Administración de memoria: Memoria y seguridad

Gunnar Wolf

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¿Seguridad?

- En un sistema de cómputo, no podemos asumir *la mejor intención*
  - Un sistema puede tener distintos usuarios, el sistema protege a los datos de unos frente a otros
  - Un usuario puede intentar modificar cómo opera determinado programa *y generar salida que aparente ser legítima*
  - Privilegios parciales/temporales: Bit SUID (a cubrirlo posteriormente, en *sistemas de archivos*), *capacidades*
- La memoria es el principal *vector de ataque*
  - Todos los datos deben pasar por memoria
  - . . . Y el procesador no puede *entenderla*, sólo *obedecerla*

¿No son suficiente los permisos?

- La memoria segmentada y paginada ya incluyen los conceptos de permisos
- Desde hace muchos años, los segmentos *de texto* no admiten escritura
- ¿Qué más puede ocurrir?

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Lidiando con información externa

- Tal vez no tendríamos que preocuparnos de mucho... Pero tenemos que lidiar frecuentemente con *información externa*
  - Información que puede romper nuestras expectativas
  - Información que no siempre viene de fuentes confiables
- Puede ser maliciosa, o presentar errores (¡inocentes!) en el comportamiento
- Los mecanismos principales de manejo de estructuras de memoria son bastante ingenuos $\rightarrow$ inseguros
  - Cualquier error en el manejo de memoria es una vulnerabilidad en potencia

Referencias bibliográficas

Para comprender mejor este tema, y para profundizar en él, sugiero fuertemente:

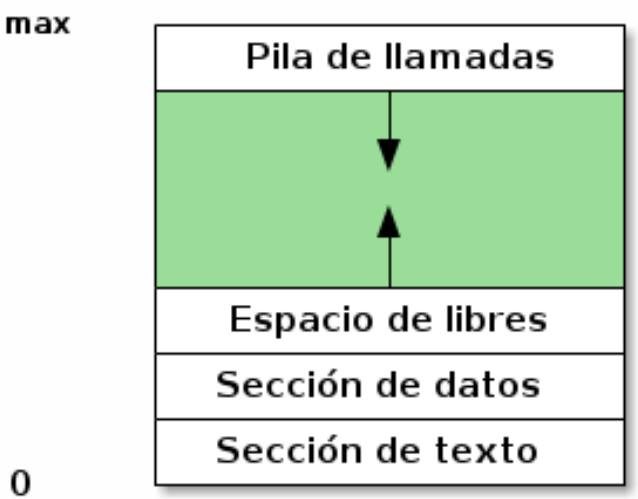
- Smashing The Stack For Fun And Profit (Aleph One, revista Phrack, 1996)
- The Tao of Buffer Overflows (Enrique Sánchez, inédito)

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

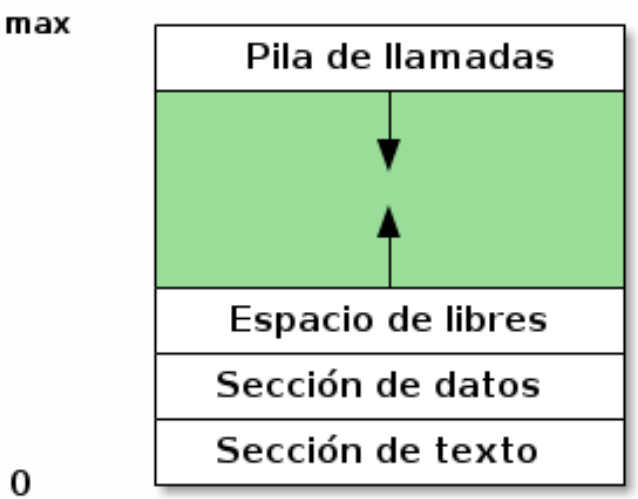
Vuelta a la organización de la memoria



- Pila de llamadas
- Espacio de *libres*
- Sección de datos
- Sección de texto

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Vuelta a la organización de la memoria



- Pila de llamadas
- Espacio de *libres*
- Sección de datos
- Sección de texto

Vamos a centrarnos en el *stack* (*pila de llamadas*):
¿Para qué sirve?
¿Cómo se estructura?

