:53 +01:00

LIMITACIÓN DE RESPONSABILIDAD
El presente documento se proporciona de acuerdo con los términos en él recogidos, rechazando expresamente cualquier tipo de garantía implícita que se pueda encontrar relacionada. En ningún caso, el Centro Criptológico Nacional puede ser considerado responsable del daño directo, indirecto, fortuito o extraordinario derivado de la utilización de la información y software que se indican incluso cuando se advierta de tal posibilidad.

AVISO LEGAL
Quedan rigurosamente prohibidas, sin la autorización escrita del Centro Criptológico Nacional, bajo las sanciones establecidas en las leyes, la reproducción parcial o total de este documento por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares del mismo mediante alquiler o préstamo públicos.

Esta publicación se distribuye bajo licencia CC BY-NC-ND 4.0 que permite compartir el material en cualquier medio o formato con la condición de reconocer adecuadamente la procedencia «Edita: Ministerio de Defensa. Secretaría General Técnica.» No se puede modificar. No se puede utilizar con fines comerciales.

ÍNDICE

1. INTRODUCCIÓN 1

2. DETECCIÓN DE PROTECCIONES 2

2.1 STACK CANARY 2

2.1.1 TÉCNICA DE DETECCIÓN 2

2.1.2 HERRAMIENTAS PARA LA DETECCIÓN AUTOMÁTICA 3

2.2 PIE 3

2.2.1 TÉCNICA DE DETECCIÓN 3

2.2.2 HERRAMIENTAS PARA LA DETECCIÓN AUTOMÁTICA 5

2.3 NX BIT 6

2.3.1 TÉCNICA DE DETECCIÓN 6

2.3.2 HERRAMIENTAS PARA LA DETECCIÓN AUTOMÁTICA 6

2.4 RELRO 6

2.4.1 TÉCNICA DE DETECCIÓN 7

2.4.2 HERRAMIENTAS PARA LA DETECCIÓN AUTOMÁTICA 8

2.5 FORTIFY_SOURCE 8

2.5.1 TÉCNICA DE DETECCIÓN 8

2.5.2 HERRAMIENTAS PARA LA DETECCIÓN AUTOMÁTICA 10

2.6 CLANG CFI 10

2.6.1 TÉCNICA DE DETECCIÓN 11

2.6.2 HERRAMIENTAS PARA LA DETECCIÓN AUTOMÁTICA 12

2.7 EMPAQUETADO/OFUSCACIÓN 13

2.7.1 TÉCNICA DE DETECCIÓN 13

2.7.2 HERRAMIENTAS PARA LA DETECCIÓN AUTOMÁTICA 15

3. APLICACIÓN DE MEDIDAS DE PROTECCIÓN 16

3.1 APLICACIÓN DE TODAS LAS MEDIDAS DE SEGURIDAD EN LOS BINARIOS 16

3.2 APLICACIÓN DE MEDIDAS DE SEGURIDAD EN EL SISTEMA OPERATIVO 17

4. DESPLIEGUE DE HERRAMIENTAS DE PROTECCIÓN..... 18

4.1 CHECKSEC 18

4.2 READELF 18

4.3 RADARE2 18

4.4 DETECT IT EASY (DIEC).....18

5. DESPLIEGUE MEDIANTE CONTENEDORES 20

ÍNDICE DE TABLAS

Tabla 1: Herramientas para detección de Stack Canary 3
Tabla 2: Herramientas para detección de PIE 5
Tabla 3: Herramientas para detección de NX..... 6
Tabla 4: Herramientas para detección de RELRO..... 8
Tabla 5: Herramientas para detección de FORTIFY_SOURCE..... 10
Tabla 6: Herramientas para detección de Clang CFI 13
Tabla 7: Herramientas para detección de Empaquetado/Ofuscación..... 15
Tabla 8: Parámetros de activación de protecciones 17

1. INTRODUCCIÓN

Durante el desarrollo del código que generará los binarios pueden añadirse diferentes vulnerabilidades que afectarían a la seguridad de la información que manejan. Las vulnerabilidades pueden ser mitigadas con protecciones que deberán ser activadas durante la compilación de la aplicación. El presente documento muestra cómo detectar las diferentes protecciones en los binarios compilados, así como su activación en tiempo de compilación.

En los ejemplos técnicos tanto la detección de la protección como la activación en tiempo de compilación serán mostrados para binarios ELF, comúnmente utilizados en los sistemas operativos **GNU/Linux**.

