centro criptológico nacional

Guía de Seguridad de las TIC
CCN-STIC 484

Buenas Prácticas de Fuzzing y Ejecución Simbólica



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

1.1 OBJETIVO 1

1.2 ALCANCE 1

1.3 TERMINOLOGÍA Y DEFINICIONES CLAVE 1

2. FUNDAMENTOS 4

2.1 ¿QUÉ ES EL FUZZING? 4

2.2 ¿QUÉ ES LA EJECUCIÓN SIMBÓLICA? 4

3. PREPARACIÓN DEL ENTORNO 5

3.1 SELECCIÓN DEL OBJETIVO (FUNCIÓN, BINARIO O MÓDULO) 5

3.2 ANÁLISIS INICIAL DEL PROGRAMA 5

4. BUENAS PRÁCTICAS PARA FUZZING 6

4.1 SELECCIÓN DEL TIPO DE FUZZER 6

4.1.1 FUZZING CON SOLO BINARIO 6

4.1.2 FUZZING CON ACCESO AL CÓDIGO FUENTE 7

4.2 SANITIZADORES 8

4.2.1 USO DE SANITIZADORES 8

4.2.2 ELECCIÓN DE SANITIZADORES 11

4.3 CONSTRUCCIÓN DEL CORPUS INICIAL 13

4.4 MONITORIZACIÓN Y LOGGING 17

4.5 REPRODUCIBILIDAD DE CRASHES 18

5. BUENAS PRÁCTICAS PARA EJECUCIÓN SIMBÓLICA 21

5.1 SELECCIÓN DEL OBJETIVO 21

5.2 DISEÑO DE RESTRICCIONES EFECTIVAS 21

5.3 INTERPRETACIÓN Y VALIDACIÓN DE RUTAS GENERADAS 23

6. SEGURIDAD Y AISLAMIENTO 26

6.1 RIESGOS DE EJECUTAR FUZZERS Y EJECUCIÓN SIMBÓLICA 26

6.2 LIMITACIONES DE RECURSOS.....28

ÍNDICE DE TABLAS

Tabla 1: Incompatibilidades entre sanitizadores..... 10
Tabla 2: Elección de sanitizador 12

1. INTRODUCCIÓN

1.1 OBJETIVO

El objetivo del documento es proporcionar una guía estructurada y práctica para llevar a cabo campañas de fuzzing y ejecución simbólica de forma segura. En ella se describen los fundamentos de ambas técnicas, recomendaciones de diseño y medidas de seguridad necesarias para minimizar riesgos.

El propósito final es facilitar que diferentes equipos puedan implementar estas metodologías y que estas estén alineadas con las mejores prácticas dentro del sector.

1.2 ALCANCE

El consecuente documento proporciona aspectos conceptuales y operativos relacionados con el fuzzing y la ejecución simbólica:

- Fundamentos técnicos de ambas metodologías.
- Estrategias de preparación del entorno.
- Uso de instrumentación y sanitizadores.
- Diseño de ejecuciones y criterios de calidad.
- Riesgos y medidas de seguridad.

1.3 TERMINOLOGÍA Y DEFINICIONES CLAVE

A lo largo del documento se emplean varios términos que conviene definir previamente para el correcto entendimiento de éste:

- **Fuzzer:** Herramienta que genera y ejecuta entradas de prueba de forma automatizada.
- **Fuzzing:** Proceso que se lleva a cabo desde que se ejecuta un fuzzer hasta que termina.
- **Semilla:** Caso de prueba inicial que sirve de base para mutar de forma aleatoria las entradas.
- **Corpus:** Conjunto de entradas para el proceso de fuzzing.
- **Cobertura (Coverage):** Medida de rutas, funciones o bloques de código ejecutados del programa objetivo durante el proceso de fuzzing.
- **Sanitizador:** Mecanismo de instrumentación que detecta comportamientos anómalos en tiempo de ejecución.
- **Solver:** Componente que resuelve sistemas de ecuaciones generados durante el análisis simbólico.
- **Checksum:** Valor calculado a partir de unos datos para verificar su integridad

- **Parsing:** Proceso por el cual un programa interpreta y descompone datos de entrada.
- **Offset:** Posición específica dentro de un archivo o bloque de memoria donde se encuentra un dato; muchos formatos estructurados se basan en offsets para localizar campos.
- **Timestamps:** Marcas temporales presentes en ciertos formatos o protocolos; su manipulación puede afectar la lógica del programa durante el fuzzing.
- **Crashear:** Producir un fallo inesperado que interrumpe la ejecución del programa; objetivo primario del fuzzing.
- **Runtimes:** Entornos de ejecución que gestionan la memoria, excepciones y recursos; durante el fuzzing pueden influir en la forma en que un error se manifiesta.
- **Buffer overflow/underflow), use-after-free, double free:** Tipos comunes de errores de memoria donde se escribe fuera de límites, se accede a memoria liberada o se libera dos veces la misma zona, generando comportamientos inseguros.
- **Memoria sombra (*shadow memory*):** Técnica usada por sanitizadores como ASan para mantener metadatos sobre el estado de cada byte de memoria y detectar accesos inválidos.
- **Redzones:** Áreas especiales alrededor de variables o estructuras, marcadas como inaccesibles por los sanitizadores, cuyo propósito es detectar desbordamientos o accesos fuera de rango.
- **Overhead:** Coste adicional en rendimiento, memoria o recursos que introduce una técnica (como la instrumentación o los sanitizadores) y que puede ralentizar la ejecución durante el fuzzing.
- **Hooks**: Puntos de inserción donde se intercepta o sustituye el comportamiento de funciones o eventos del programa para monitorizar o modificar su ejecución.
- **Syscalls** (llamadas al sistema): Funciones proporcionadas por el kernel que los programas invocan para interactuar con el sistema operativo; son un objetivo típico en fuzzing de bajo nivel.
- **Inlining**: Optimización del compilador que reemplaza una llamada a función por su cuerpo, reduciendo el coste de llamada pero afectando a la instrumentación del fuzzer o del sanitizador.
- **Backtrace**: Lista de funciones en la pila en el momento de un crash, usada para reconstruir el camino de ejecución que llevó al fallo.
- **Stubs**: Implementaciones mínimas o simuladas usadas para reemplazar funciones reales durante pruebas, útiles para aislar componentes mientras se realiza fuzzing o se depura.

- **Locale:** Conjunto de configuraciones regionales (idioma, codificación, formatos de fecha/número) que puede influir en cómo un programa interpreta y procesa la entrada durante el fuzzing.

2. FUNDAMENTOS

2.1 ¿QUÉ ES EL FUZZING?

El fuzzing es una metodología automatizada que consiste en proporcionar datos inválidos, inesperados o aleatorios como entradas de un programa. Su objetivo principal es supervisar el comportamiento del programa frente a este tipo de entradas en busca de excepciones, como desbordamientos, errores de validación, estados no manejados o vulnerabilidades explotables.

2.2 ¿QUÉ ES LA EJECUCIÓN SIMBÓLICA?

La ejecución simbólica es un método de análisis de un programa que permite determinar qué entradas provocan la ejecución de cada parte del programa. Es decir, un intérprete recorre el programa, asumiendo valores simbólicos para las entradas en lugar de obtener entradas reales como lo haría la ejecución normal del programa. De este modo, se llega a obtener expresiones en términos de símbolos para las variables del programa, y restricciones en términos de estos símbolos para posibles resultados de ramas condicionales. Por último, las posibles entradas que activan una rama pueden determinarse resolviendo dichas restricciones.

El objetivo es analizar un programa de forma sistemática para descubrir qué puede pasar con distintas entradas posibles, identificando caminos de ejecución que llevan a errores o comportamientos no deseados y, cuando hace falta, obteniendo ejemplos concretos de entradas que los provocan.

La ejecución simbólica permite identificar fallos y vulnerabilidades potenciales porque explora rutas del programa y busca entradas que hagan que se cumplan condiciones peligrosas (por ejemplo, llegar a un desbordamiento, un acceso fuera de límites, una división por cero, etc.). Además, suele producir un input concreto que dispara el problema, lo que hace mucho más fácil reproducirlo y corregirlo.

3. PREPARACIÓN DEL ENTORNO

3.1 SELECCIÓN DEL OBJETIVO (FUNCIÓN, BINARIO O MÓDULO)

La selección del objetivo se considera una fase estratégica a la hora de determinar el alcance y la eficiencia del análisis. En ocasiones, no es práctico o necesario realizar un proceso de fuzzing o ejecución simbólica sobre la totalidad de una aplicación compleja; en la mayoría de los casos, los análisis deben ser focalizados en componentes específicos que presenten superficies de ataque relevantes.

