Abrimos el abo en el ida y vemos que recibe como parámetro 2 argumentos





**sscanf** lee de una cadena y no de la entrada estándar.

• int sscanf (char ‘cadena, char \*format, argv arg2, ...)

• Rastrea la cadena de acuerdo con el formato en **format**, y almacena el valor resultante a través de **argv**, **arg2**, etc. Estos argumentos deben ser punteros.

En el bloque principal aquí vemos como están representadas las variables en el stack.

Texto

Descripción generada automáticamente con confianza media

Okey, vemos que se le pasa dos argumentos.

Por ejemplos: “AAAA” y “BBBB”

Comenzamos por el primer bloque:

Aquí esta función **sscanf** lee datos de entrada desde un buffer en este caso de **argv1**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

vemos que se le indica que sea **%u** ósea **unsigned integer** , va a retornar un numero que lo guarda en la variable **offset.** Entonces ya sabemos que el primer argumento debe ser un numero que hará de offset.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Después se calcula el tamaño de la cadena que se paso por **argv2**



Luego aquí tendremos el **size** de **argv2** en **eax,** y hace las siguientes operaciones

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Como se ve el calculo seria el siguiente: **buffer + offset \* 4 + len(argv2)**

Entonces por lo que veo aquí el salto de esa comparación dependerá del **len(argv2).**

Y después hace un **strcpy**, aquí vemos que el **destino** seria la suma del **buffer + offset \* 4,** además el tamaño del buffer es de **256 bytes.**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Si vemos la distancia que hay desde el **buffer** al **retn,** son **268 bytes**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Ahora bien, según la formula que se aplico **Destino = buffer + offset \* 4**

Eso desplazaría el inicio del **Destino** en principio por **4 bytes** (suponemos que el offset valdrá 1 siempre).

Entonces a **268** le restamos **4** y nos da el tamaño que debemos poner para pisar el **retn.**

**264 bytes** para pisar el **retn**

Muy bien le acabo de mandar **264 bytes** al **argv2,** y me tiro el mensaje **“no no”, -.-**

Okey, ya se es cuando toma el **len del segundo argumento**, se desvía en la comparacion que vimos arriba.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Lo que veo aquí, es que compara si el resultado final supera al comienzo de la siguiente variable dispuesta en la pila que le sigue, en este caso es **un\_byte**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Como vemos ahora si juega un rol importante el **len(argv2)**

Como lo veo, el tamaño máximo de **argv2** no puede superar los **255 - 4 ,** que seria el tamaño del **buffer.**

Aplicando eso no podríamos llegar a pisar **el retn** debido a que nos faltarían bytes para eso, cosa que no podemos porque tenemos el limite de **251 bytes en argv2.** Como se ve en la imagen nos faltan 12 bytes para llegar al retn.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media

**¿Cómo puedo hacer entonces? Pensemos!!!**

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja

Recordemos que teníamos esta formula: **Destino = buffer + offset \* 4**

Visto esto, vemos que dependiendo del valor del **offset** que le pasemos, sabiendo que a este valor lo controlamos por **argv1,** podemos controlar la **dirección de destino** del **strcpy.**

offset = -**63** & len(argv2)= **255** ==> Destino = 19FD3C Crashea!!! Yahoo!!

Si le metemos un **-63** en **argv1** y un **255** en **argv2** se produce el siguiente **crash.** Vemos que podemos controlar el **EIP, ESI** y **EDI.** Aquí en **EAX** tenemos la dirección del buffer que se usa en e **strcpy.**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Si vamos al **dump** a lo que apunta el registro **eax** que seria el buffer del **strcpy**, notaremos que esta gran parte en **CERO**, salvo el final del buffer.

Calendario

Descripción generada automáticamente con confianza media

Afortunadamente en el **stack** estamos parados justo en la parte que no se borro.

Tabla

Descripción generada automáticamente con confianza media

Okey, con una **cadena random** de **255** de longitud podemos ubicar cuales son los 4 bytes de la **shellcode** a cambiar para que se produzca la ejecución del **eip.**

También notamos que en **ESP+4** se encuentra la dirección del **argv2**, por lo que podremos hacer que salte ahí para ejecutar código, y así contar con todo el **shellcode completo**.

Interfaz de usuario gráfica, Texto, Aplicación, Word

Descripción generada automáticamente

Para ello saltaremos a un **POP / RETN,** de esta forma saca el primer elemento de la pila y retorna en lo que apunta el segundo.



Ahora ya estaríamos ejecutando la **shellcode para ejecutar la calculadora.** El tema es que este **abo** tiene muchas **badchars -.-**

Así que tendremos que rebuscárnosla para no enivar estos bytes malos. La idea es cargar **kernell32** y **GetProcAddress,** luego podemos usar **WinExec** y ejecutar la **calculadora.**

Tenemos todos los datos necesarios en el ABO para poder hacer esto!!!

Aquí mi **shellcode sin badchars.**

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media