Enfocándonos a la *pila de llamadas*

¿Cómo se ve una *llamada a función* ya compilada?

- Empleamos dos valores principales, en registros:

  Stack Pointer (`SP`) Apunta al *final actual* (dirección inferior) de la pila. En arquitecturas x86, emplea el registro `ESP`.

  Frame Pointer (`FP`) Apunta al *inicio* (dirección superior) del *marco del stack*. Algunos textos le llaman *base local* (`LB`), y en arquitecturas x86, emplea el registro `EBP`.
- Guarda la *instrucción de retorno* (*instruction pointer*)
- Reserva espacio para los datos locales a la función

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Comparando llamada en C y ensamblador x86-32

```
void func(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  func(1,2,3);
}
```

```
; main
        pushl $3
        pushl $2
        pushl $1
        call func

func:
        pushl  %ebp
        movl  %esp, %ebp
        subl $20, %esp
```

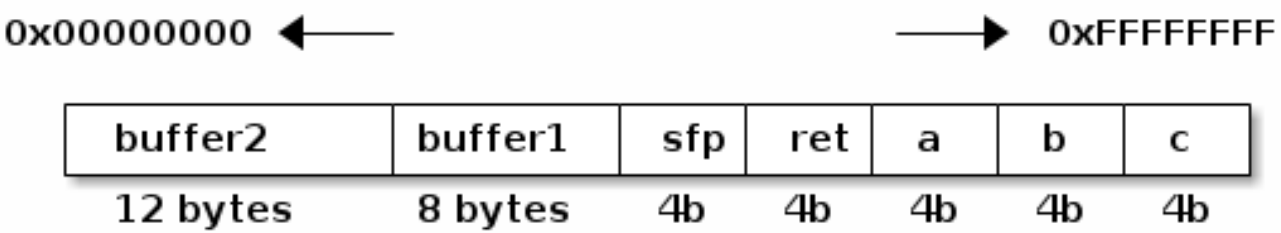


Figura: Marco del stack con llamada a `func(1,2,3)` en x86-32

¿Qué hizo el código compilado?

- Los tres primeros `pushl`: *Debajo* de la pila, agrega los tres argumentos de la función
  - Los valores literales 1, 2, 3
  - 4 bytes para cada uno (`sizeof(int())` en 32 bits)
- `call`: Agrega `ret` debajo de la pila, y brinca a la ubicación de la *etiqueta* `func`
  - Al compilar, se pierde la noción de *bloques* de función
  - Todo son llamadas por dirección que retienen la dirección de retorno
- Ya en `func`, grabamos `ebp` (el *frame pointer*), le llamamos `sfp` (*Saved Frame Pointer*)
- `movl` graba la dirección del apuntador actual al stack en el registro `esp`, y acto seguido, se le restan 20 bytes (por los 2 buffers, alineados a 32 bits)

El eslabón más débil: Funciones de manejo de cadenas

- El diseño del lenguaje C incluye soporte para cadenas a través de su biblioteca estándar
- Arreglos de caracteres (8bit), terminados por un `\0` (`NUL`)
- Presenta *familias* de funciones relacionadas para su manejo, principalmente `strcat`, `strcpy`, `printf`, `gets`
- Muchas de estas no toman en cuenta *verificación de límites*
- Otras, no *necesariamente* incluyen al `\0`

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Entran en juego los datos suministrados por el usuario

- ¿Qué pasa cuando un usuario puede proporcionar datos para que los maneje?
- Veamos qué pasa con el siguiente código *vulnerable*

```
#include <stdio.h>
int main(int argc, char **argv) {
        char buffer[256];
        if(argc > 1) strcpy(buffer, argv[1]);
        printf("Escribiste %s\n", buffer);
        return 0;
}
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Entran en juego los datos suministrados por el usuario

- ¿Qué pasa cuando un usuario puede proporcionar datos para que los maneje?
- Veamos qué pasa con el siguiente código *vulnerable*

```
#include <stdio.h>
int main(int argc, char **argv) {
        char buffer[256];
        if(argc > 1) strcpy(buffer, argv[1]);
        printf("Escribiste %s\n", buffer);
        return 0;
}
```

¿Quién puede encontrar el punto débil?

Copiado sin límites

Invocamos al programa vulnerable con un parámetro largo (pero válido):