Los criterios mínimos recomendados para determinar el objetivo son los siguientes:

- Módulos que reciben entradas desde el exterior.
- Funciones con múltiples ramas condicionales susceptibles a comportamientos no intencionados.
- Componentes comúnmente asociados con fallos de memoria o relacionados.
- Secciones críticas del software que representen un riesgo de compromiso sobre la confidencialidad, integridad o disponibilidad.

Para ello, es aconsejable realizar un análisis preliminar, mediante una inspección manual del código, con el fin de delimitar un punto óptimo de entrada para las pruebas que se quieran realizar.

3.2 ANÁLISIS INICIAL DEL PROGRAMA

Para ejecutar eficientemente el fuzzing o la ejecución simbólica es necesario llevar a cabo un análisis preliminar exhaustivo del programa. El objetivo principal de este análisis es proporcionar información contextual del comportamiento de la aplicación con el fin de identificar puntos débiles y contribuir a la planificación de pruebas más enfocadas.

Este análisis inicial debe incluir:

- Identificación de dependencias clave como librerías o módulos auxiliares, entre otros.
- Superficies de entrada como parámetros de funciones, argumentos en línea de comandos, ficheros de entrada, etc.
- Estudio del manejo de errores con el fin de verificar las excepciones y validaciones aplicadas.
- Detección de puntos críticos como bucles, conversiones o estructuras de memoria que puedan derivar en comportamientos anómalos.

4. BUENAS PRÁCTICAS PARA FUZZING

4.1 SELECCIÓN DEL TIPO DE FUZZER

La elección del fuzzer debe basarse principalmente en el tipo de entrada que procesa el objetivo, ya sea una función, un binario o un módulo. Si el objetivo no se trata de un formato de entrada estructurado, los fuzzers basados en cobertura suelen ofrecer mejores resultados, ya que permiten explorar el espacio de ejecución de forma más eficiente. Por el contrario, cuando se trabaja con formatos altamente estructurados (protocolos, formatos binarios complejos, lenguajes de dominio específico), los fuzzers que utilizan generadores basados en gramáticas suelen ser una mejor opción, al producir entradas válidas con mayor frecuencia y alcanzar estados internos más profundos.

Otro aspecto fundamental a la hora de seleccionar un enfoque de fuzzing es determinar si se dispone del código fuente del objetivo o únicamente del binario. Esta distinción afecta de forma directa a la instrumentación posible, la profundidad de análisis y la visibilidad del comportamiento interno del programa.

4.1.1 Fuzzing con solo binario

Cuando únicamente se dispone del binario el fuzzing se realiza directamente sobre este, empleando técnicas como instrumentación binaria, emulación o análisis dinámico. Esto permite auditar cualquier software sin modificarlo, siendo útil en entornos cerrados o de terceros. Al no disponer de más información sobre el software existen ciertas limitaciones:

- **La instrumentación es menos precisa:** Sobre el binario, el proceso e instrumentación del binario se realiza a nivel de instrucciones decompiladas o identificadas previamente mediante ingeniería inversa (reescritura del binario, trampas, breakpoints, hooks, etc.) o interceptar llamadas a funciones/ramas. Esto suele dar menos fidelidad que instrumentar en compilación (donde conoces exactamente bloques básicos, líneas, funciones, inlining, etc.). Además, insertar lógica extra en tiempo de ejecución suele costar más (más saltos, más overhead por cada bloque ejecutado), y por ello hacer que el proceso sea más lento.
- **El rendimiento baja**: Un fuzzer mejora cuantas más entradas por segundo puede probar. Con solo binario, a menudo es necesario emulación (p. ej., ejecutar en un motor que interpreta instrucciones) o análisis dinámico pesado (tracing, monitoreo de memoria, hooks de syscalls). Todo eso añade sobrecarga que hace que se ejecute todo más lento y tarda más por cada caso de prueba, y el fuzzer hace menos iteraciones, lo que reduce la probabilidad de encontrar rutas raras o fallos profundos en el mismo tiempo.
- **Dificultad para obtener detalles internos del fallo**: En la gran mayoría de los casos la aplicación estará compilada en modo “Release” de manera que no se

cuenta con símbolos ni información de depuración, por lo que cuando se produce un fallo producido por el proceso de fuzzing normalmente se tiene menos contexto, tal vez una dirección de memoria, un registro y un backtrace más complicado de interpretar con direcciones en vez de nombres de funciones, etc. También es más complicado determinar el estado interno como estructuras o variables y localizar la causa raíz, cómo, por ejemplo, qué campo se corrompió antes, etc. Es posible recuperar esta información mediante el desensamblado, mapas de memoria o heurísticas, pero el diagnóstico suele ser menos directo, más tedioso y costoso en tiempo.

- **No es posible usar sanitizadores**: Con el binario ya compilado, no se pueden añadir comprobaciones de forma equivalente, salvo soluciones parciales con instrumentación dinámica, que no suelen igualar la cobertura/calidad de los sanitizadores. Esto da como resultado la detección de fallos por corrupción o interrupción de la ejecución, pero se te escapan errores sutiles como corrupciones silenciosas o comportamientos no definidos que no tienen un fallo catastrófico por lo que se tiene menos información cuando ocurre.

4.1.2 Fuzzing con acceso al código fuente

Cuando se dispone del código fuente, los fuzzers pueden instrumentarlo durante la compilación. Esto permite obtener:

- **Cobertura precisa**: Al compilar con instrumentación, el compilador puede insertar contadores y marcas exactamente en bloques básicos, ramas y/o aristas del CFG (control flow graph). Eso da una señal de cobertura más fiel y estable generando menos ruido por efectos de reescritura binaria y permite al fuzzer saber con mucha precisión si un input nuevo realmente explora código nuevo. En fuzzers guiados por cobertura esa señal es la que dirige la mutación: se quedan y mutan más los inputs que abren caminos nuevos o ejecutan rutas menos frecuentes.
- **Mayor rendimiento**: Con código fuente puedes compilar el binario “pensado para fuzzing” mediante instrumentación ligera**,** la mínima necesaria y bien integrada en el flujo del compilador. El uso de flags adecuados facilita una ejecución rápida y repetible, así como la posibilidad de enlazar con puntos de entrada de fuzzing que evitan overhead de arrancar el programa completo cada vez.
  Todo esto permite más ejecuciones por segundo, que es clave para encontrar fallos antes y cubrir más superficie del programa.
- **Información detallada de ejecución**: Con símbolos e información de depuración, cuando hay un fallo se obtienen *stack traces* con nombres de funciones, ficheros y líneas, y se puede correlacionar con partes concretas del código. Además, se puede instrumentar a nivel de función/bloque para recoger métricas, logs

controlados o controles internos, y es más fácil entender qué condición llevó al fallo, reproducirlo y arreglarlo.

- **Compatibilidad con sanitizadores:** los sanitizadores funcionan añadiendo *código extra* alrededor de operaciones sensibles cómo accesos a memoria, aritmética, llamadas, etc. y, en algunos casos, cambiando el layout de memoria (redzones, shadow memory). Esa transformación está diseñada para hacerse en el pipeline del compilador (front-end/IR/back-end) y enlazarse con runtimes específicos de manera que todo esto permite tener una mayor cobertura y la detección de fallos más complejos o que requieren una casuística más completa.

4.2 SANITIZADORES

Los sanitizadores son herramientas de instrumentación que añaden comprobaciones adicionales al programa para detectar errores sutiles (especialmente de memoria y comportamiento indefinido) durante la ejecución. Esta instrumentación se inserta en el proceso de compilación, ya que requiere modificar el código generado e integrar runtimes específicos; por ello, cuando solo se dispone del binario no es posible utilizarlos con el mismo nivel de soporte y detalle.

Cuando el código fuente está disponible, el fuzzing suele ser más eficaz precisamente porque puede apoyarse en sanitizadores: permiten detectar fallos que no siempre terminan en crash y, además, generan diagnósticos mucho más informativos para localizar la causa.

4.2.1 Uso de sanitizadores

El uso de sanitizadores durante el proceso de fuzzing es una técnica altamente recomendable, ya que permite identificar errores que en ejecución normal pueden pasar desapercibidos o ser difíciles de reproducir. Los sanitizadores instrumentan el binario en tiempo de compilación para realizar verificaciones adicionales en tiempo de ejecución, de forma que errores silenciosos se convierten en fallos detectables y reproducibles.

