



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

Organización del Computador II

Trabajo práctico final

SIMD Explorer

por Juan Pablo Miceli



Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (+ +54 +11) 4576-3300

<http://www.exactas.uba.ar>

Resumen

El presente informe documenta el trabajo práctico final de la materia Organización del Computador II. El mismo fue hecho con la idea de facilitar el desarrollo de software que dependiera fuertemente de algoritmos SIMD.

Para esto, se implementó un entorno similar al de Jupyter Notebooks, que permite desarrollar algoritmos de SIMD de forma interactiva, ejecutando los programas con entradas de prueba y permitiendo analizar los estados intermedios de los registros durante la ejecución. El entorno cuenta con un frontend web con ReactJS en donde el usuario escribe el código y un backend programado en Go que se encarga de ensamblar, linkear y ejecutar el programa, usando `ptrace` para analizar la ejecución de los programas.

El servidor web debe ejecutar código arbitrario creado por usuarios, por lo tanto, como medida de seguridad se implementaron mecanismos de sandboxing en el backend para evitar que usuarios maliciosos abusen del servicio. Entre estas herramientas fueron utilizadas: `seccomp`, `minijail`¹, `namespaces` y `ulimit`.

Índice

1. Introducción	2
1.1. Introducción a Assembler y SIMD	2
1.2. Introducción y problemáticas de debuggers y GDB	2
1.3. Introducción a <code>ptrace</code>	3
1.4. SIMD Explorer	3
2. Implementación de SIMD Explorer	4
2.1. Frontend	4
2.1.1. Interfaz	4
2.1.2. Comunicación con el backend	7
2.2. Backend	7
2.2.1. Inicialización de estructuras:	7
2.2.2. Procesamiento del texto recibido:	8
2.2.3. Ensamblado, linkeo e inicio del programa:	9
2.2.4. Ejecución del programa celda a celda:	11
2.2.5. Terminación del programa	12
2.2.6. Manejo de errores	12
2.3. Seguridad	12
2.3.1. Introducción a ciberseguridad	12
2.3.2. Ciberseguridad en SIMD Explorer	13
2.3.3. Linux namespaces	13
2.3.4. <code>Seccomp</code> y <code>Seccomp-bpf</code>	13
2.3.5. Berkeley Packet Filter	14
2.3.6. <code>Minijail</code>	15
2.3.7. Resumen de las medidas de seguridad tomadas	15
2.3.8. Generación de archivos	16
2.3.9. Etapa de Ensamblado	16
2.3.10. Etapa de Linkeo	21
2.3.11. Etapa de Ejecución	24
2.3.12. Denegación de servicio	30

¹<https://google.github.io/minijail/minijail0.1>

2.3.13. Docker	30
3. Conclusiones y trabajo futuro	31

1. Introducción

1.1. Introducción a Assembler y SIMD

Para la mayoría de las personas, comenzar con la programación en assembler conlleva muchas problemáticas.

Para empezar, el código resulta mucho más largo y es necesaria una mayor concentración a la hora de escribirlo. Este incremento en la dificultad inevitablemente potencia la aparición de bugs. Por si esto no fuera suficiente, ahora los mismos serán más difíciles de localizar dada la naturaleza poco intuitiva del lenguaje.

Esta situación no mejora cuando se introducen las instrucciones SIMD, las mismas se basan en un modelo de ejecución en donde, dada una instrucción, esta se aplica sobre un conjunto de múltiples datos en paralelo. Esto permite conseguir algoritmos mucho más eficientes para operaciones que se deben hacer sobre sets de datos grandes del mismo formato, donde no es necesario un modelo de cómputo secuencial, si no que es posible hacer las operaciones en paralelo. Estas instrucciones son indispensables para algoritmos que trabajan sobre imágenes o audio. Las mejoras de rendimiento, se vuelven a pagar con un incremento en la dificultad del algoritmo y por ende, del código a escribir. Obviamente esto repercute en un aumento en la frecuencia de los bugs y en la dificultad de encontrarlos.

1.2. Introducción y problemáticas de debuggers y GDB

Para tratar con los problemas presentados, contar con un buen debugger es indispensable para el desarrollo de código. Los mismos proveen una manera de ejecutar el programa deseado de forma más controlada. Por ejemplo, se puede detener la ejecución en una instrucción específica para verificar que el estado de los registros o variables sea el deseado, incluso es posible avanzar el programa instrucción por instrucción para dar un seguimiento más preciso.

El debugger más usado para esta tarea es *GDB*², en su versión más básica el mismo provee una terminal en la que mediante comandos ingresados en la consola se puede controlar el flujo del programa, como también consultar los valores que se desean conocer. Pero también es posible descargar programas externos como *GDB dashboard*³ el cual permite ver la información de todos los registros en cualquier momento y de una manera más atractiva.

Igualmente, los mismos también presentan sus inconvenientes. El más importante de ellos es que, si al momento de depurar el programa se quiere cambiar algo en el código, se deben realizar todos estos pasos:

1. Detener la ejecución en el debugger.
2. Ensamblar y linkear el ejecutable.
3. Ejecutar el nuevo ejecutable desde GDB.
4. Añadir todos los breakpoints necesarios.
5. Correr el programa.

Además, para arreglar bugs, muchas veces se debe realizar un seguimiento paso a paso de un número grande de instrucciones. Cada vez que se modifica el código tratando de arreglar un bug, este seguimiento se debe comenzar desde cero. Para algoritmos SIMD, es muy común incluso apoyarse además en papel y lápiz para realizar el mismo.

²<https://www.gnu.org/software/gdb/>

³<https://github.com/cyrus-and/gdb-dashboard>

1.3. Introducción a ptrace

*Ptrace*⁴ es un servicio ofrecido por *linux*. El mismo proporciona un medio por el que un proceso padre puede observar y controlar la ejecución de otro proceso y examinar y cambiar su imagen de memoria y registros. Se usa principalmente en la implementación de depuradores con breakpoints y en el rastreo de llamadas al sistema.

Esta herramienta cuenta con distintas operaciones muy útiles para una ejecución más personalizada, veamos ahora las 3 que serán útiles a lo largo del proyecto:

- **PTRACE_TRACEME**: Esta operación se debe hacer desde el proceso rastreado, e indica que el mismo está dispuesto a ser monitoreado por su padre.
- **PTRACE_CONT**: Reanuda la ejecución del proceso rastreado hasta el siguiente breakpoint, hasta que el programa termine o hasta que el mismo sea desalojado por el sistema operativo debido a algún error.
- **PTRACE_GETFPREGS**: Copia los registros de punto flotante, entre ellos los registros XMM, a la estructura *user_fpregs_struct* en una dirección detallada por el usuario. Dicha estructura se encuentra en la biblioteca «sys/user.h»⁵.
- **PTRACE_SETOPTIONS**: Permite modificar algunos aspectos del comportamiento de la syscall. En el uso dado para el trabajo encenderemos los flags *PTRACE_O_EXITKILL* y *PTRACE_O_TRACEEXEC*. La primera envía un *SIGKILL* al proceso rastreado si el padre del mismo termina su ejecución. La segunda lo detiene justo después de alguna llamada a *exec*.

1.4. SIMD Explorer

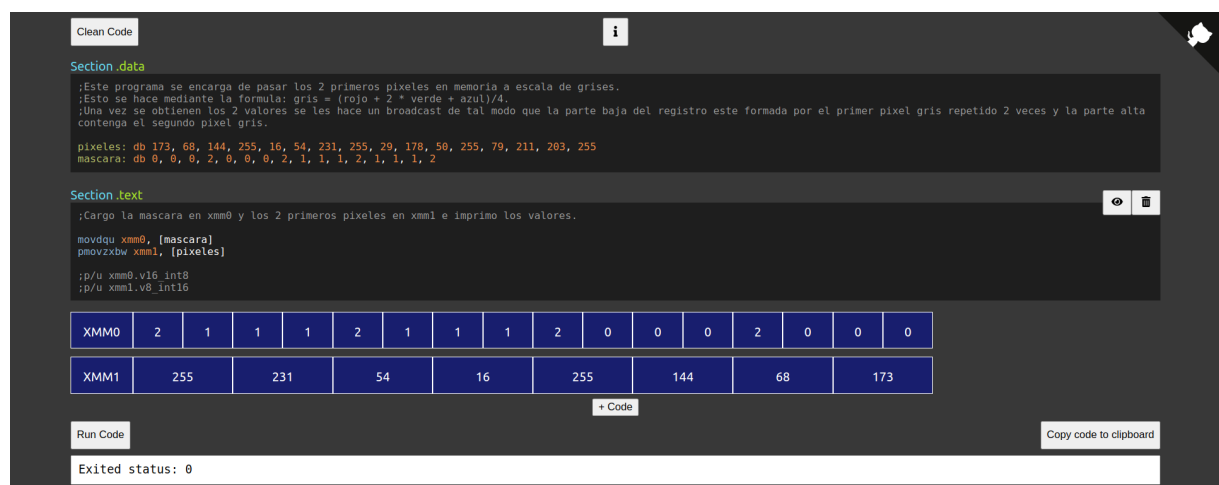


Figura 1: Interfaz de la herramienta web SIMD Explorer.

Es pensando en esas problemáticas que surge SIMD Explorer, esta es una herramienta específicamente pensada para hacer del desarrollo de código SIMD más simple y eficaz.

La misma presenta una interfaz simple como se puede ver en la figura 1, en donde se muestran celdas de texto en las cuales se pueden definir datos y código. Mientras se escribe el algoritmo de interés, se cuenta con la posibilidad de ejecutar el programa para mostrar los valores de los registros XMM deseados entre las celdas. SIMD Explorer brinda la posibilidad de separar las instrucciones agregando nuevas celdas, permitiendo hacer el análisis tan granular como se quiera.

⁴<https://man7.org/linux/man-pages/man2/ptrace.2.html>

⁵<https://code.woboq.org/qt5/include/sys/user.h.html>

Es por eso que mientras más intensivo se quiere ser con el análisis durante el desarrollo, en más celdas se debe dividir el código que se está desarrollando.

Esto resuelve los 2 problemas relatados previamente. El primero, ya que ahora lo único que se debe hacer para actualizar los cambios es ejecutar el código presionando un botón o su correspondiente shortcut de teclado. Y el problema del seguimiento queda resuelto porque ahora todos los pasos y valores intermedios del algoritmo se muestran en pantalla a la vez, por lo que solo queda leerlos y encontrar el paso donde el programa no cumple lo previsto. Parecen 2 cosas muy simples, pero esto le puede ahorrar al programador un tiempo muy significativo.

2. Implementación de SIMD Explorer

SIMD Explorer se compone de un backend programado en Go y un frontend implementado utilizando el framework ReactJS.

2.1. Frontend

Se empezará desarrollando la interfaz mostrada al usuario, y como es la comunicación de la misma con el webserver backend que es quien se encargará de todo el computo pesado. El código fuente del frontend se puede encontrar en el link presente en el pie de esta página⁶.

2.1.1. Interfaz

En este apartado se presentará la interfaz gráfica que provee el SIMD Explorer. El mismo posee 4 secciones bien definidas, que se muestran en la figura 2 y pueden ser identificadas mediante un número que las señala:

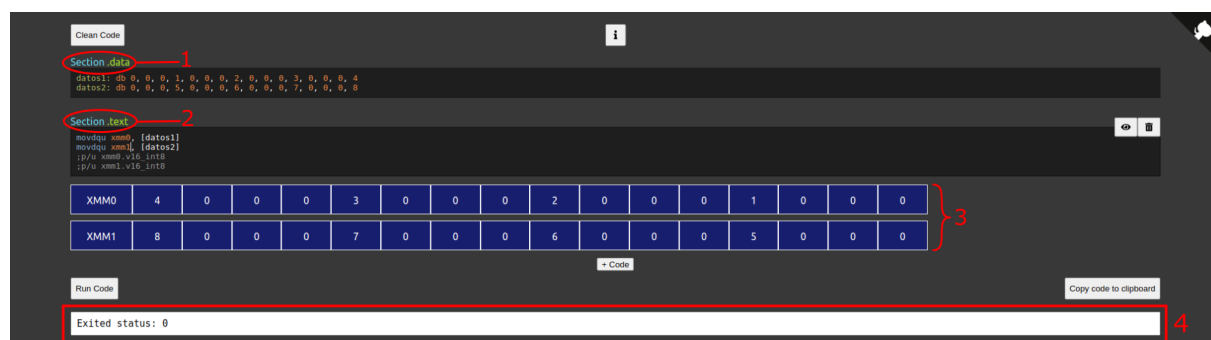


Figura 2: Diferentes secciones que componen a SIMD Explorer.

1. Sección de datos: Esta sección abarca únicamente a la primer celda de texto y en la misma se deben declarar todas las variables globales que se quieran usar a lo largo del programa.
2. Sección de texto: Aquí se escribirá el código que se desea ejecutar. Notar que esta sección puede contener tantas celdas como uno desee, a diferencia de la anterior que solo permite una de ellas.
3. Display de registros: Luego de ejecutar el programa del usuario, se le debe informar el resultado de dicha ejecución. El valor de los registros resultantes se podrá visualizar debajo de cada celda de texto.
4. Salida de la consola: La misma informará al usuario cualquier error que haya ocurrido durante la ejecución con el objetivo de que solucionar el problema sea lo más simple posible.