Los parámetros de compilación necesarios serán explicados para el compilador **Clang,** y los ejemplos de código en lenguaje de programación **C**. El objetivo será generar los binarios con las medidas de seguridad que serán explicadas en el documento para su mejor compresión.

2. DETECCIÓN DE PROTECCIONES

En los siguientes apartados se abordará las diferentes formas y herramientas que podrán ser utilizadas para detectar las protecciones dentro de los binarios.

2.1 STACK CANARY

Las vulnerabilidades del tipo de desbordamiento de buffer (**Buffer overflow**) pueden afectar a los binarios en tiempo de ejecución. Para poder mitigar la explotación de dicha vulnerabilidad se utiliza el **Stack Canary,** que establece un valor al final de los retornos de las funciones, que posteriormente será utilizado para la verificación en tiempo de ejecución, para así comprobar que no se ha modificado la pila de manera inadecuada.

2.1.1 Técnica de detección

Cuando se compila el código con **Clang** y, mediante los parámetros se habilita el **Stack Canary,** dentro del binario se introduce una función con el nombre *__stack_chk_fail*. El nombre de la función puede ser utilizado para la detección de la protección en el binario ejecutable.

Un ejemplo de su detección sería el siguiente. Tomando el siguiente código de ejemplo:

```
int test(int i) {
    return i;
}

int main(void) {
    int x;
    int i = 10;
    x = test(i);
    return x;
}
```

Se habilitaría la protección con el siguiente parámetro de compilación:

```
clang -fstack-protector-all -o <nombreBinario> <nombreFicheroCodigo>
```

Al leer los datos del binario se puede apreciar que aparece en dos ocasiones la función *__stack_chk_fail*.

```
objdump -dj .text <nombreBinario> | grep "__stack_chk_fail"
1165: e8 c6 fe ff ff            call   1030 <__stack_chk_fail@plt>
```

```
11a9: e8 82 fe ff ff            call   1030 <__stack_chk_fail@plt>
```

Por lo tanto, para binarios compilados con GCC es posible buscar la cadena de texto *__stack_chk_fail* dentro de las referencias a las funciones del código.

2.1.2 Herramientas para la detección automática

A continuación, se muestran en una tabla las herramientas que pueden ser utilizadas para la detección de la protección en los binarios compilados, así como los comandos de ejemplo para utilizarlas:

Herramienta	Comando
radare2	rabin2 -I <nombreBinario> | grep canary
checksec	checksec –file=<nombreBinario>
readelf	readelf -s <nombreBinario> | grep __stack_chk_fail

Tabla 1: Herramientas para detección de Stack Canary

2.2 PIE

Durante la ejecución de los binarios se asigna un espacio en memoria para ser utilizado por el proceso, la asignación se realiza de la misma forma en cada ejecución de forma que es fácilmente previsible y, en ataques dirigidos a la memoria, facilitan la explotación. Para añadir complejidad a una posible explotación existe la asignación aleatoria de memoria como medida de protección, esta protección es llamada **PIE** (**Position-Independent Executable**).

2.2.1 Técnica de detección

Cuando se activa la protección en tiempo de compilación, el binario no presenta indicios de tener la protección activa, sin embargo, al deshabilitarlo, el binario contiene un valor de cadena con el texto *_dl_relocate_static_pie* como una función añadida.

Tomando el siguiente código de ejemplo:

```
int test(int i) {
    return i;
}
```

```
int main(void) {
    int x;
    int i = 10;
    x = test(i);
    return x;
}
```

Para la opción con la protección simplemente se puede compilar sin parámetro o con el siguiente parámetro:

```
clang -pie <nombreFicheroCodigo> -o <nombreBinario>
```

Para compilar sin protección se utiliza la siguiente opción:

```
clang -no-pie <nombreFicheroCodigo> -o <nombreBinario>
```

Al realizar la extracción del contenido de las funciones:

```
objdump -dj .text <nombreBinario>
```

Aquí se puede apreciar la función en cuestión:

```
0000000000401050 <_dl_relocate_static_pie>:
  401050:   c3                      ret
  401051:   66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
  401058:   00 00 00
  40105b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

La función *_dl_relocate_static_pie* se utiliza para la compatibilidad entre binarios y librerías donde unos pueden tener o no activada la protección **PIE**, por lo que detectar la cadena siempre indicará que se necesita de dicha compatibilidad y que en algún binario o librería no se está utilizando **PIE**.

Existe otra forma de diferenciar entre binarios con la protección compilada. Se trata del tipo de fichero ejecutable definido en el formato ELF. Para poder encontrar la información es posible utilizar la utilidad **readelf** con la opción -h, en el valor devuelto de *type* se puede observar la diferencia.