Su uso es especialmente eficaz cuando se realiza fuzzing con acceso al código fuente, ya que la instrumentación se integra directamente durante la compilación. Esto complementa al fuzzer y amplía la visibilidad sobre posibles vulnerabilidades.

Los sanitizadores más utilizados son:

- **Address Sanitizer** (ASan): Detecta errores de memoria en *heap*, *stack* y variables globales. En la práctica es muy bueno encontrando accesos fuera de límites (buffer overflow/underflow), use-after-free, double free, y corrupciones de memoria que a veces no fallan inmediatamente. Lo hace manteniendo una “memoria sombra” (*shadow memory*) y añadiendo redzones alrededor de

buffers para saber si te sales. Suele dar informes muy útiles con *stack trace* de dónde se reservó/liberó/usó la memoria

- **Undefined Behavior Sanitizer** (UBSan): Señala comportamientos indefinidos típicos de en lenguajes como C/C++ pero no exclusivos, que pueden producir resultados arbitrarios (a veces sin crash) y que el compilador puede optimizar de formas inesperadas. Ejemplos comunes: overflow de enteros con signo, shifts inválidos cómo desplazar más bits de los que tiene el tipo o desplazar un valor negativo, división por cero, casts inválidos (p. ej. downcast incorrecto con RTTI), misaligned access, o violaciones de ciertas reglas del lenguaje.
- **Memory Sanitizer** (MSan): Detecta lecturas de memoria no inicializada como usar un valor que nunca se ha escrito, depender de bytes basura en una comparación, en un *if*, en el cálculo de un índice, o propagar datos no inicializados por estructuras/copias. Es especialmente útil porque estos bugs suelen no *crashear* y pueden causar decisiones de control erráticas o fallos intermitentes. También usa *shadow memory* para marcar qué bytes están inicializados y propagar esa "contaminación" (*taint*) por operaciones. Una limitación importante es que para que sea fiable, normalmente necesitas que todas las librerías relevantes estén instrumentadas (runtime/libc), y el *overhead* suele ser alto.
- **Thread Sanitizer** (TSan): Detecta condiciones de carrera (*race conditions*) en programas concurrentes, dos hilos accediendo a la misma memoria sin sincronización adecuada, con al menos una escritura. También suele ayudar a ver problemas de sincronización/mal uso de locks, porque monitoriza accesos y operaciones de sincronización para inferir el orden entre hilos. En fuzzing o pruebas concurrentes es muy valioso para bugs que aparecen rara vez, pero tiene alto coste de rendimiento y puede requerir ajustar el entorno (tests deterministas, minimizar ruido) para reproducir y depurar bien lo que reporta.

Algunos sanitizadores no pueden combinarse directamente entre ellos debido a conflictos en la instrumentación o en la gestión de memoria.

Estas incompatibilidades son:

Sanitizador 1	Sanitizador 2	Comentarios
ASan	MSan	Ambos dependen de instrumentación "pesada" de memoria y de un **runtime** con **shadow memory**. MSan, además, mapea una cantidad enorme de espacio de direcciones virtual (p. ej. decenas de TB) para su shadow, lo que choca con el layout/expectativas de otros sanitizadores.

ASan	TSan	Los dos necesitan su propio runtime y gestionar muchas operaciones (malloc/free, librerías, gestión de hilos, etc.). Sus modelos de instrumentación y de seguimiento no están diseñados para coexistir en un mismo ejecutable. ASan interfiere con el seguimiento de hilos de TSan
MSan	TSan	MSan se centra en propagar estado de inicialización (taint) en memoria/valores, y TSan en relaciones de sincronización y accesos concurrentes. La combinación implicaría doble instrumentación y runtimes con requisitos incompatibles, por eso no se soporta.
UBSan	Cualquier otro	UBSan está pensado para ser compatible: tiene poco impacto en ABI/layout y coste relativamente bajo. El inconveniente suele ser más bien de ruido/volumen de reportes como por ejemplo checks muy auto explicativos o reportes duplicados si ya *crashea* por otra causa, más que falsos positivos reales. Pueden combinarse, pero pueden generar falsos positivos

Tabla 1: Incompatibilidades entre sanitizadores

Aunque los sanitizadores son muy útiles para encontrar errores, añaden instrumentación al binario donde el compilador inserta comprobaciones extra alrededor de accesos a memoria, operaciones peligrosas y llamadas a librerías, y además se apoya en *runtimes* que mantienen estructuras internas como por ejemplo *shadow memory*. Eso implica más instrucciones ejecutadas, más uso de memoria y, en muchos casos, peor comportamiento de cachés; por tanto, el programa suele ir sensiblemente más lento y consumir más recursos que una *build* normal, tanto de *Debug* como de *Release*.

Por esa razón, los binarios con sanitizadores están pensados para desarrollo, testing y fuzzing, no para producción ni para benchmarks. En producción penalizan el coste y pueden alterar el comportamiento temporal del sistema (latencias, timeouts, scheduling, concurrencia), y en pruebas de rendimiento sesgan los resultados porque miden el *overhead* del sanitizador más que el rendimiento real del software. Además, algunas configuraciones pueden abortar ante detecciones para facilitar el diagnóstico, lo que tampoco encaja con un entorno productivo.

4.2.2 Elección de sanitizadores

La selección del sanitizador adecuado depende de las características del software objetivo, el tipo de errores que se desea identificar y el impacto aceptable en el rendimiento durante el fuzzing. Cada sanitizador detecta una familia diferente de problemas, por lo que elegir el más apropiado maximiza la eficiencia de la campaña de pruebas y evita sobrecargas innecesarias.

Recomendaciones prácticas según escenario:

Escenario	Sanitizador recomendado	Comentarios
Aplicaciones C/C++ estándar	ASan	Mejor equilibrio rendimiento–detección. Detecta bugs de memoria "clásicos": out-of-bounds, use-after-free, double free, corrupción de heap/stack. Suele rondar ~2× de ralentización.
Aplicaciones con estructuras complejas	MSan	Detecta lecturas/acceso de memoria no inicializada (valores no inicializados en condiciones, cálculos, índices, etc.), que a menudo no crashean. Consumo de memoria: ~2× de memoria real (≈3× con origin tracking) y mapea gran cantidad de espacio virtual.
Aplicaciones críticas en lógica	UBSan	Identifica comportamientos indefinidos: signed overflow, shifts inválidos, misaligned/null deref, algunos out-of-bounds determinables, etc. Suele ser ligero comparado con ASan/TSan/MSan (se considera "fast")
Software multihilo	TSan	Detecta condiciones de carrera (data races conditions) y errores de sincronización asociados. Ralentización típica ~5×–15× y overhead de memoria ~5×–10×

Entornos mixtos o primeros pasos	ASan + UBSan	Compatibles y complementarios, Combina memoria + UB: detectas crashes por corrupción y también UB que quizá no crashea. El coste computacional y de uso de memoria es normalmente asumible (más que TSan/MSan)

Tabla 2: Elección de sanitizador

Los costes computaciones y de uso de memoria se han estimado en base a la media de proyectos haciendo uso de estos sanitizadores, no debe tomarse como una regla, pero si como algo informativo e ilustrativo a la hora de decidir cuál de los sanitizadores aplicar.

4.2.2.1 Estrategia recomendada: enfoque por fases

Para equilibrar rendimiento y capacidad de detección, suele ser más eficaz dividir el fuzzing en fases, empezando por los sanitizadores que ofrecen más hallazgos por segundo y dejando los más costosos para campañas dirigidas. Así se obtiene valor rápido desde el inicio sin frenar el ritmo de ejecución, y a medida que el objetivo se estabiliza se amplía el tipo de errores que pueden aflorar.

- **Fase 1: Fuzzing con ASan (habitualmente en conjunto con LSan)**

  En una primera fase se prioriza ASan (habitualmente con detección de fugas activada), porque permite encontrar de forma temprana una gran parte de los fallos explotables típicos en C/C++ —accesos fuera de límites, *use-after-free*, dobles liberaciones y corrupciones de *heap/stack*— con un overhead relativamente asumible, manteniendo un número alto de ejecuciones por segundo. Además, corregir pronto este tipo de corrupciones reduce los fallos en cascada que enmascaran otros problemas, dejando el programa en un estado más limpio para análisis posteriores y haciendo que los *crashes* y reportes sean más consistentes y fáciles de depurar.