⁶https://gitlab.com/juampi_miceli/visual-simd-debugger/-/tree/master/frontend

Ya vistas todas las secciones que componen la interfaz es más simple entender que hacen todos los botones que se encuentran disponibles. Esto mismo lo veremos en la figura 3:

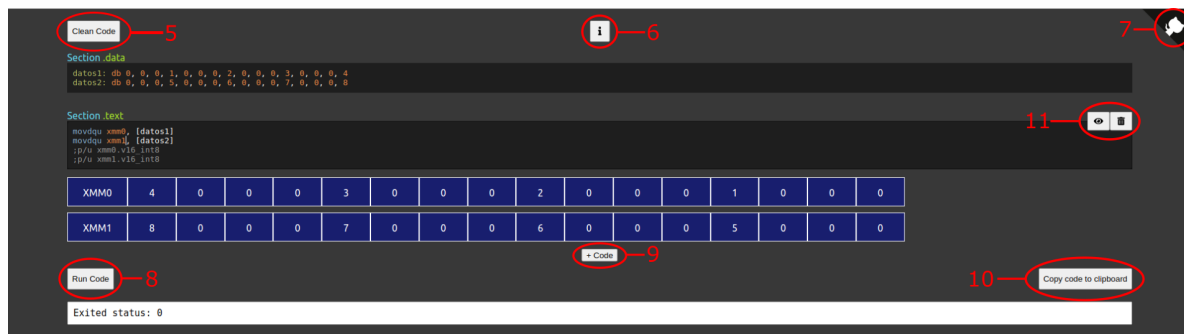


Figura 3: Diferentes botones ofrecidos por SIMD Explorer.

5. Clean Code: Permite eliminar todo el código escrito de una manera simple y rápida.
6. Help: Muestra una sección de ayuda donde se encuentran todas las respuestas a las preguntas que el usuario puede tener al usar SIMD Explorer.
7. Gitlab: Redirecciona al usuario al gitlab del proyecto SIMD explorar ya que todo el código desarrollado es abierto.
8. Run Code: Ejecuta el código ingresado y muestra la respuesta en la pantalla.
9. Add Cell: Agrega una celda de texto, este botón se encuentra en medio de todas las celdas para que se pueda agregar código en cualquier línea deseada.
10. Copy code to clipboard: Copia el código de las celdas deseadas al portapapeles para poder pegar el mismo en un archivo de texto de forma sencilla.
11. Cell Buttons: Cada celda posee 2 botones propios. El que se encuentra a la izquierda permite esconder los resultados de los registros asociados a la ejecución de la misma, además, si estos se encuentran ocultos, al guardar el código al portapapeles el contenido de esta celda no será tenido en cuenta. El botón de la derecha se utiliza para eliminar la misma.

Por último se muestran los comandos necesarios para utilizar SIMD Explorer, en la figura 4, señalados con el número «12» se pueden ver ejemplos de uso de estos comandos.

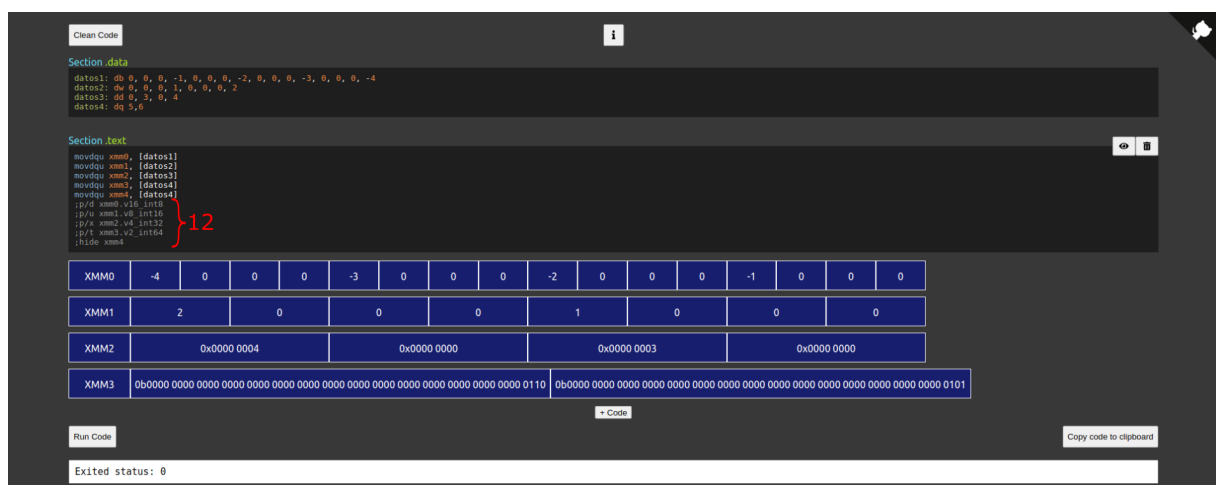


Figura 4: Ejemplo de uso de los comandos de SIMD Explorer.

Como en cada celda el programa imprime los registros que hayan sido modificados dentro de la misma, es posible que el usuario quiera realizar 2 acciones: imprimir algún registro que no haya sido modificado, o evitar la impresión de un registro que fue modificado, pero que no es de interés.

Estas opciones cuentan con un comando cada una, se muestra primero la instrucción con la cual se pueden imprimir registros:

;p/u xmm0.v8_int16

Si se quiere imprimir un registro en determinada celda, alcanza con incluir esta instrucción en esa celda especificando la base, el registro y el formato deseados de impresión. Los parámetros se especifican de la siguiente manera:

- **Base:** Base en la cual imprimir el registro. Este parámetro es opcional y es ignorado si se imprimen números flotantes. Bases posibles:
 - /d: Base 10 con signo. Esta es la base por defecto.
 - /u: Base 10 sin signo.
 - /t: Base 2 en complemento A2
 - /x: Base 16.
- **Registro:** Registro a imprimir. Los registros válidos van desde xmm0 hasta xmm15 inclusive. Este parámetro es obligatorio.
- **Formato:** Formato en el cual se subdividirá el registro. Este parámetro es obligatorio. Los formatos posibles son:
 - .v16_int8: El XMM se subdivide en 16 registros enteros de 8 bits cada uno.
 - .v8_int16: El XMM se subdivide en 8 registros enteros de 16 bits cada uno.
 - .v4_int32: El XMM se subdivide en 4 registros enteros de 32 bits cada uno.
 - .v2_int64: El XMM se subdivide en 3 registros enteros de 64 bits cada uno.
 - .v4_float: El XMM se subdivide en 4 registros de punto flotante de 32 bits cada uno.
 - .v2_double: El XMM se subdivide en 2 registros de punto flotante de 64 bits cada uno.

Se introduce ahora el uso del comando utilizado para esconder registros:

;hide xmm0

En caso que no se quiera mostrar algún registro modificado se puede agregar esta instrucción para quitar su impresión, su uso es muy simple y solo requiere un parámetro:

- **Registro:** Elegimos el registro a imprimir. Los registros validos van desde xmm0 hasta xmm15 inclusive. Este parámetro es obligatorio.

2.1.2. Comunicación con el backend

El frontend se comunica con un único endpoint ofrecido por el backend. Este es el encargado de recibir el texto plano del código ingresado por el usuario en sus respectivas celdas, y devolver los valores de los registros de interés en los puntos del programa deseado.

El llamado se realiza mediante un HTTP request de tipo POST, que es enviado al backend con un array de *CellData* transformado a JSON. Donde la estructura *CellData* es:

```
struct CellData {
    int      id          //Número de la celda
    string   code        //Código escrito en la celda
};
```

De esta manera, el backend tendrá acceso a todo el código escrito por el usuario, y además sabrá exactamente cuantas celdas hay en el código y donde está cada separación.

Enviado el request, solo resta esperar la respuesta del backend. Esta llega en formato JSON y tiene la estructura definida en el backend *ResponseObj*. La misma se muestra a continuación:

```
struct ResponseObj {
    string      ConsoleOut    //Mensaje de éxito o error generado en el backend.
    CellRegisters[] CellRegs  //Array con los registros a imprimir de cada celda.
};

typedef XMMData[] CellRegisters //CellRegisters contiene un array donde cada elemento
                                //tiene todo lo necesario para imprimir un registro XMM.

struct XMMData {
    string      XmmID        //Número del registro XMM a imprimir.
    string[]    XmmValues    //Array con los valores dentro a imprimir del registro.
};
```

Una vez se recibe el JSON, el frontend es el encargado de iterar celda por celda, y luego registro por registro, dentro de los XMM a imprimir, para mostrar todos los valores en pantalla. Estos son pasados como strings para que el frontend no tenga que encargarse de ninguna manipulación numérica con los datos obtenidos, su única responsabilidad es la de mostrar en pantalla lo recibido.

2.2. Backend

En esta sección se hablará de todos los pasos que ejecuta el backend desde que le llega el HTTP request desde el frontend, hasta que manda su respuesta HTTP con los valores deseados en el formato pedido. El código fuente del backend se puede encontrar en el link presente en el pie de esta página⁷. Durante esta sección se obviarán todos los posibles problemas de seguridad que se pueden generar al estar tratando con código de usuario en un servidor web. Todos estos detalles serán tratados en la sección de seguridad (2.3).

2.2.1. Inicialización de estructuras:

Lo primero que se debe hacer es generar la estructura *CellsData*, en la misma guardaremos el JSON que llega del frontend, además contendrá los XMM requests que se harán a *ptrace* en cada celda. Las estructuras se ven de la siguiente forma:

⁷https://gitlab.com/juampi_miceli/visual-simd-debugger/-/tree/master/backend


```

struct CellsData {
    CellData[]      Data      //Data conteniendo el código y ID de todas las celdas.
    XmmRequests[]   Requests  //Array con los registros a imprimir de cada celda.
};

struct CellData {
    int             ID        //ID de la celda.
    string          Code      //Código de la celda.
};

typedef XmmRequest[] XmmRequests //XmmRequests es un array que contiene a todos los
                                //XmmRequest dentro de una celda.

struct XmmRequest {
    int             XmmNumber //Número del registro XMM a pedir.
    string          DataFormat //Cantidad de bits de cada subregistro del XMM.
    string          PrintFormat //Base numérica para imprimir el registro XMM.
};

```

Además, es inicializada la estructura *XMMFormat* la cual contiene el valor por defecto para imprimir cada registro, es decir, en que base se debe imprimir y de cuantos bits es si alguno de los valores no se especifican. Por defecto los registros se imprimirán como enteros signados de 8 bits en base 10.

```

struct XMMFormat {
    string[]        DefaultDataFormat //Array con el formato de data de cada registro XMM.
    string[]        DefaultPrintingFormat //Array con la base a imprimir cada registro XMM.
};

```

2.2.2. Procesamiento del texto recibido:

Ahora que se cuenta con todas las estructuras, es momento de procesar el texto recibido que ya está almacenado en *CellsData*. Lo primero que se debe hacer es buscar celda a celda todos los pedidos explícitos para imprimir registros. Es decir, cada vez que el usuario escribió el comando *;print*.

Por cada uno de estos comandos encontrado, es necesario guardar el número de registro pedido, el formato numérico pedido, la base pedida y la celda en la cual se encuentra este pedido. De esta forma se podrán agregar todos los requests a la estructura de datos *XmmRequests* explicada anteriormente.

Luego de esto, se agrega al principio de la primer celda la siguiente línea:

```
section .data
```

Esto se hace para respetar lo visto en la interfaz, donde la primer celda representaba la sección de datos. Además, como solo hay una única celda que corresponde a esa sección, previo al texto de la segunda celda se deben agregar las siguientes líneas para definir la sección de texto y el inicio del programa.

```

global _start
section .text
_start:

```

Mediante esas líneas se están realizando varias acciones, primero se está haciendo global el símbolo *_start*. El mismo es usado por el linker para encontrar el inicio del programa, esta es la dirección donde

el sistema operativo colocará el contador de programa al momento de correrlo. Por lo que si el mismo no es global, el linker no será capaz de encontrarlo.

Luego, se está inicializando la sección de texto, por lo que todo lo que se encuentre por debajo de esta línea será tratado como código.

Por último, se define el símbolo `start`, de esta manera, al correr el programa el código se empezará a ejecutar en la primer línea escrita por el usuario.

Hecho esto, se deben agregar los breakpoints entre las celdas, para que *ptrace* sea capaz de conseguir los valores de los registros XMM. Para esto, basta con agregar al final de cada celda de texto la siguiente línea.

```
int 3
```

Cuando el programa del usuario se esté ejecutando y se llegue a esta instrucción, se generará una *SIGTRAP*. En este momento volverá el control al webserver Go, y se podrán obtener todos los valores deseados. Luego de esto se reanudará la ejecución desde la instrucción inmediatamente siguiente a *int 3*, o sea, la celda de código siguiente.

Lo último que se debe añadir al texto enviado por el usuario es la terminación del programa para que finalice limpiamente. Esto es tan simple como agregar las siguientes líneas al final de la última celda.

```
mov rax, 60
mov rdi, 0
syscall
```

De esta forma, se llama a la *syscall 60 (exit)*⁸ con el parámetro 0. Lo cual indica que el programa terminó sin inconvenientes.

2.2.3. Ensamblado, linkeo e inicio del programa:

Ahora que ya ha sido procesado el código de todas las celdas, solo resta crear un archivo de texto, que por el momento tendrá por nombre **out.asm**. En el mismo se escribirá todo el texto relatado en la sección de procesamiento del texto recibido (2.2.2).

Ahora solo resta seguir los pasos que se harían si quisiera ejecutar ese código en una computadora propia. Es decir, el archivo de texto primero debe ser ensamblado a un archivo objeto, y luego este debe ser linkeado a un archivo ejecutable.

Para esto, se usará la abstracción de Go de la *syscall 59 (exec)*⁹ provista en el paquete *exec*¹⁰, la misma permite ejecutar programas externos desde Go. En este trabajo, será necesario ejecutar 3 programas a lo largo de todo el request.

1. NASM¹¹: Será quien ensamble el archivo de texto (**out.asm**) en un archivo objeto (**out.o**).
2. LD¹²: Con este programa se linkeará el archivo objeto (**out.o**) para conseguir un ejecutable (**out**).
3. OUT: Por último, se ejecuta **out** para poder analizarlo de la forma pedida por el usuario.

Para poder correr *NASM* desde Go se deben ejecutar las siguientes líneas:

```
nasmlCmd = exec.Command("nasm", "-f", "elf64", "out.asm", "-o", "out.o")
err = nasmlCmd.Run()
```

Esto ejecutará el ensamblador y el webserver detendrá su ejecución hasta que *NASM* no haya finalizado. En caso de haber un error, se notificará mediante la variable *err* y *stderr* y se tomarán acciones correspondientes, explicadas en la sección de manejo de errores (2.2.6). Por último, es momento de analizar el significado de cada uno de los flags pasados al ensamblador.