```
readelf -h <nombreBinario>
```

La salida muestra el valor *DYN (Position-Independent Executable file):*

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
```

```
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable
file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1040
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13952 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

Cuando el binario no lleva la protección muestra el valor *EXEC (Executable file).*

2.2.2 Herramientas para la detección automática

A continuación, se muestran en una tabla las herramientas que pueden ser utilizadas para la detección de la protección en los binarios compilados, así como los comandos de ejemplo para utilizarlas:

Herramienta	Comando
radare2	rabin2 -I <nombreBinario> | grep pic
checksec	checksec –file=<nombreBinario>
readelf	readelf -a <nombreBinario> | grep “type:” | grep DYN

Tabla 2: Herramientas para detección de PIE

2.3 NX BIT

Los espacios de memoria pueden ser reservados para el almacenamiento de datos o para la ejecución, en el caso de utilizarse para la ejecución es posible facilitar la explotación, por lo tanto, existe una medida de protección para evitar que se ejecute código en espacios de memoria reservados a los datos en sistemas **GNU/Linux**.

2.3.1 Técnica de detección

La protección de **NX bit** se indica mediante un bit dentro del binario.

Para compilar sin la protección es necesario aplicar el siguiente parámetro de GCC:

```
clang -z execstack <nombreFicheroCodigo> -o <nombreBinario>
```

Para compilar con la protección completa sería necesario aplicar el siguiente parámetro:

```
clang -Wl,-z,now <nombreFicheroCodigo> -o <nombreBinario>
```

2.3.2 Herramientas para la detección automática

A continuación, se muestran en una tabla las herramientas que pueden ser utilizadas para la detección de la protección en los binarios compilados, así como los comandos de ejemplo para utilizarlas:

Herramienta	Comando
radare2	rabin2 -I <nombreBinario> | grep nx
checksec	checksec --file=<nombreBinario>
readelf	readelf -l <nombreBinario> | grep GNU_STACK -A1 | grep -v RWE

Tabla 3: Herramientas para detección de NX

2.4 RELRO

La protección **RELRO** (**Relocation Read-Only**) se activa para proteger las secciones del binario que contienen datos, en concreto, las zonas como **GOT** (**Global Offset Table**), estableciéndolas como de solo lectura cuando son cargadas en memoria.

2.4.1 Técnica de detección

Las técnicas de detección difieren dependiendo del nivel de protección que se aplica. La protección de **Relro** tiene tres niveles de protección, parcial, completa o inexistente, por lo que se pueden detectar tres estados distintos.

Como código de ejemplo para las pruebas se utilizará el siguiente:

```
int test(int i) {
    return i;
}

int main(void) {
    int x;
    int i = 10;
    x = test(i);
    return x;
}
```

Para compilar sin la protección se utilizan los siguientes parámetros:

```
clang -z norelro <nombreFicheroCodigo>  -o <nombreBinario>
```

Para una protección parcial de **Relro** se ejecuta con el siguiente parámetro:

```
clang -z relro <nombreFicheroCodigo>  -o <nombreBinario>
```

Para una protección completa de **Relro** se ejecuta con el siguiente parámetro:

```
clang -z relro -Wl,-z,now example.c -o <nombreBinario>
```

Para saber si el **Relro** está activado se buscará en las cabeceras del programa la cadena GNU_RELRO:

```
readelf -l <nombreBinario> | grep GNU_RELRO
```

En este punto el binario podría tener activado el **Relro** parcial o completo, ahora se diferenciará con el siguiente comando que accederá a la sección dinámica del binario:

```
readelf -d <nombreBinario> | grep BIND_NOW
```

En el caso de que la cadena BIN_NOW se encuentre en la sección se habrá detectado un **Relro** completo, en caso contrario será un **Relro** parcial.

2.4.2 Herramientas para la detección automática

A continuación, se muestran en una tabla las herramientas que pueden ser utilizadas para la detección de la protección en los binarios compilados, así como los comandos de ejemplo para utilizarlas:

Herramienta	Comando
radare2	rabin2 -I <nombreBinario> | grep relro
checksec	checksec --file=<nombreBinario>
readelf	readelf -l <nombreBinario> | grep GNU_RELRO readelf -d <nombreBinario> | grep BIND_NOW

Tabla 4: Herramientas para detección de RELRO

2.5 FORTIFY_SOURCE

Existen diferentes vulnerabilidades en tiempo de ejecución que pueden ser incorporadas al código cuando se utilizan funciones inseguras como *strcpy*. Con la protección **FORTIFY SOURCE** activada se sustituirán las funciones potencialmente vulnerables por funciones seguras en tiempo de compilación.

2.5.1 Técnica de detección

Debido a que **FORTIFY_SOURCE** tiene varios niveles de protección la detección podrá estar condicionada a sus estados.

Las diferencias de los niveles de protección son explicadas en el *man* de las *test macros* (https://man7.org/linux/man-pages/man7/feature_test_macros.7.html)

- **Nivel 0**: No se realizan comprobaciones ni en tiempo de compilación ni en ejecución del binario.
- **Nivel 1 (por defecto)**: Las protecciones se aplican en tiempo de compilación. No modifican el binario final.
- **Nivel 2**: Las protecciones se aplican a funciones concretas con comportamientos peligrosos con el buffer que pueden terminar en desbordamiento. Se pueden encontrar mecanismos de la protección en los binarios compilados.

- **Nivel 3**: Aplica la misma protección que el nivel 2 a funciones peligrosas, pero además valida las variables transmitidas a llamadas de ciertas funciones, como la función *malloc*.

Para realizar las pruebas de forma sencilla se puede utilizar el siguiente código:

```
#include <stdio.h>
#include <string.h>

int main () {
   char src[40];
   char dest[12];

   memset(dest, '\0', sizeof(dest));
   strcpy(src, "Test string to copy.");
   strncpy(dest, src, 10);

   printf("Final copied string : %s\n", dest);

   return(0);
}
```

La compilación sin la protección sería una compilación con el parámetro a 0:

```
clang -D_FORTIFY_SOURCE=0 <nombreFicheroCodigo> -o <nombreBinario>
```

Para añadir el primer nivel de protección se utilizó el siguiente parámetro:

```
clang -D_FORTIFY_SOURCE=1 -O1 <nombreFicheroCodigo> -o <nombreBinario>
```

Como se puede observar se añade el parámetro -O1, utilizado para la optimización del código compilado, que conjuntamente con las protecciones con **FORTIFY_SOURCE** realizan una protección más optima.

El siguiente nivel es el 2:

```
clang -D_FORTIFY_SOURCE=2 -O2 <nombreFicheroCodigo> -o <nombreBinario>
```

Y, por último, el más seguro, es el nivel 3, el parámetro de optimización seguirá siendo -O2:

```
clang -D_FORTIFY_SOURCE=3 -O2 <nombreFicheroCodigo> -o <nombreBinario>
```

2.5.2 Herramientas para la detección automática

A continuación, se muestran en una tabla las herramientas que pueden ser utilizadas para la detección de la protección en los binarios compilados, así como los comandos de ejemplo para utilizarlas:

Herramienta	Comando
checksec	checksec --file=<nombreBinario>
readelf	*

Tabla 5: Herramientas para detección de FORTIFY_SOURCE

*: Para la ejecución de **readelf** es necesario capturar la salida y buscar el nombre de las funciones creadas por el proceso de **FORTIFY SOURCE**:

```
readelf -s <nombreBinario> | grep -E
"__memcpy_chk|__memmove_chk|__mempcpy_chk|__memset_chk|__stpcpy_chk|__stpnc
py_chk|__strcat_chk|__strcpy_chk|__strncat_chk|__strncpy_chk|__snprintf_chk
|__sprintf_chk|__vsnprintf_chk|__vsprintf_chk|__fprintf_chk|__printf_chk|__
vfprintf_chk|__vprintf_chk"
```

2.6 CLANG CFI

La integridad del flujo de control busca impedir que un atacante redirija la ejecución hacia destinos no previstos aprovechando corrupción de memoria, por ejemplo, sobrescribiendo punteros a función o *vtables*, o forzando llamadas indirectas hacia código con una firma incompatible. Clang CFI implementa esta protección insertando comprobaciones en tiempo de ejecución antes de ciertas transferencias de control, de forma que una llamada indirecta solo se permite si el destino es coherente con el tipo esperado en ese punto. En la práctica, esto reduce la explotabilidad de muchas primitivas típicas de ROP o de secuestro de punteros, porque, aunque el atacante consiga controlar el valor del puntero, el salto puede ser bloqueado si no coincide con la información de tipo asociada a esa llamada.

Es importante tener en cuenta que CFI no se refleja como un bit simple en cabeceras ELF, y su implementación depende de instrumentación y de artefactos internos del binario. Además, la mayoría de los esquemas de Clang CFI requieren LTO, por lo que la compilación debe incluir -flto o -flto=thin y un enlazador compatible con LTO.

2.6.1 Técnica de detección

El mecanismo de protección se encuentra en diferentes partes del binario, haciendo difícil la detección mediante bits de activación, cadenas de texto o patrones dentro del código, por lo tanto, será necesario la utilización de la herramienta **checksec** en cualquier caso.

Cabe destacar que la protección de **CFI** es soportada únicamente mediante el compilador **clang**.

Para las pruebas se utilizará el siguiente código:

```
#include <stdio.h>
#include <string.h>

#define AUTHMAX 4

struct auth {
 char pass[AUTHMAX];
 void (*func)(struct auth*);
};

void success() {
 printf("Authenticated successfully\n");
}

void failure() {
 printf("Authentication failed\n");
}

void auth(struct auth *a) {
 if (strcmp(a->pass, "pass") == 0)
     a->func = &success;
 else
     a->func = &failure;
}
```

```
int main(int argc, char **argv) {
 struct auth a;

 a.func = &auth;

 printf("Enter your password:\n");
 scanf("%s", &a.pass);

 a.func(&a);
}
```

La compilación con **Clang** por defecto no lleva la protección activada, por lo tanto, este sería el ejemplo sin **CFI**:

```
clang <nombreFicheroCodigo> -o <nombreBinario>
```

Para compilar con la protección se utilizarían los siguientes parámetros:

```
clang <nombreFicheroCodigo> -o <nombreBinario> -flto -fsanitize=cfi -
fvisibility=default
```

En este caso la detección más sencilla se realiza con la herramienta **checksec**:

```
checksec --extended --file=<nombreBinario>
```

Existe una técnica de detección de protecciones mediante la consulta de los parámetros de compilación almacenados en el propio binario. Para que el compilador almacene esta información se debe especificar el parámetro "-frecord-command-line". El comando sería de la siguiente forma:

```
clang -frecord-command-line <nombreFicheroCodigo> -o <nombreBinario> -flto
-fsanitize=cfi -fvisibility=default
```

Sin embargo, esta detección puede ser fácilmente alterada mediante la modificación del binario después de ser compilada, dando lugar a falsos positivos o negativos. Por lo tanto, se desaconseja su uso.

2.6.2 Herramientas para la detección automática

A continuación, se muestran en una tabla las herramientas que pueden ser utilizadas para la detección de la protección en los binarios compilados, así como los comandos de ejemplo para utilizarlas:

Herramienta	Comando
checksec	checksec --extended --file=<nombreBinario>

Tabla 6: Herramientas para detección de Clang CFI

2.7 EMPAQUETADO/OFUSCACIÓN

Los sistemas de empaquetamiento de binarios u ofuscación no protegen contra vulnerabilidades concretas, pero establecen una dificultad o disuasión para realizar ingeniería inversa.

2.7.1 Técnica de detección

2.7.1.1 UPX

La facilidad de la detección del empaquetado mediante **UPX** dependerá de si se han eliminado cabeceras del binario que identifiquen de forma inequívoca que el software utilizado para generar el binario empaquetado es **UPX**.

Para tener el binario de prueba se utiliza el siguiente código:

```
int test(int i) {
    return i;
}

int main(void) {
    int x;
    int i = 10;
    x = test(i);
    return x;
}
```

Se genera el binario de ejemplo con el siguiente comando:

```
clang <nombreFicheroCodigo> -o <nombreBinario>
```

Para empaquetar el binario se ejecuta la herramienta **UPX** de la siguiente forma:

```
upx --best <nombreBinario>.pkg
```

Con el paquete formado se puede apreciar que existen cadenas de texto nombrando el software **UPX**, es posible extraerlas simplemente con el comando **strings**:

```
strings <nombreBinario>.pkg | grep UPX
```

El resultado sería el siguiente:

```
UPX!
$Info: This file is packed with the UPX executable packer http://upx.sf.net
$
$Id: UPX 5.02 Copyright (C) 1996-2025 the UPX Team. All Rights Reserved. $
UPX!
UPX!
```

La detección, aunque válida en este caso, podría arrojar falsos negativos en el caso de que se modificasen las cabeceras en otras versiones del software **UPX** o incluso falsos positivos en los casos en que se encuentren cadenas de texto con el valor buscado, aunque no tengan relación. Por lo tanto, no se recomienda el uso de este método de detección, utilizando en su defecto el ofrecido por herramientas con un proceso de detección que analice la entropía del binario para detectar si está empaquetado u ofuscado.

2.7.1.2 VMProtect

El software de **VMProtect** realiza el empaquetado del binario además de la ofuscación para impedir su lectura.

El código utilizado para generar el binario de ejemplo es el siguiente:

```
int test(int i) {
    return i;
}

int main(void) {
    int x;
    int i = 10;
    x = test(i);
    return x;
}
```

Se compila con las opciones por defecto:

```
clang <nombreFicheroCodigo> -o <nombreBinario>
```

Posteriormente se empaqueta con las opciones por defecto.

```
vmprotect_con <nombreBinario> <nombreBinario>.pkg.vm
```

A simple vista, al contrario que **UPX**, el binario no tiene ningún tipo de cabecera o cadena de texto indicando el uso de la herramienta **VMProtect**. Solamente será posible detectar mediante la medición de la entropía de la información almacenada en el binario.

2.7.2 Herramientas para la detección automática

A continuación, se muestra una tabla con la herramienta que puede ser utilizada para la detección de la ofuscación en los binarios compilados:

Herramienta	Comando
Detect It Easy (DiE)	diec -e <nombreBinario>

Tabla 7: Herramientas para detección de Empaquetado/Ofuscación

3. APLICACIÓN DE MEDIDAS DE PROTECCIÓN

Durante la compilación es posible indicar al compilador que será necesario aplicar medidas que quedarán reflejadas en el código de los propios binarios, protegiendo la memoria, controlado los accesos y la escritura de sus datos.

3.1 APLICACIÓN DE TODAS LAS MEDIDAS DE SEGURIDAD EN LOS BINARIOS

En esta sección se indica cómo aplicar todas las medidas de protección a través de un comando parametrizando.

Para aplicar todas las medidas de seguridad será necesario utilizar el compilador **Clang**, debido a que el compilador **GCC** no tiene la capacidad de aplicar la protección **CFI**.

El comando utilizado para habilitar todas las protecciones es el siguiente:

```
clang -fstack-protector-all -fstack-clash-protection -pie -Wl,-z,relro -
Wl,-z,now  -D_FORTIFY_SOURCE=3 -O2 -flto -fsanitize=cfi -
fvisibility=default <nombreFicheroCodigo> -o <nombreBinario>
```

Los parámetros indican las siguientes opciones.

Parámetro	Descripción
-fstack-protector-all	El parámetro aplica las protecciones de **Stack Canary** contra el **Stack smashing** a todas las funciones. https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-fstack-protector-all
-fstack-clash-protection	Parámetro para habilitar la protección contra los **Stack Clash Attack**. https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-fstack-clash-protection
-pie	Activa la protección **PIE**. https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-pie
-Wl,-z,relro	Establece la protección **RELRO** en el binario.

-Wl,-z,now	Habilita la protección **NX**.
-D_FORTIFY_SOURCE=3 -O2	Define el nivel de protección de la medida **FORTIFY SOURCE**.
-flto -fsanitize=cfi -fvisibility=default	El parámetro habilita la protección **CFI**.

Tabla 8: Parámetros de activación de protecciones

3.2 APLICACIÓN DE MEDIDAS DE SEGURIDAD EN EL SISTEMA OPERATIVO

Aunque la medida de seguridad no se podrá detectar en el binario, es posible, a nivel de sistema operativo aplicar configuraciones en sus ficheros que harán efectivas las protecciones de **ASLR** en el Kernel.

Las configuraciones para aplicar la protección de **ASLR** pueden ser aplicadas de forma temporal o permanente.

Para aplicar de forma temporal se deberá ejecutar el siguiente comando:

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

El valor de "2" hablita la opción más segura para proteger la memoria. Siendo las opciones las siguientes:

- 0 – Sin protección de aleatorización.
- 1 – Aleatorización conservadora. Las bibliotecas compartidas, la pila, mmap(), VDSO y el montón se aleatorizan.
- 2 – Aleatorización completa. Además de los elementos enumerados en el punto anterior, la memoria gestionada a través de brk() también se aleatoriza.

Para establecer de forma permanente la configuración es necesario editar el fichero /etc/sysctl.conf añadiendo o modificando la siguiente línea:

```
kernel.randomize_va_space=2
```

Una vez modificado es necesario activar la nueva configuración ejecutando el siguiente comando:

```
sudo sysctl -p
```

4. DESPLIEGUE DE HERRAMIENTAS DE PROTECCIÓN

Las herramientas que se utilizan para detectar las protecciones activadas en los binarios pueden ser desplegados de forma sencilla en un entorno GNU/Linux. Para los ejemplos se utilizará la distribución Debian versión 12.12.

4.1 CHECKSEC

Para la instalación de **Checksec**, es necesario descargar el binario del repositorio de Github.

```
wget https://github.com/slimm609/checksec/archive/refs/tags/2.6.0.zip
```

Posteriormente se descomprime el paquete descargado:

```
unzip 2.6.0.zip
```

(Opcional) Se mueve el binario a la carpeta para su uso por consola:

```
cp checksec-2.6.0/checksec /usr/local/bin/checksec
```

4.2 READELF

La instalación de la utilidad **Readelf** se realiza mediante el paquete *binutils*. Simplemente será necesario instalarlo mediante *apt*:

```
apt install binutils
```

4.3 RADARE2

Radare2 (**rabin2**) cuenta con paquetes para distribuciones **GNU/Linux**, por lo que será necesario descargar e instalar:

```
wget
https://github.com/radareorg/radare2/releases/download/6.0.4/radare2_6.0.4_
amd64.deb
```

Se instalará con *dpkg*:

```
dpkg -i radare2_6.0.4_amd64.deb
```

4.4 DETECT IT EASY (DIEC)

Para instalar el binario es posible utilizar el paquete para la distribución específica de **GNU/Linux** directamente de su repositorio:

```
wget https://github.com/horsicq/DIE-
engine/releases/download/3.10/die_3.10_Debian_12_amd64.deb
```

Y se instala mediante *apt*:

```
apt install -y ./die_3.10_Debian_12_amd64.deb
```

5. DESPLIEGUE MEDIANTE CONTENEDORES

Para el uso de las herramientas dentro de un contenedor se facilitan los siguientes *dockerfile* que generaran los dos contenedores con las aplicaciones necesarias para el análisis de los binarios.

El contenedor con las aplicaciones de **checksec** y **readelf** se podrá formar con el siguiente *dockerfile*:

```
FROM alpine:latest

RUN apk add --no-cache bash binutils openssl file ca-certificates wget
gcompat jq unzip

RUN wget https://github.com/slimm609/checksec/archive/refs/tags/2.6.0.zip
&& unzip 2.6.0.zip && cp checksec-2.6.0/checksec /usr/local/bin/checksec

RUN wget https://github.com/upx/upx/releases/download/v5.0.2/upx-5.0.2-
amd64_linux.tar.xz && tar -xvf upx-5.0.2-amd64_linux.tar.xz && cp upx-
5.0.2-amd64_linux/upx /usr/local/bin/

ENTRYPOINT ["/bin/bash","-c"]
```

Se construye el contenedor con el siguiente comando de docker:

```
docker build -t general:latest -f Dockerfile
```

El contenedor con **diec** tendrá el siguiente *dockerfile*:

```
FROM debian:12.12-slim

RUN apt update

RUN apt install -y jq wget

RUN wget https://github.com/horsicq/DIE-
engine/releases/download/3.10/die_3.10_Debian_12_amd64.deb

RUN apt install -y ./die_3.10_Debian_12_amd64.deb

ENTRYPOINT ["/bin/bash","-c"]
```

Y se construirá con el siguiente comando:

```
docker build -t diec:latest -f Dockerfile
```

Para Radare2 existe un contenedor listo para utilizarse en Docker hub, con el siguiente comando se descargaría la imagen necesaria:

```
docker pull radare/radare2
```

PUBLICACIONES DE DEFENSA

GOBIERNO DE ESPAÑA
MINISTERIO DE DEFENSA
SUBSECRETARÍA DE DEFENSA
SECRETARÍA GENERAL TÉCNICA
SUBDIRECCIÓN GENERAL DE PUBLICACIONES Y PATRIMONIO CULTURAL