- **Fase 2: Fuzzing con UBSan**

  Una segunda fase con UBSan permite afinar la campaña detectando comportamientos indefinidos que no siempre se manifiestan como un crash inmediato, pero que pueden romper invariantes/variables, provocar estados inconsistentes o incluso permitir bypasses lógicos. En este punto ya se han eliminado buena parte del ruido de corrupción de memoria, por lo que los reportes de UBSan suelen ser más accionables; aquí aparecen casos como desbordamientos con signo, desplazamientos inválidos, divisiones por cero o conversiones problemáticas, que pueden ser el origen real de fallos posteriores o de comportamientos inesperados que el fuzzer estaba alcanzando sin señales claras.

- **Fase 3: Fuzzing específico según el objetivo (campañas dirigidas)**

  En etapa final, se enfoca según el tipo de software y el objetivo de auditoría, dedicando campañas específicas a los sanitizadores más caros en recursos, pero más especializados. Si se trata de componentes con estructuras complejas, parsers o rutas donde pequeñas decisiones dependen de datos parcialmente inicializados, MSan ayuda a descubrir lecturas de memoria no inicializada que alteran condiciones, cálculos o índices sin necesidad de crashear. Por otro lado, si el proyecto es concurrente, TSan se vuelve clave para sacar a la luz condiciones de carrera y fallos de sincronización que suelen ser intermitentes y difíciles de reproducir, aunque implique reducir el rendimiento.

Con este enfoque escalonado se cubren progresivamente más familias de vulnerabilidades y se emplean los recursos donde aportan más valor, evitando sobrecargar el fuzzing desde el principio. Se comienza con lo que da más hallazgos por unidad de tiempo (ASan), luego se amplía a errores menos evidentes (UBSan) y finalmente se invierten recursos en campañas costosas pero muy valiosas (MSan/TSan) solo cuando hay razones para ello. Así se cubren progresivamente más familias de vulnerabilidades sin frenar el fuzzing desde el primer minuto.

4.3 CONSTRUCCIÓN DEL CORPUS INICIAL

En fuzzing, el corpus es el conjunto de entradas semilla como archivos, mensajes o casos de prueba, con las que se inicia y se alimenta la campaña. Estas entradas actúan como punto de partida: el fuzzer las ejecuta, mide qué partes del programa alcanzan y, a partir de ellas, genera nuevas variantes mediante mutaciones o recombinaciones, conservando aquellas que aportan cobertura nueva o comportamientos interesantes.

Un corpus inicial bien construido es una de las piezas más determinantes para que una campaña de fuzzing sea efectiva y produzca resultados útiles. En la práctica, el fuzzer aprende y progresa a partir de esas entradas semilla: si el corpus refleja bien el formato y los casos de uso reales, el fuzzer puede generar mutaciones que conservan la estructura mínima necesaria para que el programa procese el input en vez de descartarlo en las primeras validaciones. Esto se traduce en más ejecuciones productivas, mayor cobertura, y una probabilidad mucho más alta de alcanzar rutas de código internas, estados complejos y condiciones límite donde suelen aparecer errores.

Por ese motivo, conviene que el corpus incluya ejemplos válidos y representativos del formato objetivo como archivos, mensajes, peticiones, etc., cubriendo distintas variantes permitidas y casos habituales, pero también incorporando entradas que fuercen límites y combinaciones relevantes, como tamaños pequeños y grandes, campos opcionales presentes y ausentes, valores extremos, codificaciones distintas o estructuras anidadas. A partir de esa base, pequeñas variaciones controladas como cambios en longitudes, cabeceras, checksums, campos de control o delimitadores, suelen ser más útiles que inputs completamente aleatorios, porque proporcionan

puntos de apoyo para que las mutaciones mantengan coherencia suficiente como para atravesar más lógica del programa y descubrir fallos en fases de parsing, validación y procesamiento profundo

Un corpus adecuado debe:

- **Cubrir la mayor variedad de estructuras posibles del formato de entrada**. Esto implica incluir semillas que ejerzan todas las "formas" que admite el formato: distintas versiones, cabeceras alternativas, campos opcionales presentes y ausentes, diferentes órdenes de secciones cuando el formato lo permite, listas con longitudes variables, estructuras anidadas a distintas profundidades, y combinaciones poco comunes pero válidas. La idea es que el fuzzer no se quede atascado explorando siempre el mismo tipo de mensaje/archivo y, desde el inicio, tenga puntos de entrada hacia ramas del parser y del procesamiento que solo se activan con ciertas variantes estructurales.
- **Casos de uso reales, incluidos datos típicos utilizados por usuarios o sistemas**. Si el corpus se parece a lo que el software procesa en producción, el fuzzing será más representativo y tenderá a recorrer rutas que importan de verdad: configuraciones habituales, tamaños y distribuciones de datos comunes, patrones de contenido típicos, por ejemplo, nombres, rutas, timestamps, compresión, codificaciones, y también entradas ruidosas realistas cómo truncados, caracteres extraños, espacios, valores repetidos que ocurren en la práctica. Además, esto ayuda a priorizar vulnerabilidades con impacto real, porque se descubren en rutas que se ejecutan con frecuencia y no solo en casos artificiales.
- **Ejemplos válidos y completos, importante si el objetivo son validaciones estrictas.**
  Muchos objetivos descartan entradas en etapas tempranas si fallan checks de integridad cómo magic bytes, longitudes, checksums, firmas, coherencia entre campos. Si el corpus no pasa esos filtros, el fuzzer solo ejercitará la capa superficial y no llegará al código profundo donde suelen aparecer los fallos. Por eso conviene aportar muestras que cumplan las reglas del formato, incluyendo relaciones internas coherentes, longitudes que cuadran, offsets correctos, tablas consistentes, checksums válidos si aplica. En formatos complejos, unas pocas semillas que cubran múltiples casos son la clave para desbloquear la exploración.
- **Permitir mutaciones efectivas, usando entradas lo bastante pequeñas y diversas para generar nuevas rutas de ejecución**.
  Entradas grandes o muy redundantes dificultan que una mutación encuentre en algo útil: pequeños cambios pueden romper la validez global y hacer que el input se rechace antes. En cambio, semillas más pequeñas, modulares y variadas facilitan que el fuzzer haga cambios localizados que sigan siendo procesables, lo que incrementa la probabilidad de descubrir nueva cobertura. También ayuda

incluir semillas que difieran en puntos sensibles cómo valores límite, longitudes cercanas a umbrales, presencia/ausencia de campos, porque esas diferencias sirven como palancas para que las mutaciones atraviesen condiciones, activen manejadores alternativos y exploren rutas que no aparecen con datos más uniformes.

Se recomiendan los siguientes tipos de entradas:

- **Entradas válidas**: Son la base del corpus porque permiten que el programa supere los filtros iniciales del parser y llegue a la lógica real de procesamiento, donde suelen aparecer fallos más interesantes. En formatos con validación estricta, una semilla válida no solo debe tener cabeceras correctas, sino también coherencia interna: longitudes que cuadran con el contenido, offsets que apuntan a zonas existentes, recuentos consistentes con el número de elementos, y checksums o campos de integridad correctos si el formato los exige. También conviene que estas entradas válidas cubran distintas combinaciones permitidas (campos opcionales, compresión sí/no, secciones presentes/ausentes, diferentes codificaciones), porque cada variación válida puede desbloquear rutas profundas distintas.

  Estas entradas válidas se pueden obtener de fuentes reales y reproducibles cómo documentación oficial del formato con ejemplos, suites de test o repositorios del propio proyecto, muestras generadas por herramientas de referencia o clientes oficiales si los hubiera, y capturas/exports de uso normal generados por los usuarios. Si existe un generador del formato (CLI, API o librería), también se puede crear un pequeño generador que produzca archivos correctos variando opciones, por ejemplo, con/sin compresión, distintas secciones opcionales, distintos tamaños, etc., así se garantiza validez y diversidad.

- **Entradas levemente corruptas**: Se trata de inputs parcialmente válidos en los que se introduce una pequeña inconsistencia controlada, de forma que el programa sigue entrando en el parser y avanza lo suficiente como para ejercitar el manejo de errores. Son útiles para ver si el software falla de forma segura o si aparecen problemas como lecturas fuera de rango al confiar en longitudes incorrectas, bucles inesperados, estados parcialmente inicializados o liberaciones dobles al abortar. Ejemplos típicos son: un campo de longitud que difiere ligeramente del tamaño real, un offset que apunta cerca del final, un recuento de elementos que no coincide con la lista, un delimitador cambiado, o un checksum incorrecto manteniendo el resto coherente. Este tipo de semillas suele activar caminos de “error handling” que están menos probados y pueden contener bugs.