⁸<https://man7.org/linux/man-pages/man3/exit.3.html>

⁹<https://man7.org/linux/man-pages/man3/exec.3.html>

¹⁰<https://golang.org/pkg/os/exec/>

¹¹<https://www.nasm.us/>

¹²<https://man7.org/linux/man-pages/man1/ld.1.html>

- -f: Este flag se utiliza para elegir el formato del archivo objeto. En este caso ese formato es ELF64.
- -o: Se utiliza para elegir el nombre del archivo de salida. Como se mencionó antes, el mismo será **out.o**

Una vez hecho esto, se puede pasar a linkear el programa con *LD*, el código será el mismo que fue visto para el ensamblado, pero en este caso serán usados los parámetros necesarios para linkear el programa.

```
linkingCmd = exec.Command("ld", "-nostdlib", "-static", "-o", "out", "out.o")
err = linkingCmd.Run()
```

Por segunda ocasión, el webserver Go debe esperar a que el programa externo termine su ejecución para reanudar su propio hilo. En este caso, ese programa será el encargado de linkear el archivo objeto. Los errores serán tratados de la misma forma que en la etapa de ensamblado. En el comando de ejecución nuevamente son utilizados flags para lograr correr el linker con una configuración más personalizada. Se detalla a continuación la funcionalidad de cada uno de estos parámetros.

- -nostdlib: Este flag detiene al linker de buscar bibliotecas que no hayan sido explícitamente pedidas en el comando. En este caso no se pide ninguna biblioteca por lo que a priori el programa no debería tener nada mapeado en memoria más allá de lo necesario para el mismo proceso.
- -static: Este flag asegura que *LD* no linkee ninguna biblioteca compartida.
- -o: Se utiliza para elegir el nombre del archivo de salida. Como mencionamos antes, el mismo será **out**

Por último se debe ejecutar **out**, pero en este caso es necesario que se pueda mantener una interacción asíncrona entre el webserver Go y el programa del usuario que es corrido bajo *ptrace*. Ambos programas serán corridos en un hilo de ejecución diferente de forma concurrente, por lo que el segundo debe ser capaz de informarle al primero el momento en el que haya llegado a una interrupción o haya finalizado, como también enviarle el valor de los registros pedidos ya que estos deben ser enviados de nuevo al usuario. Es por esto que el mismo se ejecutará de la siguiente forma:

```
exeCmd = exec.Command("out")
exeCmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}
startErr = exeCmd.Start()
exeCmd.Wait()
exePID = exeCmd.Process.Pid
optErr = syscall.PtraceSetOptions(exePID, syscall.PTRACE_O_EXITKILL|syscall.PTRACE_O_TRACEEXEC)
```

Lo primero que se puede ver, es que en la segunda línea se agrega una instrucción nueva. La misma se encargará de avisarle al proceso hijo, en este caso el ejecutable del usuario, de ejecutar la operación *PTRACE_TRACEME* una vez el mismo inicie. Además, en la línea siguiente se cambia la función *Run()* por la función *Start()*, esta inicia el proceso hijo y continua la ejecución de ambos procesos de forma concurrente.

Luego de eso se debe esperar a que el hijo esté «listo», esto significa que haya sido correctamente cargado en memoria y que esté detenido en la primer instrucción listo para seguir corriendo. Para eso se usa la función *Wait()* que espera hasta que el proceso hijo termine o este detenido, en este trabajo, se espera el segundo caso. Una vez ese proceso existe y está listo para ser ejecutado se le setean las opciones con la función *PtraceSetOptions*.

Como siempre, se deben tratar correctamente los errores presentes en *startErr* y *optErr*, se hablará de eso en la sección de manejo de errores (2.2.6).

2.2.4. Ejecución del programa celda a celda:

En este momento se inicializará y empezará a llenar la estructura *responseObj* presentada en la sección de comunicación con el backend (2.1.2), la misma será llenada con los registros XMM que sean requeridos en un ciclo que recorrerá todas las celdas.

Como también serán impresos los registros que hayan sido modificados entre el inicio y el fin de una celda, antes de entrar en el ciclo se debe obtener el valor de todos los registros XMM mediante la operación *PTRACE_GETFPREGS*. Esto se hace para dentro del ciclo comparar si los XMM siguen iguales luego de ejecutar la celda actual. Una vez se tienen los registros, se guardarán en *oldXMMHandler*, variable que nos facilitará la manipulación del set de registros más adelante y tendrá la siguiente estructura:

```
struct XMMHandler {
    XMM[]      Xmm      //Array con los 16 registros XMM.
};

typedef uint8[] XMM      //Cada XMM es un array de 16 enteros sin signo de 8 bits.
```

El ciclo tendrá la siguiente forma:

```
int i = 0
while running(pid) do:
    newXMMHandler = getXMMRegs(pid)

    requestedRegisters = getRequestedRegisters(cellRequests[i], newXMMHandler, xmmFormat)
    changedRegisters   = getChangedRegisters(oldXMMHandler, newXMMHandler, xmmFormat)
    finalRegisters     = joinWithPriority(requestedRegisters, changedRegisters)

    append(response.CellRegs, finalRegisters)

    oldXMMHandler = newXMMHandler

    ptraceCont(pid)

    wait(pid)
    i++
```

Lo primero a realizar es guardar el estado de los registros XMM en la variable *newXMMHandler*. Esto se hace mediante un wrapper que internamente llama a la syscall *ptrace* con la operación *PTRACE_GETFPREGS* y convierte el valor devuelto a la estructura querida (*XMMHandler*).

Luego, haciendo uso de los requests conseguidos en la sección de procesamiento del texto recibido (2.2.2) (*cellRequests[i]*), el formato de impresión por default (*xmmFormat*) y los datos presentes en los registros XMM (*newXMMHandler*), es posible guardar los registros en el formato y base numérica deseados que el usuario solicitó en esa celda. Estos, son almacenados en la variable *requestedRegisters* de tipo *CellRegisters*.

Además de eso, también se imprimen los registros que hayan sido modificados en la ejecución de la última celda. Para esto, es necesario tener los valores antes de ejecutar la celda (*oldXMMHandler*), y en el momento actual (*newXMMHandler*) ya que estos deben ser comparados uno a uno. En el caso en el que se encuentre con diferencias será agregado a los registros cambiados. En este caso, ya que el usuario no pidió explícitamente la impresión del registro, no se cuenta con el formato deseado de impresión. Aquí, debemos hacer uso de los formatos por defecto provistos en *xmmFormat*. Una vez conseguidos estos valores, se almacenan en la variable *changedRegisters*, también de tipo *CellRegisters*.

Por último, es necesario atender el caso en el que el usuario explícitamente haya pedido imprimir un registro que fue modificado en la última celda, ya que de no hacer esto el registro se imprimiría 2 veces. Una vez porque lo pidió el usuario y otra vez porque el registro fue cambiado. En este caso solo se quiere realizar la impresión pedida por el usuario. Es por esto que se juntan los requests conseguidos en las 2 líneas anteriores en la nueva variable *finalRegisters*. El criterio para ver si un registro entra en *finalRegisters* es el siguiente:

- Todos los registros presentes en *requestedRegisters* entran primero.
- Luego se intenta meter uno a uno los registros presentes en *changedRegisters*. Si el registro a insertar ya se encuentra en *finalRegisters*, no lo insertamos y se avanza al siguiente registro. Caso contrario, el mismo es agregado.

Una vez conseguidos estos registros son agregados a lo que será la respuesta que que será enviada al cliente (*response.CellRegs*).

Luego de esto, ya que no es necesario hacer nada más con los registros XMM conseguidos mediante *ptrace*, se puede actualizar *oldXMMHandler* con el nuevo valor.

Ya terminando, solo resta continuar la ejecución del código hasta el siguiente breakpoint mediante la operación *PTRACE.CONT*, es importante esperar a que la misma se detenga utilizando la syscall *wait()*. Ya que de no hacer esto, lo más probable es que se ejecute la próxima iteración del ciclo antes de que el proceso hijo esté listo para ser leído por el padre. Lo cual inducirá a una lectura incorrecta o incluso a una detención del programa.

2.2.5. Terminación del programa

Una vez se termina el loop, esto debería querer decir que el programa terminó. El caso en el que no lo haga será analizado en la sección de manejo de errores (2.2.6). Lo último que resta hacer ahora es guardar el estado de salida del programa del usuario.

Por último, se deben borrar los 3 archivos creados para el usuario: el de texto, el objeto y el ejecutable, para luego, limpiar el filesystem, devolver al cliente el *response* creado en la sección de ejecución celda a celda (2.2.4) y finalizar la ejecución.

2.2.6. Manejo de errores

Hasta este momento, nunca se tuvieron en cuenta los casos en que alguno de todos los pasos del backend fallen, claramente, esto es algo que no es posible obviar en el mundo real. Primero, por las buenas prácticas de programación en general, pero especialmente porque para que el programa termine sin ninguna falla, se debe suponer que el código del usuario no tiene ningún error, lo cual es sabido que es bastante infrecuente.

Los errores más fáciles de manejar son los que ocurren al principio de la ejecución, estos no dependen del usuario y se pueden dar por algún error de comunicación entre el servidor y el cliente o por un error creando el archivo de texto en el webserver, por ejemplo si este lee un JSON cuya estructura no reconoce. Aquí lo único que se puede hacer es informar al cliente el error correspondiente mediante el campo *ConsoleOut* del *response* y terminar la ejecución.

Una vez se crea algún archivo, a lo anterior se debe sumar el paso de borrar los archivos creados, ya sea que solo se haya creado el archivo de texto, o los 3 archivos usados. Este caso se puede dar por ejemplo si la creación del archivo de texto es exitosa, pero falla su ensamblado. En ninguno de estos casos es permitido dejar archivos huérfanos en el webserver ya que esto con el tiempo lo llevaría a consumir todo su espacio disponible en el disco.

Por otra parte, si se ha llegado a iniciar el código del usuario y se genera algún error durante la ejecución del mismo, además se debe terminar el proceso, dado que de no hacerlo el servidor iría acumulando procesos «zombie» que no dejarían de ocupar memoria hasta que el mismo sea reiniciado.

Es por esto que una vez se termina el ciclo hablado en la sección de ejecución celda a celda (2.2.4) debe chequearse si en verdad el proceso asociado al usuario se finalizó. Si este se encuentra vivo una vez terminado el ciclo, se lo termina y se informa al usuario de esta situación.

2.3. Seguridad

2.3.1. Introducción a ciberseguridad

En prácticamente todo proyecto que brinde un servicio online se debe pensar en el aspecto de la seguridad. Esto se debe a que potencialmente millones de personas en simultáneo tendrán acceso al servicio.

Por este motivo, es incorrecto suponer que ninguna de ellas tendrá intenciones de generar un ataque y por eso es necesario preparar la mayor cantidad de barreras que sean posibles.

Mientras más grande sea el universo de entradas del programa, más vulnerabilidades tendrá el mismo y por ende más fácil le será a un usuario malicioso encontrarlas. Por este motivo, en estos casos los esfuerzos por encontrar y arreglar esas vulnerabilidades deben amplificarse.

Esta tarea no es trivial en lo absoluto, es por esto que todas las empresas desarrolladoras de software, por más grandes que sean deben publicar nuevas actualizaciones de sus productos con parches que arreglen estas vulnerabilidades que seguramente habían estado en el programa mucho tiempo, pero que fueron descubiertas en la actualidad.

2.3.2. Ciberseguridad en SIMD Explorer

Uno de los objetivos de SIMD Explorer es poder ofrecerlo como un servicio web para que la tarea de programar algoritmos SIMD sea más simple para cualquier persona en el mundo. Esto implica que el servidor en donde esté alojado el servicio de SIMD Explorer debe ensamblar los programas enviados por los usuarios y ejecutarlos. Esto contrae un gran riesgo de seguridad para el servidor, ya que un usuario malicioso podría querer abusar del sistema y obtener control sobre el servidor. A continuación se hará un análisis de los posibles problemas de seguridad que presenta cada parte del sistema y cómo son protegidos. Cada llamada del usuario al servidor puede ser separada en 4 etapas principales:

1. Generación de archivos
2. Etapa de ensamblado
3. Etapa de linkeo
4. Etapa de ejecución

Se presentarán una a una estas etapas más adelante, pero primero se explicarán algunos conceptos que serán de utilidad conocer para el mejor entendimiento del informe:

2.3.3. Linux namespaces

Los namespaces de linux son una prestación del kernel que divide los recursos de modo que un conjunto de procesos ve un conjunto de recursos, mientras que otro conjunto de procesos ve un conjunto diferente de recursos. De esta forma, los procesos del primer grupo funcionarán de manera aislada, haciendo parecer que tienen sus propios recursos de hardware e impidiendo que estos interfieran con procesos fuera de su namespace, asimismo tampoco podrán ser visibles para procesos externos.

Existen muchos tipos de namespaces, pero en este trabajo solo nos enfocaremos en 2:

- PID namespace: Proveen aislamiento sobre el ID de cada proceso en el espacio. Esto quiere decir que dos procesos que vivan en distintos namespaces no tendrán forma de conocer la existencia del otro. Esto permite que estos procesos puedan tener el mismo ID. Gracias a este aislamiento es posible denegar a procesos peligrosos la capacidad de rastrear o atacar a procesos vulnerables que se encuentren fuera de su grupo.
- Mount namespace: Proveen aislamiento a los puntos de montaje vistos por los procesos dentro del namespace, por ende, cada conjunto de procesos verá jerarquías de archivos distintas. Esta herramienta es fundamental para crear virtual filesystems (VFS) y evitar que procesos peligrosos vean archivos que no le son necesarios.

2.3.4. Seccomp y Seccomp-bpf

Seccomp es un servicio que provee linux que permite limitar las llamadas al sistema que un proceso puede realizar. Originalmente se propuso como una herramienta para que un proceso se ejecute de forma que solo se pueden realizar las syscalls *exit*, *sigreturn* y, sobre archivos ya abiertos realizar *read* y *write*. Este servicio proveía un fuerte aislamiento pero es poco útil debido a que es demasiado limitante. Es por

esto que surgió *seccomp-bpf*, este es una extensión de la herramienta anterior que permite definir una política de filtrado de syscalls para los procesos. Es así que se puede bloquear cada llamada al sistema de manera particular permitiendonos además personalizar estos bloqueos.

Con esta herramienta, para cualquier syscall arbitraria se pueden realizar 4 acciones:

- Permitir: Es como si el filtro no existiera, la llamada al sistema es ejecutada con normalidad.
- Denegar: Cuando el proceso intente llamar a la syscall en cuestión será inmediatamente detenido y terminado mediante un *SIGKILL*.
- Permitir pero no ejecutar: Cuando el proceso llame a la syscall, este no será terminado como en el caso anterior, pero tampoco se ejecutará la llamada al sistema. En su lugar se ejecutará una función «dummy» cuyo único propósito es simular que se ha pasado por la syscall. Por lo tanto, lo único que esta función hace es retornar un código exitoso (o fallido según se necesite) al proceso que la invocó. De esta forma se puede evitar que un proceso haga acciones peligrosas, pero aún así ejecutar el resto del programa que podría ser de utilidad.
- Permitir la ejecución de forma condicional: Antes de ejecutar la syscall deseada, se evaluarán los argumentos recibidos para ver que coincidan con argumentos avalados. Es decir, la llamada al sistema se ejecutará si y solo si lo hace como se definió en el filtro, en caso que la llamada no cumpla estos requisitos, el proceso será terminado como si la syscall hubiera sido denegada. Esto permite limitar en gran medida el peligro que estas llamadas tienen, y nos podrían permitir asegurar que el usuario no pueda abusar de estas syscalls.

Estas acciones se deben especificar mediante una política de filtrado, la misma se expresa en un lenguaje llamado *Berkeley Packet Filter* y de ahí el nombre de la nueva herramienta resulta *seccomp-bpf*.

2.3.5. Berkeley Packet Filter

Se considera que una explicación más amplia de *BPF* puede ser de gran ayuda a la hora de entender como funciona el filtro realizado para este trabajo. Cuando se trabaja con esta sintaxis, *BPF* define una máquina virtual que puede ser implementada dentro del kernel. Esta maquina virtual posee un pequeño set de instrucciones, definido por códigos de operación de tamaño fijo y único.

Estas instrucciones no permiten que se ejecuten loops de ningún tipo, ya que el tamaño de los saltos condicionales debe ser positivo. Debido a esto, el flujo de ejecución del programa se puede definir como un grafo dirigido acíclico. Esto nos da garantías de que el programa termina, lo cual es imprescindible ya que está corriendo dentro del kernel, y si este se colgara el resultado sería catastrófico. Para además asegurar que la ejecución termina de forma veloz, el número máximo de instrucciones aceptadas es de 4096.

Esta máquina virtual, cuenta con un registro acumulador, un área de datos (que en el contexto de *seccomp* contendrá a la syscall de entrada junto con el valor de sus parámetros) y un contador de programa, simple ya que todas las instrucciones son del mismo tamaño.

Cada instrucción es almacenada en una estructura de tipo «*sock_filter*» definida en la biblioteca *linux/filter.h*¹³ y su formato se puede ver a continuación:

```
struct sock_filter    /* Bloque de instruccion */
{
    __u16 code;      /* Código de operación */
    __u8 jt;         /* Salto en condición verdadera */
    __u8 jf;         /* Salto en condición falsa */
    __u32 k;         /* Campo de usos múltiples */
};
```

En la VM se cuenta con instrucciones de carga y almacenamiento (load and store), así también como instrucciones de salto, de aritmética simple y de retorno. Las instrucciones de utilidad para este trabajo son las siguientes:

¹³<https://android.googlesource.com/platform/external/kernel-headers/+donut-release/original/linux/filter.h>

- **Load:** Para cargar el valor de la syscall o de alguno de sus parámetros desde el área de datos hacia el registro acumulador.
- **Jump:** Existen tanto saltos condicionales como incondicionales. En el contexto de *seccomp-bpf* se hará uso únicamente de saltos condicionales de tipo *JEQ*, es decir, saltos que se ejecutan si existe una igualdad entre el registro acumulador y el inmediato *k*.
- **Return:** Existen muchas instrucciones de retorno posible. Para este trabajo importan únicamente las instrucciones *ALLOW* y *KILL* para permitir la ejecución de la syscall, o terminar el proceso mediante un *SIGKILL* respectivamente.

Por último, veamos el formato del área de datos, la misma está definida por la estructura «*seccomp_data*» presente en la biblioteca *seccomp.bpf.h*¹⁴ escrita por Will Drewry y Kees Cook y se puede ver a continuación:

```
struct seccomp_data          /* Área de datos*/
{
    int nr;                  /* Número de syscall*/
    __u32 arch;              /* Arquitectura del procesador */
    __u64 instruction_pointer; /* Contador de programa */
    __u64 args[6];           /* Parámetros recibidos por la syscall */
};
```

2.3.6. Minijail

*Minijail*¹⁵ es una aplicación desarrollada por Chrome OS para ejecutar procesos de manera más segura. Entre otras cosas, este programa utiliza por debajo linux namespaces y *seccomp-bpf*. Como el uso de las herramientas nombradas no es simple, el objetivo de *minijail* es proveer una abstracción de las mismas para que su uso sea lo más sencillo posible y estas medidas de seguridad sean implementadas de forma veloz y sin errores que hagan que el tiempo invertido en seguridad haya sido en vano. En este trabajo se utilizó *minijail* para maximizar la seguridad en los procesos peligrosos.

2.3.7. Resumen de las medidas de seguridad tomadas

Los peligros a enfrentar en las 4 etapas siguientes pueden agruparse en dos grandes grupos:

- **Código peligroso:** Procesos ejecutando código propio, o indeseado, como por ejemplo código del kernel o aplicaciones no permitidas.
- **Acceso indebido:** Procesos teniendo capacidad de lectura o escritura a archivos que no les incumben, como ser contraseñas, información de otros usuarios o binarios del servidor.

Veamos de manera muy resumida que se va a hacer en cada etapa para mitigar estos peligros:

- **Generación de archivos:** Cada usuario tendrá su propia carpeta cuyo nombre será creado de forma aleatoria y encriptada, lo cual ayudará a aislar sus archivos para que ningún otro usuario pueda leerlos o modificarlos. Además al encriptar el nombre de los archivos se disocia al usuario de la información recibida.
- **Ensamblado:** Con la ayuda de *minijail* se creará un VFS para reducir al máximo los archivos visibles por *NASM* así como también se creará un PID namespace para evitar que el programa interactúe con procesos externos. Por último se limitará la cantidad de syscalls que este programa podrá realizar, dejando únicamente las que son absolutamente necesarias y bloqueando todas las restantes.

¹⁴<https://outflux.net/teach-seccomp/step-2/seccomp-bpf.h>

¹⁵<https://google.github.io/minijail/>

- Linkeo: El plan de acción es el mismo que en la etapa de ensamblado, pero en este paso se abren nuevas posibilidades a la hora del bloqueo de llamadas al sistema.
- Ejecución: Se desarrollará un programa propio que utilice los filtros *seccomp-bpf* para prohibir todas las syscalls con el objetivo de profundizar los conocimientos sobre este servicio. Esta aplicación será la encargada de ejecutar el binario del usuario de forma segura. Denegar todas las llamadas al sistema no será del todo posible por lo cual será necesario tomar recaudos extras en la implementación del mismo. Igualmente, el número tan limitado de syscalls permitidas hace posible no tener que preocuparnos en implementar namespaces de ningún tipo.

2.3.8. Generación de archivos

Lo primero que se creará para el usuario será una carpeta con un nombre generado de manera aleatoria con ayuda del paquete «crypto/rand»¹⁶ provisto por Go. Este nombre aleatorio se usará también en los archivos generados en las próximas etapas. Generar el nombre de la carpeta de esta forma dificultará al atacante el poder adivinar o predecir la ubicación de los archivos generados para otros usuarios, esta será la primer técnica que se utiliza en SIMD Explorer para proveer aislamiento entre usuarios.

Además, para evitar que un usuario malicioso saturé el espacio de almacenamiento del servidor, o simplemente haga llamadas muy pesadas el tamaño de los archivos generados estará restringido a 30KiB. Cabe aclarar que este es el tamaño máximo que podrán tener, tanto el archivo que se generará con el texto ingresado por el usuario como el archivo luego del ensamblado y el que se consigue luego del linkeo. Esto es necesario ya que mediante macros se pueden generar archivos muy pesados luego de ensamblar archivos de texto livianos. Por último, tan pronto termine la ejecución se elimina la carpeta que contiene todos los archivos del usuario para evitar el consumo de espacio en disco innecesario.

2.3.9. Etapa de Ensamblado

Para ensamblar el archivo de entrada se utiliza *NASM* con los parámetros vistos en la sección de ensamblado, linkeo e inicio del programa (2.2.3). Dado que el usuario controla casi en totalidad el archivo de entrada que se va a ensamblar, es necesario tener cuidado de que los problemas de seguridad que tenga el mismo ensamblador no sean vulnerados.

Una primer problemática sería que el usuario tenga acceso al filesystem del servidor, ya que en el mismo se encuentran archivos muy delicados, como por ejemplo el ejecutable del propio servidor que puede ser modificado. Estos privilegios los puede obtener tanto mediante llamadas a sistemas, o mediante directivas del mismo *NASM*.

Lo primero a notar, es que mediante directivas como *%include* o *incbin* *NASM* puede acceder a archivos sin ningún inconveniente, esto es un enorme problema ya que le estamos dando al usuario permisos de lectura a absolutamente todos los archivos del servidor, entre ellos la información de otros usuarios, contraseñas y otros archivos que se deseen tener ocultos. Este problema no puede ser paleado mediante un parseo del archivo, ya que debido al poder de los macros un usuario podría escribir la directiva de manera que sea imposible detectarla en un simple análisis del texto de entrada. Como no se tiene intención de bloquear tajantemente los macros, es necesario encontrar una solución alternativa.

Lo segundo es que si el usuario logrará de alguna manera explotar un bug del ensamblador y ejecutar código propio, el mismo podría hacer lo que quisiera, como espiar a otros procesos, leer archivos y lo que sería peor, modificarlos. De esta forma el atacante podría corromper archivos de configuración o el binario que se encarga de correr el servidor. Luego, bastaría con forzar un reinicio del mismo para que el webserver ejecute código escrito por el atacante.

Es aquí que se comienza con el uso de *minijail*. Para empezar a aprovecharlo, lo único que se debe hacer es embeber el comando para ejecutar a *NASM* dentro del nuevo comando que ejecutará a *minijail*. Se muestra como queda esto en el código a continuación.

¹⁶<https://golang.org/pkg/crypto/rand/>

```
nasmCmd := exec.Command("minijail0",
    "-p",
    "-n",
    "-S", "../policies/nasm.policy",
    "-v",
    "-P", "/var/empty",
    "-b", "randomClientFolder/",
    "-b", "/usr/bin/",
    "-b", "/proc",
    "-r",
    "nasm", "-f", "elf64", "randomFile.asm", "-o", "randomFile.o")
```

Se desarrolla ahora flag a flag que es lo que se está ejecutando:

- -p: Corre *NASM* dentro de un nuevo PID namespace. Esto hará imposible que el proceso que corre ejecuta al ensamblador pueda ver o afectar a otros procesos que no sean descendientes del mismo.
- -n: Enciende el bit «no_new_privs» del proceso que corre a *NASM*. Esto impide al proceso y a todos sus descendientes el obtener nuevos privilegios hasta el fin de su ejecución.
- -S: Recibe el archivo que tiene las políticas de *seccomp-bpf* para limitar el uso de syscalls. Se mostrará más adelante el contenido de este archivo.
- -v: Corre *NASM* dentro de un nuevo VFS namespace. Esto hará que los puntos de montaje del proceso sean independientes de los puntos de montaje del sistema.
- -P: Cambia la raíz del virtual filesystem a **/var/empty** mediante «pivot_root». Esto limita enormemente el acceso que tiene el ensamblador sobre los archivos del sistema.
- -b: Enlaza un directorio por fuera del virtual filesystem dentro del mismo. Es posible además especificar si queremos que ese directorio tenga permisos de escritura. En este caso es necesario acceder a «usr/bin» para poder leer el binario del mismo *NASM*. También será imperioso acceder a la carpeta generada para el usuario que en primera instancia no es creada en el virtual filesystem, este directorio debe ser enlazado ya que en el mismo se leerá el archivo de entrada para luego escribir el archivo objeto, es por eso que también es necesario que esta carpeta tenga permiso de escritura. Por último, se debe enlazar la carpeta «/proc».
- -r: Remonta «/proc» como read only. Esto provee seguridad extra sobre el mount de la línea anterior.

Queda entonces ver cuales son las llamadas a sistema que puede ejecutar *NASM*. Para limitar el acceso a estas llamadas *minijail* provee una sintaxis de alto nivel en donde se debe ingresar la llamada a sistema que queremos junto con su permiso de acceso. Es importante aclarar que este filtro se aplicará luego de haberse iniciado la libc, esto es muy útil ya que la cantidad de llamadas al sistema que se hacen únicamente para iniciar la libc es enorme y gran parte de ellas son muy peligrosas. Para tener una idea visual de todo lo que pasa siquiera antes de empezar la ejecución de un programa, veamos que pasa si hacemos un seguimiento de las llamadas a sistema de un programa en C que contiene un *main* vacío. Es decir que lo único que realiza es inicializar la libc. Para realizar este análisis fue usado el programa *strace*, el cual se encuentra en Linux y fue desarrollado específicamente para ver las syscalls realizadas por un proceso. Los resultados de correr *strace* sobre el programa en C vacío son los que se muestran en la figura 5.

```
(juampi|~/Desktop)> strace -c ./dummy
```

% time	seconds	usecs/call	calls	errors	syscall
0,00	0,000000	0	1		read
0,00	0,000000	0	2		close
0,00	0,000000	0	2		fstat
0,00	0,000000	0	7		mmap
0,00	0,000000	0	4		mprotect
0,00	0,000000	0	1		munmap
0,00	0,000000	0	1		brk
0,00	0,000000	0	6		pread64
0,00	0,000000	0	1	1	access
0,00	0,000000	0	1		execve
0,00	0,000000	0	2	1	arch_prctl
0,00	0,000000	0	2		openat
100.00	0,000000		30	2	total

Figura 5: Resumen de las llamadas a sistema ejecutadas únicamente por la lib.