  Para generar estas entradas corruptas se parte de las válidas y se aplica de manera automática o manual corrupciones pequeñas y controladas para que sigan pareciéndose al formato cómo: truncar unos pocos bytes al final, insertar

o borrar un byte en una sección concreta, modificar un campo de longitud u offset ligeramente, cambiar un delimitador, o romper un checksum manteniendo todo lo demás igual. Esto puede hacerse con scripts simples o usando el propio fuzzer en modo de mutación sobre semillas, pero guardando como semillas aquellas que siguen alcanzando parsing profundo antes de fallar. La clave es evitar corrupciones aleatorias masivas que hagan que el programa rechace el input al primer byte. Para este caso, es útil contar con la especificación del formato de entrada de los datos para poder entender que valores se están modificando y conseguir un mejor resultado.

- **Casos límite**: Buscan forzar condiciones frontera extremas y umbrales internos del código, que es donde se concentran muchos fallos: tamaños 0 y 1, longitudes máximas permitidas, valores justo por encima o por debajo de límites, listas vacías frente a listas muy grandes, y combinaciones de campos opcionales que producen estructuras mínimas o máximas. También incluyen valores extremos numéricos (por ejemplo, máximos de 8/16/32/64 bits, negativos si aplica, o valores especiales), cadenas vacías, caracteres no ASCII, y entradas con padding inusual. La ventaja es que ayudan a disparar ramas de validación, cálculos de tamaño, reservas de memoria y conversiones que no se ejercitan con datos normales.

  Estos se pueden generar a partir de una estructura válida del formato, usando un generador disponible o creando uno sencillo que permita fijar tamaños y valores en los campos críticos. Con esa base se producen grupos de archivos con tamaños 0 y 1, tamaños máximos razonables, valores justo por debajo y por encima de límites habituales, listas vacías y listas muy grandes, cadenas vacías y cadenas muy largas, y combinaciones mínimas y máximas de campos opcionales. Si el formato no define límites de forma clara, los deduciría revisando el código fuente cuando esté disponible u observando el comportamiento del parser, y después construiría casos alrededor de esos umbrales para forzar rutas límite.

  En este caso, disponer de la especificación del formato de entrada es especialmente útil, ya que permite entender qué campos y restricciones se están modificando y dirigir las variaciones de forma más precisa, obteniendo resultados más efectivos.

- **Versiones alternativas del formato**: Muchos formatos y protocolos evolucionan: cambian cabeceras, se añaden campos, se deprecian secciones, o aparecen variantes con extensiones. Incluir semillas de distintas versiones obliga al objetivo a recorrer rutas específicas de compatibilidad (parsers por versión, lógica de migración, manejo de campos desconocidos o legacy) que pueden ser una fuente importante de vulnerabilidades. Además, si el software soporta negociación de versión o autodetección, estas semillas ayudan a cubrir las ramas que deciden qué parser se aplica y cómo se interpretan los datos, evitando que

el fuzzing se quede anclado en una única variante moderna y se pierda código antiguo que sigue activo.

Las muestras se pueden obtener buscando muestras de cada versión, documentación histórica del formato, repositorios antiguos del proyecto, etiquetas/releases anteriores, compat tests, y archivos de ejemplo distribuidos con herramientas o SDKs viejos. Si el software incluye compatibilidad hacia atrás, a menudo hay fixtures o ejemplos en el propio repositorio. Cuando no existan muestras se puede emplear herramientas de conversión/export de versiones (si existen), de nuevo variando opciones en la medida de lo posible para que no sea una única muestra por versión.

Una forma práctica de empezar es apoyarse en muestras reales ya conocidas, como las que encajan en los tipos anteriores, y aprovechar también entradas procedentes de incidencias reales, por ejemplo, casos que usuarios hayan reportado al provocar fallos.

Además, el corpus no debería quedarse estático: conviene que vaya evolucionando a lo largo de la campaña incorporando entradas que aporten nueva cobertura y permitan explorar rutas adicionales del programa.

4.4 MONITORIZACIÓN Y LOGGING

Durante una campaña de fuzzing es recomendable registrar de forma continua métricas que permitan saber si el proceso está siendo efectivo y si el entorno se mantiene estable. Las más útiles suelen ser la velocidad de ejecución en ejecuciones por segundo, el tiempo total acumulado, el tamaño del corpus efectivo, la cantidad de rutas nuevas descubiertas a lo largo del tiempo, el número de crashes únicos, los timeouts o cuelgues y el consumo de CPU y memoria. Estas métricas no solo sirven para ver el progreso, sino que también ayudan a detectar problemas prácticos como una caída brusca de rendimiento por throttling, saturación de memoria, inputs que provocan bucles largos, o configuraciones del objetivo que hacen que la mayoría de las entradas se rechacen demasiado pronto. En campañas largas, tener un histórico permite comparar runs, identificar en qué momento se estancó la cobertura y justificar cambios de estrategia como modificar el punto de entrada del fuzzing para reducir inicialización innecesaria y hacer que cada entrada ejercite más lógica interna, añadir semillas, cambiar límites de tiempo o ajustar el entorno de ejecución.

Además del conteo bruto de fallos, el logging debería guardar información que permita reproducir y depurar. Lo habitual es conservar, por cada crash, el input que lo provoca, un identificador estable del fallo para deduplicar, por ejemplo, hash del stack trace o del tipo de señal, y la salida del runtime cuando se usan sanitizadores, ya que ahí suele estar el detalle de la causa. También es recomendable registrar parámetros exactos de ejecución y build del objetivo, como flags de compilación, tipo de sanitizador, versión

del fuzzer, commit del binario, sistema operativo y configuración relevante, porque pequeñas diferencias cambian la reproducibilidad. Cuando el volumen de fallos es alto, resulta especialmente útil aplicar minimización automática de inputs y reducción de casos a crashes representativos, ya que acelera mucho el triaje y evita perder tiempo con duplicados.

En cuanto a cobertura, muchos fuzzers y toolchains generan artefactos para medir qué parte del código se está ejecutando. Según la tecnología usada, esa información puede almacenarse como ficheros específicos del fuzzer o del compilador, por ejemplo, perfiles de LLVM, ficheros de gcov u otros formatos que algunas herramientas agrupan bajo extensiones como .cov. La utilidad principal es poder mapear la actividad del fuzzer a funciones, líneas o ramas alcanzadas y, sobre todo, localizar zonas que siguen sin explorarse. Con esos datos puedes mejorar el corpus de forma dirigida, añadiendo semillas que activen funcionalidades concretas, creando inputs válidos para pasar validaciones iniciales, incorporando diccionarios de tokens del formato, o modificando el punto de entrada del fuzzing para que el procesamiento comience directamente en una capa más interna del programa. También ayuda a decidir si compensa invertir en fases específicas, por ejemplo, activar un mutador más estructural, cambiar el enfoque de generación de entradas o separar módulos para fuzzing más focalizado cuando se observa que una parte del programa queda sistemáticamente fuera de alcance.

4.5 REPRODUCIBILIDAD DE CRASHES

Para investigar bien un crash descubierto durante fuzzing hay que poder reconstruir exactamente las mismas condiciones en las que apareció. Eso empieza por conservar el archivo de entrada que lo provoca, pero también exige guardar el binario exacto que se estaba ejecutando, las opciones de compilación utilizadas y la configuración del entorno. Sin esa información se puede perder el fallo, obtener resultados distintos o perseguir un problema que solo ocurre por una combinación concreta de sistema, librerías o configuración. La reproducibilidad es la base para decidir si el crash corresponde a un error real en el software, a un efecto secundario del entorno o a un comportamiento no determinista como una race condition o fallo similar.