Se puede ver que en total son 30 las llamadas a sistema, entre las cuales se encuentran syscalls para acceder a archivos, privilegios y memoria. Se puede hacer este mismo estudio al correr *NASM* con *strace*, se entenderá entonces que son necesarias todas llamadas que existan en *NASM* que no se hayan visto hasta el momento, como así todas aquellas cuyo número de usos (identificable en la columna «calls») haya aumentado.

Veamos ahora el resultado de ejecutar *NASM* bajo la lupa de *strace* en la figura 6.

```
(juampi|~/Desktop)> strace -c nasm -f elf64 output.asm -o output.o
```

% time	seconds	usecs/call	calls	errors	syscall
0,00	0,000000	0	3		read
0,00	0,000000	0	1		write
0,00	0,000000	0	8		close
0,00	0,000000	0	13		fstat
0,00	0,000000	0	5		lseek
0,00	0,000000	0	14		mmap
0,00	0,000000	0	4		mprotect
0,00	0,000000	0	8		munmap
0,00	0,000000	0	6		brk
0,00	0,000000	0	6		pread64
0,00	0,000000	0	1	1	access
0,00	0,000000	0	1		execve
0,00	0,000000	0	2	1	arch_prctl
0,00	0,000000	0	8		openat
100.00	0,000000		80	2	total

Figura 6: Resumen de las llamadas a sistema ejecutadas al ensamblar un binario mediante *NASM*.

Lo primero que se nota es que el número de llamadas a sistema asciende de 30 a 80. Igualmente es un hallazgo interesante el saber que casi la mitad de syscalls llamadas al ensamblar un binario sean

únicamente para iniciar la *libc*.

Teniendo a mano estos 2 resultados, se pueden identificar como innecesarias para la ejecución de *NASM* luego de la inicialización de la *libc* a las siguientes syscalls:

- *mprotect*: Modifica la protección de acceso de la memoria para el proceso actual.
- *pread64*: Lee y escribe archivos del sistema.
- *access*: Verifica los permisos que posee un usuario sobre un archivo pasado como parámetro.
- *execve*: Ejecuta un binario dado su path.
- *arch_prctl*: Setea el estado de un proceso o hilo.

Dejando esas llamadas a sistema fuera, la lista final de syscalls permitidas durante la ejecución del ensamblador es la siguiente:

- *read*: Se utiliza para leer archivos.
- *write*: Se utiliza para escribir archivos.
- *close*: Cierra descriptores de archivo.
- *fstat*: Obtiene los metadatos de un archivo.
- *lseek*: Re posiciona el «cursor» sobre un archivo abierto.
- *mmap*: Crea un nuevo mapeo en el espacio de memoria virtual del proceso.
- *munmap*: Elimina el mapeo creado por la llamada «*mmap*».
- *brk*: Modifica el tamaño del segmento de datos del proceso.
- *openat*: Abre o crea un archivo.
- *exit_group*: Finaliza la ejecución del proceso.

Lo primero a notar es que de esta forma tan simple se ha pasado de permitir más de 300 syscalls a permitir solo las 10 que el proceso necesita. En el caso en que el proceso quisiera usar alguna de las syscalls no permitidas, el mismo sería terminado en el acto antes de poder hacer ningún tipo de daño. El estudiar todas las syscalls disponibles y sus maneras de explotar nuestro sistema sería un trabajo muy laborioso, es por esto que el poder reducir esta lista a 10 simples llamadas a priori es un paso enorme hacia un servidor más seguro.

La mayoría de las syscalls permitidas son en un principio inofensivas. Sin embargo, es necesario tener precauciones con «*read*», «*write*», «*openat*» y «*close*». Estas 4 llamadas permiten al usuario acceder al sistema de archivos, por lo que serían las responsables de todas las catástrofes nombradas anteriormente.

Veamos igualmente que estas problemáticas fueron atrapadas en el llamado a *minijail*, ya que en el mismo se limitó el acceso de lectura de *NASM* únicamente a las carpetas que necesita:

- *user*: Carpeta donde se encuentra únicamente el archivo de entrada del usuario actual.
- */usr/bin*: Carpeta donde se encuentra el binario de *NASM*.
- */proc*: Carpeta con información de los demás procesos, sin información sensible debido al PID namespace.

De esta forma ningún archivo sensible queda a la vista del ensamblador, ni el binario del servidor, ni las contraseñas, ni nada que sea necesario mantener escondido.

Por otra parte, *NASM* solo tiene permiso de escritura en la carpeta generada para el usuario, por lo que no debería ser capaz de modificar ejecutables o de crear sus propios archivos en direcciones no deseadas.

Se muestra ahora el poder de *minijail* comparando los recursos visibles para *NASM* antes y después de utilizarlo. Primero se mostrarán los procesos visibles hacia el programa:

```

backend_1 | =====
backend_1 | =Procesos visibles para NASM/LD sin aplicar ningun namespace=
backend_1 | =====
backend_1 | $ ps -aux
backend_1 | USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
backend_1 | root         1   0.0  0.0 1078076 7576 ?        Ssl   03:16   0:00 ./http
backend_1 | root        42   0.0  0.0   7636  2676 ?        R    03:17   0:00 /bin/ps -aux
backend_1 |

```

Figura 7: Recursos visibles para los procesos *NASM* y *LD* sin utilizar *minijail*.

En la figura 7 se puede ver el resultado de ejecutar *ps -aux*. Se encuentra como PID 1 el proceso que corre el servidor en Go, luego es posible notar que el PID 42 corresponde al proceso de *ps*. Esto indica que en el mismo contexto hay muchos más procesos que ocupan los números menores, entre ellos se podrían encontrar los ejecutables enviados por otros usuarios, o incluso podrían haber procesos propios del servidor.

En cambio, mirando el resultado de la ejecución del mismo comando en la figura 8, es posible notar que el proceso de PID 1 ya no es más el servidor, si no que es *minijail*. Esto da una idea de que *ps* se encuentra en un PID namespace. No solo eso, si no que además el PID del proceso *ps* es el 2 mostrando así que ningún otro proceso es accesible desde el proceso ejecutado con *minijail*. Estos resultados son análogos a los que se obtendrían si reemplazáramos el comando *ps* con *NASM* o con *LD*.

```

backend_1 | =====
backend_1 | =Procesos visibles para NASM/LD luego de aplicar minijail con PID y VFS namespaces=
backend_1 | =====
backend_1 | $ ps -aux
backend_1 | USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
backend_1 | 0           1   0.0  0.0   2720   180 ?        S    03:19   0:00 /usr/bin/minijail0 -p -v -P /var/empty -b /clients/S8349dGHC3PH0w== 1 -b /bin/ps -b /proc -r /bin/ps -aux
backend_1 | 0           2   0.0  0.0   7780   1300 ?        R    03:19   0:00 /bin/ps -aux
backend_1 |

```

Figura 8: Recursos visibles para los procesos *NASM* y *LD* utilizando *minijail*.

Este mismo análisis se puede hacer viendo esta vez los puntos de montaje disponibles al momento de ejecutar el ensamblador o el linker. Esto se puede lograr mediante el comando *findmnt -output TARGET,SOURCE,VFS-OPTIONS*.

```

backend_1 | =====
backend_1 | =Filesystems visibles para NASM/LD sin aplicar ningun namespace=
backend_1 | =====
backend_1 | $ findmnt --output TARGET,SOURCE,VFS-OPTIONS
backend_1 | TARGET      SOURCE      VFS-OPTIONS
backend_1 | /            overlay     rw,relatime
backend_1 | /proc        proc        rw,nosuid,nodev,noexec,relatime
backend_1 | /dev         tmpfs       rw,nosuid
backend_1 | /dev/pts     devpts      rw,nosuid,nodev,noexec,relatime
backend_1 | /dev/mqueue  mqueue      rw,nosuid,nodev,noexec,relatime
backend_1 | /dev/shm     shm         rw,nosuid,nodev,noexec,relatime
backend_1 | /sys         sysfs       rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup  tmpfs       rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/systemd  cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/freezer  cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/devices  cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/pids      cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/cpuset    cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/net_cls,net_prio  cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/cpu,cpuacct  cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/perf_event  cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/memory    cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/blktio    cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/rdma      cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /sys/fs/cgroup/hugetlb   cgroup[ /docker/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125] rw,nosuid,nodev,noexec,relatime
backend_1 | /etc/resolv.conf         /dev/nvme0n1p6[ /var/lib/docker/containers/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125/resolv.conf] rw,relatime
backend_1 | /etc/hostname            /dev/nvme0n1p6[ /var/lib/docker/containers/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125/hostname] rw,relatime
backend_1 | /etc/hosts               /dev/nvme0n1p6[ /var/lib/docker/containers/152d56a0bc8bb0e6357f8e4086f7474d36afc47986b78228152b18a24ef5a125/hosts] rw,relatime

```

Figura 9: Recursos visibles para los procesos *NASM* y *LD* sin utilizar *minijail*.

Analizando la figura 9 se puede ver que son varios los puntos de montaje visibles. Para empezar, se puede notar que la raíz se encuentra montada. Al no usar *minijail*, esta es la carpeta raíz del servidor, es decir que el proceso tiene acceso a todos los archivos del sistema, entre ellos el binario del servidor mismo y los archivos generados por otros usuarios. Es posible notar además, que todas las carpetas tienen permisos de escritura, lo cual ya se anticipó que es muy peligroso.

Comparemos esta situación con la que se muestra en la figura 10 donde se utiliza *minijail* con los flags detallados anteriormente. Se puede ver que aquí también se tiene montada la raíz, pero en este caso existe una gran diferencia, aquí la raíz se encuentra en */var/empty*. Es decir, de deternos aquí el proceso no tendría acceso a absolutamente ningún archivo del sistema, ya que este directorio comienza

vacío. Nótese también, que está montada la carpeta creada específicamente para el usuario, que como se anticipó, cuenta con permisos de escritura, ya que es necesario que *NASM* y *LD* sean capaces de escribir sus archivos de salida allí. Por último, se monta la carpeta con los binarios necesarios (en este caso el binario de *findmnt*) y la carpeta */proc*, ambas con acceso de solo lectura por lo que no se corre riesgo de pisar ninguno de los archivos dentro de las mismas. Entonces, se puede ver claramente que los archivos accesibles gracias a *minijail* son mínimos en comparación al caso de uso sin el mismo.

Antes de terminar, se debe notar que la problemática de levantar archivos mediante directivas de *NASM* está más que solucionada por lo desarrollado en el párrafo anterior. Por más directivas para husmear en el sistema de archivos con las que se cuenten, si en este sistema no hay archivos útiles que buscar, las mismas pierden toda su utilidad, y desde el punto de vista del desarrollador, su peligro.

```
backend_1 | =====
backend_1 | =Filesystems visibles para NASM/LD luego de aplicar minijail con PID y VFS namespaces=
backend_1 | =====
backend_1 | $ findmnt --output TARGET,SOURCE,VFS-OPTIONS
backend_1 | TARGET          SOURCE          VFS-OPTIONS
backend_1 | /               overlay[/var/empty]    rw,relatime
backend_1 | |-/clients/S8349dGHc3PH0w== overlay[/clients/S8349dGHc3PH0w==] rw,relatime
backend_1 | |-/bin/findmnt   overlay[/bin/findmnt]  ro,relatime
backend_1 | `-/proc         proc              ro,relatime
```

Figura 10: Recursos visibles para los procesos *NASM* y *LD* utilizando *minijail*.

Por último, veamos que en ningún momento se está asegurando que la ejecución de *NASM* efectivamente termine. La ejecución del request depende fuertemente de la terminación del ensamblador, ya que la misma se bloquea en el momento de llamar a *NASM*. Es decir, si el atacante lograra de alguna manera asegurar la no terminación de *NASM*, el hilo que ejecuta la petición del usuario quedaría esperando ese resultado de manera infinita, consumiendo así memoria y uso de CPU. Varios de estos requests bloqueantes podrían dar una latencia elevada en los tiempos de respuesta obtenidos para los requests inofensivos, ofreciendo así una mala experiencia al usuario, o en el peor de los casos forzaría a reiniciar el servidor, impidiendo el uso del servicio hasta que el mismo esté operativo nuevamente.

Para evitar este problema será utilizada la syscall «*prlimit64*» la cual permite setear límites en los recursos de un proceso. En este caso, el interés de esta syscall es el de limitar el tiempo de uso de la CPU, esto se hace mediante el flag «*RLIMIT_CPU*». El mismo es un límite en segundos que indica el tiempo máximo que un proceso puede usar la CPU. Pasado este tiempo se le envía al proceso un *SIGKILL* y el mismo es terminado.

Este límite será usado en todos los procesos ejecutados, los cuales són: el ensamblador, el linker y el ejecutable. Es por este motivo que desde aquí en más este problema será considerado como resuelto. El tiempo máximo elegido es de 2 segundos.

2.3.10. Etapa de Linkeo

Esta etapa debería ser levemente más segura que la anterior, ya que para empezar *LD* es un programa que cuenta con mucha más jerarquía que *NASM*, siendo el primero usado por prácticamente todos los sistemas operativos desde hace muchos años, por lo que es de esperar que el mismo haya sido mucho más depurado que *NASM* ante posibles ataques. Además, el archivo que le llega a *LD* ha pasado una primera capa de filtrado que fue la etapa de ensamblado, donde con suerte se han dejado atrás muchos ataques posibles.