El primer paso del análisis consiste en volver a ejecutar el programa con la misma entrada y comprobar que el fallo aparece de forma consistente. Para ello hay que replicar los mismos argumentos y variables de entorno, la misma versión del código o del binario, y los mismos sanitizadores, ya que cambian el layout de memoria, el orden de ejecución y la forma de reportar el error. También influyen detalles como el límite de tiempo por ejecución, el número de hilos, el directorio de trabajo, el locale, la configuración de asignación de memoria y la versión de las librerías del sistema. Por ejemplo, un use after free puede manifestarse como un crash claro con ASan, pero parecer un fallo intermitente sin sanitizador, mientras que una condición de carrera puede desaparecer si se ejecuta con un único hilo o si cambia el scheduling. Por eso, si

se quiere que cualquiera pueda reproducir el problema, conviene documentar y versionar el comando exacto de ejecución y adjuntar la información del build, incluyendo commit, flags y runtime de los sanitizadores.

Asegurar la reproducibilidad mejora la colaboración entre equipos, permitiendo que otra persona pueda ejecutar el mismo caso y ver el mismo fallo sin haber estado presente durante la campaña de fuzzing. Para lograrlo, no basta con compartir el input: conviene acompañarlo de un paquete mínimo de contexto que incluya el binario exacto, un comando de ejecución listo para copiar y pegar, y la información del entorno. En la práctica, esto suele traducirse en guardar el commit o versión del código, los flags de compilación, el tipo de sanitizador, la versión del compilador y del fuzzer, y, si aplica, un contenedor o imagen reproducible. Muchos fuzzers ya ayudan en esta tarea porque separan automáticamente los inputs que causan fallos, mantienen logs asociados y suelen deduplicar crashes por firmas, de forma que el equipo se centra en fallos únicos y no en miles de repeticiones del mismo problema.

Además, la reproducibilidad es lo que permite evaluar el alcance real del crash y decidir su prioridad. Solo con una reproducción controlada se puede clasificar el impacto en disponibilidad, integridad o confidencialidad, y determinar si el fallo es explotable y con qué fiabilidad. Por ejemplo, un crash que aparece siempre al procesar un archivo puede ser un claro problema de denegación de servicio, mientras que un desbordamiento fuera de límites con escritura y control sobre el tamaño puede apuntar a ejecución de código si se dan condiciones adicionales. En cambio, hay fallos que dependen de timings, de ASLR, del planificador de hilos o de la disposición del heap, y entonces el análisis debe medir si el comportamiento es estable o si requiere circunstancias poco realistas. Esta fase también ayuda a separar casos que parecen graves pero no lo son, como reportes causados por instrumentación de sanitizadores, abortos por checks internos del runtime o entradas que disparan asserts solo en builds de depuración. Sin reproducir y aislar el fallo, es fácil confundir ruido con vulnerabilidades reales o invertir tiempo en problemas que no tienen relevancia práctica.

Es normal que, tras una campaña de fuzzing larga, se acumulen gran cantidad de crashes, porque un mismo defecto puede dispararse con múltiples entradas diferentes y además la clasificación automática suele sobrecontar, ya que variaciones pequeñas en la ruta ejecutada o en el stack trace pueden hacer que el fuzzer los guarde como diferentes aunque la causa sea la misma; por eso, cuando se duplican por causa raíz, lo habitual es que el número de fallos realmente distintos sea una fracción muy pequeña del total, típicamente por debajo del 1% y en campañas grandes de larga duración incluso alrededor del 0,01% al 0,1%, lo que implica que más del 99% de los crashes acaban siendo duplicados o variantes del mismo bug. A esto se suma que no todo crash tiene el mismo impacto, muchos corresponden a lecturas nulas o asserts que normalmente se traducen sobre todo en pérdida de disponibilidad, mientras que los casos más preocupantes suelen ser corrupciones de memoria con indicios de escritura

fuera de límites o use after free con efectos de escritura, porque se acercan más a primitivas explotables y se tratan como potencialmente críticos hasta que el triaje demuestre lo contrario.

5. BUENAS PRÁCTICAS PARA EJECUCIÓN SIMBÓLICA

5.1 SELECCIÓN DEL OBJETIVO

Intentar aplicar ejecución simbólica a un binario completo suele ser inviable porque el número de rutas posibles que genera crece rápidamente en cuanto aparecen ramas, bucles, estructuras de datos dinámicas y llamadas a librerías. En un programa real, además, el flujo puede depender de estado externo como ficheros, red, reloj, señales, variables de entorno o syscalls, y modelar todo eso con precisión obliga a introducir aproximaciones que añaden aún más bifurcaciones o generan restricciones muy difíciles para el solver. A esto se suma que la memoria dinámica y los punteros disparan la complejidad: cada acceso indexado, cada comparación contra longitudes calculadas y cada salto indirecto introduce condiciones que el solver debe mantener consistentes. El resultado típico es que el análisis se queda atascado explorando millones de caminos superficiales, consumiendo tiempo en simplificar fórmulas, o devolviendo respuestas poco útiles porque ha tenido que asumir modelos demasiado genéricos para el sistema y las librerías.

Por eso se acota el análisis a funciones o bloques concretos, idealmente aquellos que transforman directamente la entrada en decisiones de control, como parsers, validadores o rutinas de deserialización. Al reducir el alcance se puede construir un punto de entrada controlado donde la única fuente de incertidumbre sea el buffer de entrada y, si hace falta, unos pocos parámetros adicionales, dejando el resto del estado con valores concretos. Esto reduce de manera drástica el número de caminos y, sobre todo, produce restricciones más pequeñas y estructuradas, lo que permite que el solver encuentre soluciones en tiempos razonables y genere entradas que de verdad alcanzan condiciones interesantes. Por ejemplo, si se analiza solo la función que parsea un encabezado, el solver puede deducir rápidamente qué bytes activan una rama concreta o qué valores fuerzan un overflow de longitud, mientras que en el binario completo ese mismo efecto quedaría oculto por la lógica de inicialización y las capas previas de ejecución, manejo de errores y capas de I/O. Además, al trabajar sobre una región acotada es más fácil imponer límites realistas que mejoran la calidad de resultados, como acotar el tamaño de la entrada, fijar iteraciones máximas de bucles, concretar parte del input para atravesar validaciones tempranas y modelar llamadas externas con stubs sencillos que no introduzcan nuevos caminos.

5.2 DISEÑO DE RESTRICCIONES EFECTIVAS

En ejecución simbólica no conviene tratar como simbólico todo lo que entra en el programa o todo el estado que lo rodea. Cuantos más bytes, variables o punteros se simbolizan, más expresiones y dependencias se generan y más difícil se vuelve el problema para el solver, que tendrá que mantener consistentes muchas restricciones a la vez. Por eso, lo habitual es aplicar simbolización y restricciones solo a los valores que

realmente gobiernan el comportamiento del bloque que se está analizando, de modo que el solver trabaje con un conjunto reducido de incógnitas y el análisis no se pierda en rutas que no aportan valor. Esta selección es una de las decisiones más importantes para que el análisis sea escalable, porque marca la diferencia entre obtener casos de prueba útiles en segundos y quedarse atascado explorando combinaciones irrelevantes.

Para escoger qué entradas modelar de forma simbólica, se parte de la lógica real del código y de qué variables controlan ramas, validaciones y cálculos sensibles. En la práctica, suele ser suficiente simbolizar el input controlado por el usuario, como un buffer de bytes, un tamaño asociado, flags de configuración que cambian el modo de parseo o identificadores que determinan qué rutina se ejecuta. A partir de ahí, conviene concretar el resto del estado para reducir incertidumbre: valores como rutas de archivo, timestamps, identificadores de proceso, direcciones, o datos externos que no afectan a la propiedad que se quiere comprobar. Si se simboliza algo que apenas influye en el flujo, se multiplica el número de combinaciones sin ganar cobertura relevante, y el solver invertirá tiempo en satisfacer restricciones que no conducen a rutas nuevas.

Un ejemplo típico es un programa que lee un archivo o un mensaje y empieza por un encabezado. En ese encabezado suele haber un campo que indica el tipo de contenido y otros campos que indican longitudes o tamaños. El programa usa el tipo para elegir qué parte del código ejecutar y usa las longitudes para decidir cuántos bytes leer o copiar y cuántas iteraciones hacer en bucles. En ese caso, suele ser muy efectivo hacer simbólicos los primeros bytes del encabezado y el tamaño total del buffer, porque de esos valores salen comparaciones que abren ramas distintas y operaciones sensibles que pueden acabar en desbordamientos o lecturas fuera de rango.