Igualmente, en esta etapa los problemas a enfrentar son exactamente los mismos: el acceso indeseado al filesystem o el abuso de las syscalls. Es por esto que aquí también será usado *minijail* y se mantendrán los flags usados con *NASM*. De esta forma también se contará con el PID namespace lo cual permitirá aislar a los demás procesos de el linker. Asimismo, los puntos de montaje serán los mismos ya que el binario de *NASM* y el de *LD* se encuentran en la misma carpeta y la salida del archivo ejecutable también se realizará en la carpeta del cliente, por lo que el linker no necesita permiso de escritura en ningún otro directorio.

Por último, es momento de ver las llamadas a sistema que necesita el linker. Para eso se realizará el mismo ejercicio que en la sección anterior, utilizando *strace* para ver cuales de las syscalls son realmente necesarias. El resultado de dicha ejecución se puede ver en la figura 11.

% time	seconds	usecs/call	calls	errors	syscall
17,42	0,000344	14	23	12	openat
15,70	0,000310	11	27		mmap
14,58	0,000288	8	35		lseek
11,34	0,000224	10	21		fstat
10,18	0,000201	20	10		mprotect
6,89	0,000136	11	12		read
4,96	0,000098	8	11		close
3,80	0,000075	12	6		pread64
3,70	0,000073	18	4		fcntl
2,84	0,000056	56	1		munmap
2,68	0,000053	7	7		brk
1,92	0,000038	12	3	1	stat
0,96	0,000019	19	1		chmod
0,76	0,000015	7	2		umask
0,76	0,000015	15	1		getrusage
0,61	0,000012	6	2	1	arch_prctl
0,56	0,000011	1	6		write
0,35	0,000007	7	1		prlimit64
0,00	0,000000	0	1	1	access
0,00	0,000000	0	1		execve
100.00	0,001975		175	15	total

Figura 11: Resumen de las llamadas a sistema ejecutadas al linkear un archivo objeto mediante *LD*.

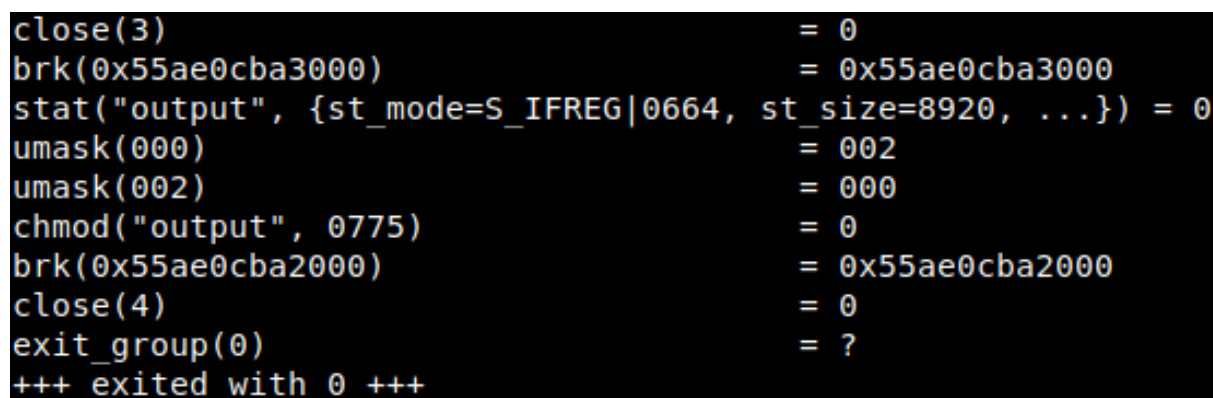
En esta ocasión la cantidad de syscalls asciende hasta casi 180. Por lo que se puede notar que este programa es mucho más dependiente de las llamadas a sistema que *NASM*. Por ese motivo es que se sumarán muchas syscalls nuevas a las listadas en la sección anterior. Igualmente, la lista de llamadas a sistema denegadas no se modifica en gran medida, solo obtiene una adhesión de *munmap*, syscall utilizada para eliminar un mapeo de memoria creado por *mmap*. Veamos entonces la lista de llamadas que serán permitidas durante la ejecución del linker.

- read: Se utiliza para leer archivos.
- write: Se utiliza para escribir archivos.
- close: Cierra descriptores de archivo.
- stat: Obtiene los metadatos de un archivo.
- fstat: Mismo comportamiento que «stat».
- lseek: Re posiciona el «cursor» sobre un archivo abierto.
- mmap: Crea un nuevo mapeo en el espacio de memoria virtual del proceso.
- brk: Modifica el tamaño del segmento de datos del proceso.
- fcntl: Manipula descriptores de archivos.
- unlink: Borra un link a un archivo. Si este link era el último, entonces elimina el archivo.

- `chmod`: Modifica los permisos de un archivo.
- `umask`: Setea la mascara del modo de archivo al momento de la creación para un proceso.
- `getrusage`: Obtiene el uso de recursos de un proceso o hilo.
- `prlimit64`: Obtiene/setea los recursos de un proceso.
- `openat`: Abre o crea un archivo.
- `exit_group`: Finaliza la ejecución del proceso.

Nuevamente son usadas llamadas a sistemas peligrosas. No se volverá a hablar de aquellas que fueron nombradas en la sección de seguridad en el ensamblado (2.3.9), ya que se vió que al tratarse de la manipulación de archivos, los mismos no deberían generar problemas por el enorme control que se ejerce sobre el poder de acceso y sobre los permisos que tiene el proceso con respecto al filesystem. Las syscalls que serán evaluadas en este caso serán las que tratan permisos o recursos de un proceso.

Lo primero a entender es que se está usando el linker para crear un archivo ejecutable, de ahí la necesidad de usar «`chmod`». Esta situación no es para nada ideal, ya que no es deseado que un usuario pueda cambiar el modo de un archivo a su antojo. Sin embargo, realizando un seguimiento ordenado de las syscalls realizadas por el linker se pudo ver que *LD* primero crea, escribe y cierra el archivo y justo antes de terminar su ejecución cambia los permisos del archivo de salida para permitir la ejecución del mismo. Este análisis se puede ver en la figura 12.



```

close(3)                = 0
brk(0x55ae0cba3000)      = 0x55ae0cba3000
stat("output", {st_mode=S_IFREG|0664, st_size=8920, ...}) = 0
umask(000)              = 002
umask(002)              = 000
chmod("output", 0775)   = 0
brk(0x55ae0cba2000)      = 0x55ae0cba2000
close(4)                = 0
exit_group(0)           = ?
+++ exited with 0 +++

```

Figura 12: Llamadas a sistema finales realizadas por *LD* a la hora de linkear un archivo objeto.

Como muestra la figura, luego de cerrar el descriptor de archivo 3, asociado al archivo binario «output», se realizan una serie de syscalls que no aportan a esta explicación y luego se llama a *chmod* para convertir «output» en un archivo ejecutable, justo antes de cerrar el descriptor de archivo 4, asociado a «output.o» y de finalizar el proceso.

Es decir que si hubiera una forma de que el linker no haga ese último paso de convertir el archivo en ejecutable, y el mismo fuera realizado desde el webserver de Go nuestro problema estaría resuelto ya que el linker no necesitaría el permiso a «`chmod`».

Como vimos antes, es posible «enmascarar» llamadas al sistema mediante *seccomp-bpf*, es decir, simular su ejecución cuando en realidad no se está tomando ninguna acción peligrosa. Esta funcionalidad es también provista de la mano de *minijail* por lo cual su implementación es muy simple. Enmascarando la syscall que trae problemas, se puede hacer creer al linker que los permisos del archivo han sido cambiados exitosamente y por consecuente terminará exitosamente con el archivo creado. Será el webserver Go el encargado de cambiar este modo en el archivo, lo cual no representa ningún problema de seguridad. De esta forma, aunque un atacante pueda ejecutar «`chmod`» desde el linker, esta syscall será solo una función «dummy» cuya única funcionalidad es la de retornar un estado de éxito.

Siguiendo el análisis con *strace*, también se pudo notar viendo la figura 13 que «`unlink`» se usa únicamente para eliminar al archivo con el mismo nombre que el de salida en el caso en que este existiera, es decir, para poder crear un archivo limpio.


```
unlink("output") = 0
openat(AT_FDCWD, "output", O_RDWR|O_CREAT|O_TRUNC, 0666) = 3
```

Figura 13: Llamadas a sistema donde *LD* elimina el archivo «output» para poder crearlo en un FS limpio.

Como es posible saber con seguridad que el archivo repetido no existirá, se tiene la libertad de no ejecutar este paso. Gracias a esto, se puede enmascarar también esta syscall quitándole así el poder de eliminar archivos al usuario. Pese a que esto a priori no es peligroso, debido a todo el control hecho sobre el filesystem, se considera que es una barrera de protección que no tiene ningún costo y que aporta otra capa de seguridad en caso de que la capa anterior sea comprometida. La figura 13 nos sirve además para verificar que al momento de la creación del binario la respuesta obtenida es un 3, haciendo referencia al descriptor de segmento nombrado en los párrafos anteriores.

Por último, otras 2 llamadas a sistema que pueden ser enmascaradas son «prlimit64» y «umask». El poder enmascarar la primera asegura que el usuario no podrá modificar los permisos del proceso que corre el linker. Si pudiera hacer esto el mismo podría quitar la restricción de tiempo puesta desde el servidor, permitiendo que el linker trabaje el request del usuario. Esto desperdiciaría poder de cómputo y memoria, por lo que con varios de estos requests el servidor se vería forzado a detenerse, perdiendo la capacidad de brindar servicio. El enmascarar «umask» prevendrá que el usuario pueda darle más permisos a archivos que no deberían tenerlos.

2.3.11. Etapa de Ejecución

Queda entonces la última etapa en el request del usuario, en la cual su código es al fin ejecutado y es posible recolectar la información pedida por el usuario. Esta ejecución es muy peligrosa porque aquí el usuario ni siquiera tiene que encontrar una vulnerabilidad para poder ejecutar su código, como si lo tuvo que hacer con el ensamblador y con el linker. En este caso basta con escribir la syscall que quiere ejecutar para poder hacer uso de la misma.

Pese a esta naturaleza tan peligrosa, la solución a este problema no es tan complicado como se pensaría. Dado que el SIMD Explorer tiene el propósito de ayudar en el desarrollo de algoritmos SIMD, se cree que tiene sentido bloquear absolutamente todas las syscalls en pos de la seguridad del servidor, ya que en esta programación las mismas no juegan un rol importante. Se aceptará una única syscall con el objetivo de que el programa termine de manera exitosa, la llamada para salir del proceso *exit*, la misma se usa en todas las ejecuciones y no representa ningún tipo de peligro.

Un programa que no tiene acceso a syscalls no representa absolutamente ningún tipo de peligro, ya que el mismo estará limitado a acceder a su memoria y a hacer cuentas. Lo cual es exactamente lo que se necesita para este momento.

Como en esta ocasión lo único que se hará será denegar el uso syscalls y no es necesaria ninguna otra funcionalidad provista por *minijail*, es posible dejar este programa de lado y realizar esta configuración a mano. Esto permitirá entender más a fondo una de las herramientas usadas por *minijail*: *seccomp*.

Para esto fue desarrollado *microjail*, un programa más modesto que *minijail* pero que permitirá cumplir con los requisitos de seguridad necesarios. El código fuente de este programa se puede encontrar en el link presente en el pie de esta página¹⁷.

El único objetivo a cumplir es el de asegurarse que el ejecutable creado para el usuario no pueda acceder a ninguna syscall a excepción de *exit*. Para lograr esto se hará uso de los filtros que provee *seccomp-bpf*¹⁸, estos son los mismos que utiliza *minijail* por detrás al leer los archivos *.policy*.

Este filtro se debe configurar y activar dentro del proceso que se quiera limitar, en este caso, el proceso que correrá el código del usuario. Cuando se ejecuta un programa externo desde Go, el mismo se ejecuta en un proceso hijo al webserver, es por eso que este filtro no se puede ejecutar desde aquí. Es necesario un wrapper en C que permita hacer un *exec* de otro programa sin crear otro proceso mediante un *fork*. Este wrapper será el encargado de crear y activar el filtro para luego darle el control al código del usuario, todo esto sin cambiar de proceso.

¹⁷https://gitlab.com/juampi_miceli/visual-simd-debugger/-/tree/master/backend/microjail

¹⁸<https://en.wikipedia.org/wiki/Seccomp>

Para lograr esto, se desarrollará a continuación la lógica para ejecutar el programa del usuario.

Lo primero que se debe hacer es llamar a *microjail* desde Go, pasándole el nombre del archivo que se quiere ejecutar. En este momento Go creará y ejecutará un nuevo proceso el cual será rastreado mediante *ptrace*.

Comenzada la ejecución de *microjail*, se tiene permiso de ejecutar cualquier syscall deseada, eso es debido a que aún no se ha activado el filtro. Se empieza entonces con la configuración y activación del mismo, una vez que este filtro es activado el proceso perderá todos los privilegios con los que contaba.

Lo último que nos queda por hacer es ejecutar el archivo pasado por parámetro mediante una llamada a *exec* cediendo todo el control al usuario.

Se puede notar aquí un sutil pero importante problema. Luego de deshabilitar todas las syscalls es necesario ejecutar el nuevo archivo, pero al momento de hacer esto no tenemos permiso para llamar a *exec* ya que lo bloqueamos al activar el filtro *seccomp-bpf*.

Como el filtro tiene que ser activado necesariamente antes de ejecutar el código del usuario no es posible rodear este problema de ninguna manera, no existen formas de, mediante *seccomp-bpf*, permitir que la syscall se ejecute una sola vez, ni que su permiso sea revocado al entrar a otro programa, ya que el proceso sigue siendo el mismo. Es necesario entonces permitir al usuario la llamada al sistema *exec*. Esto en principio es un gran problema ya que permitiría al atacante ejecutar cualquier archivo en el sistema, recordemos que en este caso no se cuenta con ningún namespace del sistema de archivos.

A priori, se puede pensar que mediante *seccomp-bpf* se puede requerir que *exec* solo tenga permitido ejecutar si lo que va a ejecutar es efectivamente el archivo deseado. De este modo, desde ensamblar el usuario solo podría ejecutarse a si mismo lo cual, no es ideal, pero tampoco es un ataque muy peligroso.

Igualmente, esta solución no es válida ya que el chequeo de seguridad no se realiza utilizando el contenido del string que contiene el archivo a ejecutar. Si no que para la comparación se utiliza la posición de memoria donde vive ese archivo.

Se puede entonces fijar la dirección de memoria donde vive ese string, esto dificultará al atacante el uso de *exec*, ya que debe ser capaz de conocer y modificar la posición de memoria que le fue asignada al string para pasarla a la syscall.

Podemos igualmente aumentar la dificultad que tendrá el atacante para poder ejecutar archivos arbitrarios, en los siguientes párrafos se desarrollarán las soluciones implementadas.

Lo primero a notar es que *exec* toma 3 parámetros. Estos son:

1. *file*: Nombre del archivo a ejecutar junto con su path.
2. *argv*: Array de argumentos (strings) pasados al programa.
3. *envp*: Variable de entorno que puede contener paths donde buscar el archivo ejecutable.

Estos 3 parámetros se pasan por referencia, es decir que reciben una dirección de memoria. Como para el uso actual, no son necesarios *argv* ni *envp*, se requerirá mediante *seccomp-bpf* que estas 2 direcciones sean *null*. O sea, que no apunten a ningún string. Al hacer esto, el atacante es limitado a ejecutar programas sin argumentos y que no requieran *envp*.

Queda ahora proteger la dirección de memoria donde se encuentra el string contenedor del ejecutable que será pasado como parámetro a *exec*. Veamos primero una característica de los sistemas operativos puede resultar de utilidad.

En las distribuciones modernas se implementa una técnica de seguridad que consiste en la disposición aleatoria del espacio de direcciones (ASLR por sus siglas en inglés), la misma fue desarrollada con el objetivo de combatir la explotación de vulnerabilidades basadas en la corrupción de memoria. ASLR dispone de forma aleatoria las posiciones del espacio de direcciones de las áreas de datos clave de un proceso, incluyendo la base del ejecutable y las posiciones de la pila, el heap y las bibliotecas.

Visto esto, se puede notar que la mitad del problema ya ha sido resuelta, y es que la posición del string en la memoria tiene una posición que no resulta simple de predecir, se puede igualmente aportar incluso más seguridad.

Para empezar, actualmente se está usando en *microjail* como dirección de memoria fijada, la posición donde vive el string recibido como parámetro desde Go. Por defecto, estos datos son almacenados en la pila, y por ende, la posición de memoria que pedirá *seccomp-bpf* estará dentro del stack en el contexto

de ejecución de *microjail*. Pese a que esta posición es aleatoria por lo expuesto anteriormente sobre ASLR y que los mapeos generados para *microjail* serán destruidos y unos nuevos serán creados para el binario del usuario, el inicio del stack suele estar en el mismo rango en todos los procesos, ya que la pila históricamente siempre estuvo en la última sección de la memoria.

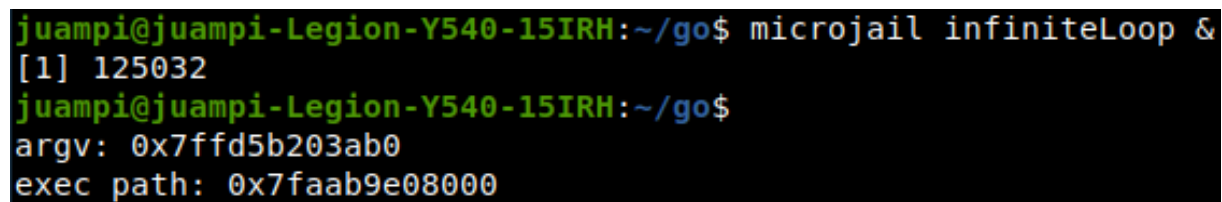
Esto podría causar que en el nuevo mapeo generado para el usuario, específicamente en el stack, se encuentre disponible la posición de memoria necesaria para ejecutar *exec*. Esto no quita que el atacante deba ser capaz de localizarla y modificarla, cosa para nada trivial, pero sabemos que si de alguna manera fuera capaz de predecir los mapeos aleatorios de ASLR, sería capaz de ejecutar cualquier programa dentro del sistema.

Para reducir las probabilidades de que esto ocurra, se dejará de usar la dirección de memoria perteneciente al stack que se estaba usando anteriormente. En su lugar, se pedirá al kernel que, mediante *mmap* mapee una página de memoria exclusivamente para contener a este string. Esta página también se encontrará en una posición aleatoria de la memoria, pero no necesariamente estará cerca del final de la misma, su posición podría ser cualquiera. De esta manera las chances de que esta misma memoria sea mapeada nuevamente a la hora de ejecutar el código del usuario son significativamente menores. Entonces, suponiendo nuevamente que el atacante es capaz de predecir exactamente todos los mapeos generados mediante ASLR, si este no tiene la página que necesita mapeada, de nada le servirá poder conocer la dirección del string, ya que recordemos que en el momento en que el usuario tenga el control, *mmap* entra en la lista de los llamados a sistema que se encuentran terminantemente prohibidos y entonces no podrá ser capaz de modificar la dirección de ninguna manera.

Con el fin de facilitar la comprensión de lo relatado en los últimos párrafos, se realizó una ejecución de *microjail* con un programa muy simple cuyo único propósito era ejecutar un loop infinito. El mismo no ejecuta ninguna syscall por lo cual no será expulsado en ningún momento.

Se podrá obtener de la ejecución: la posición de memoria tanto del string pasado por parámetro como la que se obtiene luego del *mmap*. Además, se mostrarán los mapeos de memoria de ambos ejecutables a fin de compararlos.

Primero, se mostrará el resultado de la ejecución de *microjail* en la figura 14.



```
juampi@juampi-Legion-Y540-15IRH:~/go$ microjail infiniteLoop &  
[1] 125032  
juampi@juampi-Legion-Y540-15IRH:~/go$  
argv: 0x7ffd5b203ab0  
exec path: 0x7faab9e08000
```

Figura 14: Resultado visto al ejecutar *microjail* en el background.

Lo primero que se puede notar, es que debido a que se está corriendo *microjail* en el background, se imprime un número que indica el PID del proceso que se encargará de ejecutar *microjail*. Además, también se imprimen las 2 posiciones de memoria pedidas. La primera es *argv*, esta es la posición donde se encuentra guardado el string pasado como parámetro, en cambio *exec path* es la dirección del string obtenida luego del *mmap*. Por si sola, esta imagen no aporta demasiada información, para tener más información acerca de estas 2 direcciones, se verán ahora en la figura 15 los mapeos de memoria asociados al proceso de *microjail* antes de hacer el *exec*. Esto se hará imprimiendo la información aportada por el archivo `/proc/<PID>/maps`, en nuestro caso el PID será el obtenido en la figura 14 (125032).

```

juampi@juampi-Legion-Y540-15IRH:~$ cat /proc/125032/maps
55ae28cd7000-55ae28cd8000 r--p 00000000 103:06 2753240 /usr/bin/microjail
55ae28cd8000-55ae28cd9000 r-xp 00001000 103:06 2753240 /usr/bin/microjail
55ae28cd9000-55ae28cda000 r--p 00002000 103:06 2753240 /usr/bin/microjail
55ae28cda000-55ae28cdb000 r--p 00002000 103:06 2753240 /usr/bin/microjail
55ae28cdb000-55ae28cdc000 rw-p 00003000 103:06 2753240 /usr/bin/microjail
55ae29a35000-55ae29a56000 rw-p 00000000 00:00 0 [heap]
7faab9bcb000-7faab9bf0000 r--p 00000000 103:06 3671706 /lib/x86_64-linux-gnu/libc-2.31.so
7faab9bf0000-7faab9d68000 r-xp 00025000 103:06 3671706 /lib/x86_64-linux-gnu/libc-2.31.so
7faab9d68000-7faab9db2000 r--p 0019d000 103:06 3671706 /lib/x86_64-linux-gnu/libc-2.31.so
7faab9db2000-7faab9db3000 ---p 001e7000 103:06 3671706 /lib/x86_64-linux-gnu/libc-2.31.so
7faab9db3000-7faab9db6000 r--p 001e7000 103:06 3671706 /lib/x86_64-linux-gnu/libc-2.31.so
7faab9db6000-7faab9db9000 rw-p 001ea000 103:06 3671706 /lib/x86_64-linux-gnu/libc-2.31.so
7faab9db9000-7faab9dbf000 rw-p 00000000 00:00 0
7faab9ddc000-7faab9ddd000 r--p 00000000 103:06 3671605 /lib/x86_64-linux-gnu/ld-2.31.so
7faab9ddd000-7faab9e00000 r-xp 00001000 103:06 3671605 /lib/x86_64-linux-gnu/ld-2.31.so
7faab9e00000-7faab9e08000 r--p 00024000 103:06 3671605 /lib/x86_64-linux-gnu/ld-2.31.so
7faab9e08000-7faab9e09000 rw-p 00000000 00:00 0 ← MMAP
7faab9e09000-7faab9e0a000 r--p 0002c000 103:06 3671605 /lib/x86_64-linux-gnu/ld-2.31.so
7faab9e0a000-7faab9e0b000 rw-p 0002d000 103:06 3671605 /lib/x86_64-linux-gnu/ld-2.31.so
7faab9e0b000-7faab9e0c000 rw-p 00000000 00:00 0
7ffd5b1e3000-7ffd5b205000 rw-p 00000000 00:00 0 ← STACK [stack]
7ffd5b22a000-7ffd5b22d000 r--p 00000000 00:00 0 [vvar]
7ffd5b22d000-7ffd5b22e000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]

```

Figura 15: Mapeos de memoria asociados al proceso de *microjail* antes de ejecutar el *exec*.

Si se hacen las comparaciones de memoria, se puede ver que como se había adelantado *argv* se encuentra dentro de la memoria asociada al stack, cerca de la parte más alta del mismo. Sin embargo, este nos es el caso de *exec path*, esta dirección se encuentra en una página de memoria distinta, la obtenida por *mmap* y ha sido marcada en la figura para facilitar el avistado de la misma. Queda ver ahora en la figura 16 el mapeo de memoria luego del *exec*, ya que se quiere confirmar que la dirección de *exec path* es efectivamente desmapeada.

```

juampi@juampi-Legion-Y540-15IRH:~$ cat /proc/125032/maps
00400000-00402000 r-xp 00000000 103:06 133898 /home/juampi/go/infiniteLoop
7ffd317000-7ffd3191000 rwxp 00000000 00:00 0 ← STACK [stack]
7ffd31d6000-7ffd31d9000 r--p 00000000 00:00 0 [vvar]
7ffd31d9000-7ffd31da000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]

```

Figura 16: Mapeos de memoria asociados al proceso de *microjail* luego de ejecutar el *exec*.

Es fácil notar que la cantidad de mapeos luego del *exec* es enormemente menor, en este caso toda la memoria asociada a biblioteca o pedida dentro de la ejecución ha sido desmapeada. Solo quedan 5 mapeos: el primero es donde se encuentran las instrucciones que el programa debe ejecutar, el segundo es efectivamente el stack, notemos que en esta ocasión este stack no se solapa con el obtenido antes del *exec*, pero esto no siempre será así. Los 3 siguientes mapeos se realizan en todos los procesos pero no entran en el alcance de este trabajo, ya que no aportan información útil para el mismo.

Queda entonces en evidencia lo difícil, si no imposible que será para el atacante el encontrar y modificar esta posición de memoria. Pero además es un buen momento para comprobar lo adelantado en la sección de ensamblado, linkeo e inicio del programa (2.2.3), en la misma al momento de linkear se uso el flag *nostdlib*, el trabajo del mismo se puede ver comparando las figuras 15 y 16 ya que en donde el flag fue usado no se encuentra ninguna biblioteca, mientras que donde ese flag no se usó se pueden ver más de 10 mapeos de memoria únicamente para las mismas. El no tener las bibliotecas mapeadas asegura que el usuario no podrá hacer uso de ninguna función proveniente de la *libc*.

Se mostró entonces que es prácticamente imposible el abuso de la syscall *exec* desde assembler. Igualmente, si el atacante pudiera de alguna manera sortear las defensas anteriores, nos queda una última defensa que dificultará en gran medida el ataque. Hasta ahora, se habló de los 3 parámetros que recibe *exec*, lo que no se dijo es que toda syscall toma 6 registros. Dependiendo de cuantos de estos necesite la misma realmente, quedarán más, o menos registros sin usar. Se puede sin embargo pedirle a *seccomp-bpf* que compare estos valores con algún valor esperado.

Para terminar de dificultar el uso de *exec*, obtendremos 3 números de 64 bits de manera aleatoria, esto es debido a que nos quedan 3 registros libres. Cada uno de estos números será pasado a uno de los registros que han quedado sin usar y será requisito por *seccomp-bpf* que estos valores sean exactamente los generados de forma aleatoria, ya que de no ser así el programa será terminado.

De esta manera, si el atacante quisiera usar *exec*, se debe sumar a todas las problemáticas anteriores el deber ser capaz de predecir el valor de 192 bits que tienen que ser pasados desde assembler.

Es importante recalcar, que en el remoto caso en el que un atacante fuera capaz de ejecutar código arbitrario, ya sea mediante *exec*, mediante el apuntado del contador de programa a alguna instrucción dentro de un binario, o utilizando cualquier otra técnica, las restricciones sobre las syscalls permanecerán tanto como lo haga el proceso, incluso restringiendo el uso a llamadas del sistema de los hijos que este tenga. Es decir que cualquier código que logre ejecutar será muy poco peligroso.

Se incluirá ahora una explicación del filtro *BPF* implementado en C con el objetivo de facilitar el entendimiento de lectura del mismo y que además, este ejemplo quede plasmado en conjunto con su documentación para que pueda ser tomado como puntapié inicial en la creación de filtros futuros.