En cambio, hacer simbólicos campos que no cambian el comportamiento, como un timestamp o un identificador de sesión que simplemente se guarda en una estructura sin influir en ninguna condición, normalmente no aporta rutas nuevas y solo hace el problema más grande para el solver. Algo similar ocurre con valores que vienen del entorno, como la hora del sistema o variables de entorno: si no son el foco del análisis, suele ser mejor fijarlos a valores concretos y, como mucho, probar una o dos configuraciones representativas, evitando que el solver pierda tiempo explorando combinaciones que no ayudan a encontrar fallos en la lógica que se está evaluando. Estos campos son candidatos para probar valores aberrantes o extremos, pero no para incluir en pruebas donde se realizan mutaciones sobre los datos con el fin de generar nuevas rutas ya que agregan una variabilidad que añade complejidad, pero no mejora el alcance o la cobertura de las pruebas.

Además de decidir qué simbolizar, también es útil restringir de forma explícita el dominio de los símbolos para guiar la exploración. Por ejemplo, si se analiza un formato que exige una cabecera concreta, imponer esa cabecera como restricción inicial evita que el solver pierda tiempo generando entradas que nunca pasarán la validación. Si un bucle depende de una longitud, acotar esa longitud a un rango razonable permite

explorar más rutas útiles sin disparar el número de iteraciones simbólicas. Con estas restricciones, el análisis se vuelve más dirigido: el solver encuentra entradas que atraviesan los checks iniciales, alcanza condiciones interesantes y produce ejemplos reproducibles que sirven directamente para pruebas o para alimentar un fuzzer con semillas de alta calidad.

Aplicar ejecución simbólica sobre una aplicación completa suele ser poco práctico, y por eso es más efectivo centrarla en partes concretas del programa. La mayoría de frameworks permiten arrancar la simbolización desde un punto de interés, como una función concreta del parser, una rutina de validación, un manejador de un tipo de mensaje o un tramo del flujo que ya se sabe cómo alcanzar. En la práctica esto equivale a elegir una porción del código donde realmente se toman decisiones relevantes y donde suelen aparecer errores, y evitar que el análisis se consuma en inicializaciones, manejo de argumentos, configuración del entorno o rutas de error genéricas que no aportan información. Al reducir el alcance, se generan menos estados y menos bifurcaciones, y el solver dedica su tiempo a resolver restricciones que conducen a casos útiles en lugar de dispersarse en caminos secundarios.

Este planteamiento se puede ver como una estrategia de dividir el problema: en vez de intentar resolver de una vez todo el comportamiento del binario, se formulan múltiples análisis pequeños, cada uno con un objetivo claro, como forzar una rama concreta, alcanzar un assert, comprobar una precondición o explorar un bloque de parsing. Por ejemplo, se puede fijar un buffer de entrada como simbólico, imponer una cabecera válida para atravesar los primeros checks y pedir al solver que encuentre un valor que active una ruta específica o que lleve a una operación peligrosa. De esta forma el coste computacional baja de manera notable, los resultados se obtienen antes y con mejor calidad, y aumenta la probabilidad de llegar a rutas profundas que con un análisis global quedarían fuera de alcance por la cantidad de estados intermedios y condiciones acumuladas.

5.3 INTERPRETACIÓN Y VALIDACIÓN DE RUTAS GENERADAS

Los resultados de la ejecución simbólica conviene tratarlos como una hipótesis que hay que validar. El solver puede encontrar un conjunto de valores que hace verdadera una condición en el modelo que se le ha dado, pero este por lo general es una simplificación: puede haber funciones externas aproximadas, librerías sustituidas por stubs, límites de bucles recortados o partes del estado fijadas a constantes. Por eso es importante revisar las rutas obtenidas y confirmar que el comportamiento es reproducible y tiene sentido en el programa real. La validación busca evitar dos problemas habituales: rutas que solo existen porque el modelo es incompleto y rutas que son técnicamente factibles, pero llevan a estados internos que en una ejecución normal no pueden ocurrir.

La comprobación práctica consiste en tomar los valores generados y ejecutarlos como inputs reales en el binario, preferiblemente en el mismo entorno de pruebas y con la

misma instrumentación. Si el objetivo consume un formato estructurado, hay que verificar que el input respeta la estructura completa, no solo la condición que se estaba intentando forzar. Esto incluye checks implícitos que el solver puede no conocer, como relaciones entre campos, offsets que apuntan a regiones válidas, contadores que coinciden con el número real de elementos, restricciones de alineamiento, o requisitos de codificación como UTF-8 bien formado.

Un ejemplo común es que el solver produzca un número negativo para cumplir una desigualdad en una comparación, pero que el programa, por diseño, interprete ese campo como unsigned o lo valide más tarde como no negativo, anulando la ruta o provocando un error diferente. Otro caso típico es una cadena que cumple una condición concreta, pero que luego falla porque incluye bytes prohibidos, no respeta la longitud indicada por otro campo, o rompe la coherencia semántica del formato, como un identificador que debe pertenecer a un conjunto concreto o un checksum que debe coincidir con el contenido.

Para evitar este tipo de resultados poco realistas, suele ser útil añadir restricciones adicionales en el análisis y validar iterativamente. Si al ejecutar el input real el programa rechaza la entrada por una validación posterior, esa validación se puede convertir en una nueva restricción para guiar al solver hacia entradas más coherentes. En formatos con checksums, por ejemplo, se puede modelar el cálculo o fijar el checksum a una función concreta; en formatos con campos de longitud, se pueden imponer relaciones entre longitud y tamaño real del buffer. Este ciclo de generar, ejecutar, observar dónde falla y ajustar restricciones permite pasar de soluciones teóricas a casos reproducibles que ejercitan rutas reales y, cuando el objetivo es seguridad, identificar cuáles de esas rutas conducen a fallos con impacto y bajo condiciones plausibles.

Por este motivo, una fase de validación manual o semiautomática es imprescindible para separar lo útil de las que no son viables o legítimas. La ejecución simbólica puede generar caminos que encajan perfectamente en las restricciones del modelo, pero que en la práctica no se dan porque faltan variables o requisitos que el solver no conocía, porque una librería se modeló de forma simplificada o porque el input viola reglas implícitas del formato. Validar los inputs generados permite descartar rutas artificiales, reducir el ruido y concentrar el esfuerzo en casos que realmente pueden ocurrir con entradas plausibles. Esto es especialmente importante cuando el objetivo es seguridad, ya que un camino que solo existe bajo supuestos irreales no aporta valor, mientras que un caso que reproduce un fallo con una entrada verosímil sí puede convertirse en un hallazgo relevante.

La forma más fiable de validación es ejecutar el binario real con el input producido y observar el comportamiento. En esa reproducción se comprueba si la ruta prevista se recorre de verdad, por ejemplo, verificando cobertura, logs o trazas, y si el efecto buscado aparece como se esperaba, como alcanzar una rama concreta, disparar una condición de error o llegar a una operación peligrosa. Si además aparece un fallo, se

puede evaluar su interés con evidencias: tipo de crash, informe del sanitizador, estabilidad de la reproducción y condiciones necesarias para que ocurra. Esta fase funciona como un puente entre el análisis simbólico y la confirmación empírica, y es la que convierte una solución del solver en un resultado fiable, repetible y aplicable al software tal y como se ejecuta en un entorno real.

En muchos objetivos las entradas no llegan como ficheros, sino a través de sockets, APIs, tuberías (pipes), memoria compartida u otros mecanismos de comunicación, y eso cambia la forma de validar un caso generado. En esos escenarios suele ser necesario crear pequeños scripts o herramientas que hablen el mismo protocolo que el programa, establezcan la conexión, negocien el estado inicial si lo hay y envíen los datos en el orden y con el framing correctos para reproducir el fallo. Por ejemplo, si el servicio espera primero un saludo o un comando antes del payload, no basta con pegar el input directamente, este hay que envolverlo con los encabezados adecuados, respetar longitudes, tiempos de espera y cierres de conexión. Este tipo de validación requiere a menudo desarrollos sencillos pero específicos, como un cliente mínimo en Python o un pequeño cliente de prueba orientado a red, para convertir el input del fuzzer o del solver en tráfico real y poder confirmar el comportamiento en condiciones lo más parecidas posible a las de producción.