Para empezar, el filtro fue construido siguiendo el diagrama de flujo presente en la figura 17, de esta manera se están realizando todos los chequeos hablados a lo largo de esta sección. En la figura es más simple entender la lógica de los filtros *seccomp-bpf* explicada en las secciones de *seccomp* (2.3.4) y *BPF* (2.3.5): se realizan una serie de verificaciones ordenadas, avanzando de estado estrictamente para adelante y llegando en algún momento a una instrucción final, la cual puede permitir la ejecución o terminar el proceso por completo.

Veamos ahora como se traduce este diagrama a código dentro de la máquina virtual de *BPF*:

Lo primero es traducir los códigos de operación necesarios de números a macros entendibles por un ser humano. Esto es provisto por la biblioteca *linux/filter.h* mencionada en la sección de *BPF* (2.3.5) y nos permite realizar sentencias de manera más simple y prolija.

La mayoría de los macros definidos son reemplazos del código de operación, pero también se cuenta con «*BPF_STMT*» y «*BPF_JUMP*». Estos son simplemente una facilidad que se da para definir los valores de cada instrucción dentro de la estructura «*sock_filter*». Como es de esperarse «*BPF_STMT*» se usa para sentencias de carga de datos y «*BPF_JUMP*» para instrucciones de salto. La definición de estos macros se muestra a continuación:

```
#define BPF_STMT(code, k) { (unsigned short)(code), 0, 0, k }

#define BPF_JUMP(code, k, jt, jf) { (unsigned short)(code), jt, jf, k }
```

Ya podemos empezar a analizar el código escrito para el trabajo. Empecemos con las sentencias necesarias para determinar si la syscall entrante es *exit*.

```
BPF_STMT(BPF_LD + BPF_W + BPF_ABS, syscall_nr), // Carga el número de syscall al acumulador
//BPF_LD: Operación de carga
//BPF_W: El tamaño del operando será de tipo word
//BPF_ABS: La carga del valor se realiza del área de datos
//syscall_nr: Macro que accede al número de syscall en el área de datos

BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_exit, 0, 1), // Si no es EXIT salta una instrucción
//BPF_JMP: Operación de salto
//BPF_JEQ: El salto se realiza en caso de igualdad.
//BPF_K: Se comparará el acumulador contra el parámetro recibido en el argumento k (__NR_exit)

BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW), // Retorna el código ALLOW

//Continúa el filtro verificando si el número de la syscall corresponde a EXEC
```

Como es de esperar, el chequeo para verificar si la syscall corresponde a *exec* es muy similar. Por lo tanto, asumamos que este chequeo ya se hizo y queda verificar el valor de los argumentos.

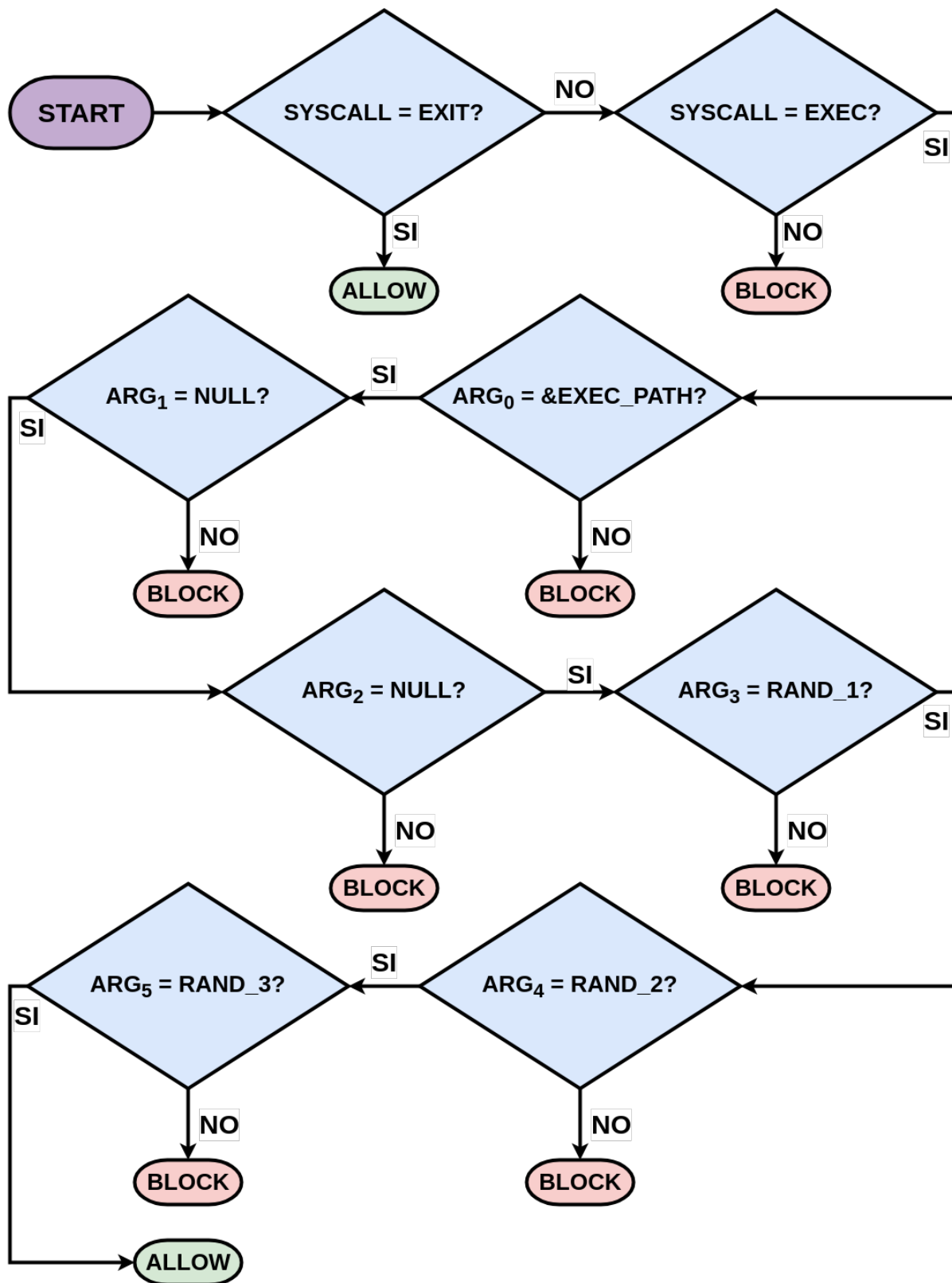


Figura 17: Diagrama de flujo de referencia para el filtro utilizado en microjail.

```

BPF_STMT(BPF_LD + BPF_W + BPF_ABS, syscall_arg(0)), // Carga el primer argumento al acumulador
//syscall_arg(i): Macro que accede al argumento i de la syscall recibida en el área de datos

BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, &execPath, 1, 0), // Si la dirección recibida en el
                                                         // argumento 0 es igual a la de execPath
                                                         // salta una instrucción

BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL), // Se termina el proceso debido al argumento inválido

//Continúa el filtro verificando que el valor de los demás argumentos sea válido

```

La lógica para verificar los demás argumentos nuevamente es similar a la ya desarrollada, por lo que se considera que no es necesaria su exposición en el informe.

Finalizada la implementación de *seccomp-bpf*, resta ahora resolver una última problemática, trivial en comparación con todo lo ya visto. Hasta este momento un atacante podría de alguna manera generar un programa que no termine, pero que periódicamente se detenga en un breakpoint. De esta forma, se esquivaría el control hecho por *prlimit64*, ya que el control del programa está siendo devuelto al webserver antes de ser expulsado. Esta sería una buena manera de dejar el programa corriendo de manera infinita para consumir recursos del servidor.

La solución a esto es muy simple, la guarda del loop dejará de verificar que el programa haya terminado y el mismo se ejecutará tantas veces como el número de celdas que se hayan recibido en el request. Luego de eso el programa será finalizado independientemente de si se llegó al final o no.

2.3.12. Denegación de servicio

Una problemática a tener en cuenta es que hasta este punto, el webserver podría ser víctima de un ataque de denegación de servicio. El mismo se genera enviando un número excesivo de requests en un tiempo muy acotado, provocando que el servidor no tenga la capacidad para atender todas estas llamadas. Pese a que este ataque no sea complicado de realizar, se considera que la protección contra el mismo se encuentra muy por fuera del alcance de este trabajo. Aún así, se cree que es importante conocer la existencia del mismo.

2.3.13. Docker

En el caso en que todo lo anterior fallara y los archivos fueran dañados, la recuperación del mismo requeriría un formateo total de la maquina física en la cual se está ejecutando el webserver. Esto sería tedioso y muy poco performante. El proceso de deploy y update de la aplicación, a priori, tampoco es un escenario tan sencillo y automático.

Es por eso que se eligió la herramienta *docker* para contener al webserver. Este programa utiliza contenedores para facilitar la creación, implementación y ejecución de aplicaciones ya que vincula la aplicación y sus dependencias dentro de un contenedor.

El mismo proporciona una manera estándar de ejecutar la aplicación sin que sea necesario tener en cuenta la arquitectura o el sistema operativo donde se corra. En primera instancia esto es una gran ventaja ya que facilita en gran medida tanto el proceso de deploy como el de update de una aplicación, ya que los mismos se reducen ahora a la ejecución de un par de comandos independientemente si se modifica la pc donde se corre, el sistema operativo, o la versión del mismo aplicativo.

Las aplicaciones que se encuentran dentro de *docker* suelen ser muy livianas, por lo que, en adición a lo hablado en el párrafo anterior se agrega que el arranque y reboot de las mismas es muy veloz, lo cual repercute en un tiempo de arranque muy corto.

Otra ventaja que trae el usar *docker*, es que debido a su naturaleza, la aplicación correrá dentro de un namespace que será exclusivamente para ella. Es decir que tendrá un filesystem propio que nada tiene que ver con el filesystem del servidor físico y lo mismo aplicará para los procesos.

Esto provee incluso otra capa de aislamiento para con el equipo físico que se suma a todas las anteriores ya relatadas. Es decir, supongamos que fallan todas las defensas previas y un atacante puede

corromper archivos, este problema se solucionará en menos de 10 segundos destruyendo el contenedor actual y creandolo nuevamente.

Esto se debe a que *docker* utiliza imágenes, las cuales son plantillas de quienes se crearan los contenedores con la aplicación en si. Tomando una analogía de la programación orientada a objetos, una «imagen» sería lo que se llama «clase» en programación, mientras que el «contenedor» se correspondería a una «instancia particular» de esa clase. Entonces reemplazar un contenedor dañado es tan simple como instanciar el mismo nuevamente usando la misma imagen que se usó originalmente. Estas se pueden guardar en repositorios o de forma local, ya que debido al aislamiento proporcionado por *docker* es menos peligroso que las mismas sean comprometidas.

En una aplicación que no utilice *docker*, para lograr esto se debería tener un backup de estos archivos en algún repositorio los cuales serían usados como archivos limpios y una vez descargados los mismos se debería instalar la aplicación en el servidor y ponerla a correr.

Obviamente, no se puede suponer que esta técnica es infalible, ya que, pese a que *docker* es un programa desarrollado por una gran compañía y es el sinónimo actual de sandboxing y containers, como mencionamos en la introducción, ningún programa queda exento de tener vulnerabilidades que puedan ser explotados.

Igualmente, el mismo es actualizado con frecuencia para emparchar estas vulnerabilidades y se considera que la dificultad de encontrar un exploit de *docker* es muy superior a la de encontrar uno para este trabajo.

Es por esto que el uso de *docker* para contener la aplicación incrementa de forma considerable la seguridad del webserver en general.

3. Conclusiones y trabajo futuro

A lo largo de este trabajo fueron usados conocimientos vistos en la materia para poder desarrollar el asistente visual.

Sin embargo, también se desarrollan muchos conceptos pertenecientes a los sistemas operativos que fueron muy útiles especialmente para la parte de seguridad. Gracias al desarrollo de la aplicación fue posible entender a fondo muchos de estos temas, como ser, namespaces, ptrace y procesos entre otros.

Al momento de entregar este informe, SIMD Explorer ha sido utilizado por alumnos de la materia y gran mayoría de los que lo han usado han afirmado que la herramienta les ha servido para entender cómo funcionaba alguna instrucción en particular y para realizar ejercicios tanto del parcial sobre SIMD como del trabajo práctico asociado a este tema.

Estos eran exactamente los objetivos deseados con esta herramienta, es por esto que se considera que la misma cumple su función de manera exitosa. Además se ha escuchado cuales son las funcionalidades deseadas por quien cursa la materia y está iniciándose a estos algoritmos.

La funcionalidad más deseada es la posibilidad de visualizar registros de propósito general ya que los mismos tienen algunos usos en algoritmos SIMD, por lo que se considera que el agregarlos podría hacer la herramienta incluso más útil para los usuarios. Las demás mejoras recomendadas fueron con respecto a la interfaz brindada al usuario, esto es esperable ya que se priorizó el poder brindar un producto funcional para que el mismo pudiera ser aprovechado durante la cursada, igualmente se tratará de mejorar esta experiencia a la brevedad.

Una mejora que se podría implementar con respecto a la seguridad sería la de tratar el problema de la denegación de servicio (sección 2.3.12), esto podría realizarse limitando la cantidad de requests por minuto que se pueden realizar desde la misma IP, el problema con esta aproximación es que si la herramienta está siendo utilizada por muchos estudiantes o desarrolladores desde la misma red es muy probable que su experiencia no sea óptima pese a que no se esté intentando realizar ningún ataque. Otra manera de abordar esta problemática sería añadir un sistema de autenticación para usuarios donde los estudiantes de las distintas facultades que utilicen SIMD Explorer tengan acceso ilimitado al webserver mientras que el rate-limiting se aplicaría a usuarios no registrados. Una ventaja de esto es que si algún alumno intentará denegar el servicio mediante el ataque desarrollado en la sección de denegación de servicio (2.3.12), el mismo podría ser identificado por las credenciales ingresadas y se lo podría agregar a una lista de usuarios bloqueados.