6. SEGURIDAD Y AISLAMIENTO

El fuzzing y la ejecución simbólica exponen al software a entradas malformadas y a combinaciones de estado poco habituales que rara vez aparecen en uso normal. Esto provoca fallos latentes, cuelgues, consumo excesivo de memoria o CPU, y en algunos casos puede activar vulnerabilidades reales con efectos inesperados, como corrupción de memoria, escrituras fuera de límites o ejecución de rutas peligrosas. Además, muchas pruebas se ejecutan miles o millones de veces, lo que amplifica el impacto de cualquier comportamiento no determinista, fugas de recursos o acumulación de estado. Por ello, es fundamental aplicar medidas de seguridad y aislamiento que protejan el sistema anfitrión, eviten efectos colaterales sobre otros servicios y permitan controlar de forma estricta qué acciones puede realizar el binario bajo prueba durante la campaña.

En la práctica, el aislamiento busca que el objetivo se ejecute en un entorno acotado, con permisos mínimos y límites de recursos, de forma que un crash o un input malicioso no afecte al resto del equipo o de la infraestructura. Esto incluye separar la ejecución en máquinas virtuales o contenedores, restringir acceso a red y a disco cuando no sea necesario, usar usuarios sin privilegios, y aplicar límites como tiempo máximo por ejecución, memoria máxima y número de procesos o hilos permitidos. Con estas medidas se reduce el riesgo de que una entrada provoque bloqueos persistentes, que el programa modifique ficheros del sistema, que realice conexiones no deseadas o que afecte a otros componentes del entorno, manteniendo la campaña controlada y reproducible incluso cuando se están explorando comportamientos extremos del software.

6.1 RIESGOS DE EJECUTAR FUZZERS Y EJECUCIÓN SIMBÓLICA

El fuzzing y la ejecución simbólica consisten en alimentar una aplicación con un volumen muy alto de entradas no confiables, muchas de ellas diseñadas para forzar casos extremos y romper supuestos internos. Esto introduce riesgos directos para el sistema donde se ejecutan las pruebas y para la integridad del entorno, porque el objetivo puede bloquearse, consumir recursos de forma descontrolada o comportarse de manera inesperada al entrar en estados que no estaban contemplados. Un input puede provocar bucles muy largos, asignaciones masivas, fugas de memoria que se acumulan, tormentas de logs o creación de miles de ficheros temporales, hasta el punto de degradar o inutilizar la máquina de pruebas.

Además del impacto por consumo de recursos, existe el riesgo de efectos secundarios sobre el propio entorno. Si el binario bajo prueba tiene acceso a red o a disco, una entrada malformada podría desencadenar operaciones no deseadas, como conexiones salientes, escrituras en rutas sensibles, borrados accidentales o corrupción de datos de prueba compartidos. En objetivos con componentes concurrentes, también pueden aparecer fallos no deterministas que dejan el proceso en un estado inconsistente y generan resultados difíciles de interpretar. Cuando se exploran vulnerabilidades reales

de memoria, hay que asumir que algunas entradas podrían llegar a comportamientos equivalentes a una explotación, por lo que es importante tratar el programa como potencialmente vulnerable y ejecutar las pruebas en un entorno controlado y aislado.

Entre estos riesgos se encuentran:

- Una ejecución de fuzzing puede hacer que el objetivo acabe en estados inconsistentes o provoque bloqueos que deriven en inestabilidad del sistema. Al ejecutar miles de casos, es común que el programa entre en rutas de error poco probadas y acabe en situaciones como deadlocks, bucles que no terminan, esperas indefinidas por I/O o acumulación de recursos que no se liberan. Esto no solo afecta al proceso objetivo, también puede degradar la máquina de pruebas: picos de CPU sostenidos, consumo de memoria hasta agotar el sistema, saturación de disco por ficheros temporales o logs, e incluso inestabilidad general si el objetivo genera un número elevado de procesos o hilos. Por ejemplo, un input que fuerza una longitud enorme puede disparar una reserva masiva y provocar swapping, haciendo que el resto del sistema se vuelva prácticamente inutilizable.
- Manipular punteros, estructuras o buffers no válidos puede llevar a corrupción de memoria. Muchos de los fallos que el fuzzing busca precisamente son corrupciones de memoria, como escrituras fuera de límites, use after free o dobles liberaciones. Cuando esto ocurre, el programa puede comportarse de forma errática, fallar en puntos aparentemente no relacionados o producir resultados no deterministas. Normalmente el daño queda contenido dentro del proceso, pero si el objetivo interactúa con componentes externos, puede provocar efectos secundarios como escrituras de datos corruptos a disco, emisiones de mensajes mal formados a otros servicios o saturación de recursos compartidos. En términos de aislamiento, lo importante es asumir que, si el objetivo corrompe su memoria, deja de ser fiable y conviene reiniciarlo y limitar su capacidad de afectar al resto del entorno.
- Si el programa presenta fallos de seguridad, el fuzzer puede activarlos. El fuzzing no solo encuentra crashes, también puede desencadenar vulnerabilidades reales. Un input puede provocar lectura fuera de límites y terminar filtrando información a logs o a respuestas de red, puede activar rutas que permitan ejecución de comandos si existen llamadas peligrosas, o puede aprovechar errores de autorización que solo aparecen con combinaciones raras de estado. En entornos donde el objetivo corre con privilegios altos o tiene acceso amplio a red y disco, esto es especialmente delicado: aunque el fuzzer no intente explotar, una entrada que llega a una primitiva peligrosa puede causar efectos equivalentes, como modificar ficheros, abrir conexiones o ejecutar acciones no previstas.
- No deben utilizarse datos reales durante el fuzzing, solo datos ficticios o generados para pruebas. El corpus y cualquier semilla deben estar libres de

información sensible, porque el fuzzing tiende a copiar, mutar y redistribuir datos, y además muchas herramientas guardan inputs que provocan fallos, logs detallados y trazas que pueden terminar en sistemas de almacenamiento compartidos. Si se usan datos reales, se corre el riesgo de filtrar credenciales, tokens, datos personales o información propietaria a través de archivos de crash, reportes automáticos o simples volcados de memoria. Por ejemplo, si el objetivo procesa mensajes con cabeceras de autenticación o cookies, esos valores podrían quedar incrustados en un caso minimizado y circular por el equipo. Por eso se recomienda usar datos sintéticos, anonimizar y rotar cualquier secreto de prueba, y asegurarse de que los artefactos de la campaña se almacenan en un repositorio controlado.

Por tanto, para que el fuzzing sea seguro y útil debe ejecutarse en un entorno controlado y aislado, con permisos mínimos y límites de recursos, de forma que cualquier fallo, corrupción de memoria o consumo anómalo quede contenido y no afecte al sistema anfitrión ni a otros servicios. Esta necesidad de contención es precisamente la razón por la que no es recomendable ejecutar campañas de fuzzing en producción, donde una entrada problemática podría degradar la disponibilidad, alterar datos reales o generar efectos secundarios difíciles de revertir.

6.2 LIMITACIONES DE RECURSOS

Las campañas de fuzzing tienden a consumir muchos recursos porque fuerzan al programa a procesar entradas extremas, recorren rutas de error que no liberan recursos correctamente y repiten la ejecución miles o millones de veces. Es habitual que aparezcan picos de CPU por bucles largos, parsing costoso o descompresión, y también incrementos de memoria por asignaciones excesivas, fragmentación del heap o fugas que se acumulan durante la vida del proceso. Incluso sin que exista un bug, un fuzzer puede descubrir combinaciones de entrada que disparan casos patológicos, como estructuras profundamente anidadas, tablas enormes o parámetros que provocan trabajo cuadrático. Si no se controla, esto puede degradar la máquina de pruebas, afectar a otros servicios del sistema y sesgar los resultados al convertir la campaña en una lucha contra timeouts y falta de recursos.

Por ello es importante imponer límites explícitos que mantengan el entorno estable y que hagan el fuzzing más predecible. En la práctica se suelen fijar límites de tiempo por ejecución para evitar que una entrada monopolice la CPU, límites de memoria para impedir que el proceso agote el sistema, y restricciones sobre número de procesos o hilos para evitar tormentas de creación de tareas. También conviene controlar el espacio de disco dedicado a artefactos de fuzzing, ya que los logs, los casos que provocan fallos y los corpus intermedios pueden crecer rápidamente. Por ejemplo, si el objetivo es un parser de archivos, limitar el tamaño máximo de entrada y el tiempo de procesamiento por caso evita que un único archivo enorme o malicioso convierta la campaña en una

sucesión de ejecuciones lentas sin avance real de cobertura. Con estos límites se protege el host y, al mismo tiempo, se mejora la eficiencia de la campaña al forzar al fuzzer a centrarse en inputs que producen señal útil dentro de un presupuesto razonable.

PUBLICACIONES DE DEFENSA

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