```
$ ./ejemplo1 `perl -e 'print "A" x 120'`
Escribiste:
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$
```


Copiado sin límites

Invocamos al programa vulnerable con un parámetro largo (pero válido):

```
$ ./ejemplo1 `perl -e 'print "A" x 120'`
Escribiste:
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$
```

Invocamos al programa vulnerable con un parámetro demasiado largo para el arreglo:

```
$ ./ejemplo1 `perl -e 'print "A" x 500'`
Escribiste:
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
$
```

Parecería que el sistema *atrapó* al error exitosamente.

Copiado sin límites

Invocamos al programa vulnerable con un parámetro largo (pero válido):

```
$ ./ejemplo1 `perl -e 'print "A" x 120'`
Escribiste:
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$
```

Invocamos al programa vulnerable con un parámetro demasiado largo para el arreglo:

```
$ ./ejemplo1 `perl -e 'print "A" x 500'`
Escribiste:
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
$
```

Parecería que el sistema *atrapó* al error exitosamente. ¿O no?

¿Qué significa nuestro `Segmentation fault`?

- La ejecución del programa fue interrumpida tras procesar un argumento de 500 bytes para un arreglo medía sólo 256
  - . . . Pero esto *no significa* que el fallo haya sido atrapado
  - De hecho, *no lo fue*

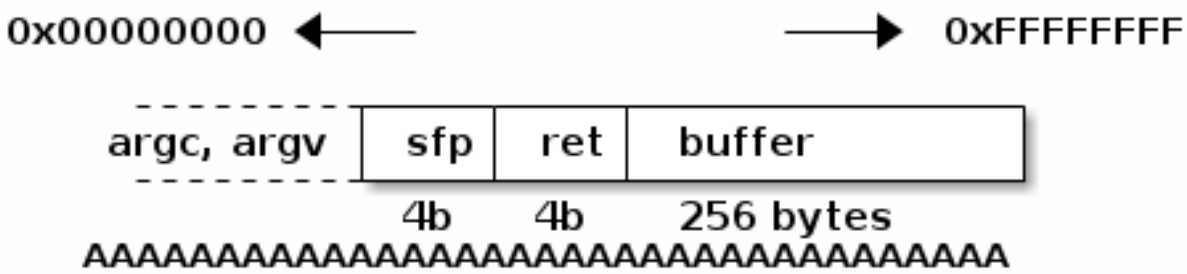


Figura: Estado de la memoria después del `strcpy()`

¿Qué significa nuestro `Segmentation fault`?

- El `Segmentation fault` se debe a que la *dirección de retorno* es la 0x41414141
  - No es una dirección válida y ejecutable
- El código está *demostrado vulnerable*
  - . . . Pero no lo hemos podido explotar aún

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Convirtiendo una falla de segmento en un ataque

- Conociendo el acomodo *exacto* de la memoria, podemos predecir cuánto mide el espacio de cada una de las variables
  - ...Y del SFP, y de la dirección de retorno
- Si no lo tenemos, con ayuda de un depurador (como `gdb`) podemos leer el estado de los registros o de ubicaciones específicas de memoria a lo largo de la vida del programa
- En este caso, la longitud *mágica* es de 264 bytes (256 del buffer + 4 de `RET` + 4 de `SFP`
- ¿Qué pasa si sobreescribimos `RET` con una dirección *válida*?

Brincando a otro punto del programa

- Es necesario comprender cómo el compilador *transforma* el aparente órden lógico de ciertas operaciones
- Si estamos en una rutina de verificación (digamos, de contraseña) y conocemos la dirección de quien invocó a esta verificación:

```
if (valid_user(usr, pass)) {
  /* (...) */
} else {
  printf("Error!\n");
  exit 1;
}
```

```
pushl [%rax]
pushl [%rbx]
call valid_user
bne $0, %ebx, user_fail
```

- La llamada a `valid_user()` regresa al `if` un valor verdadero (0) o falso (cualquier otra cosa)
- Si hacemos que el flujo *brinque* a inmediatamente después del `bne`, libramos la verificación

Brincando hacia dentro del stack

- Si en vez de 'AAAAAAAA...' le damos al sistema código ejecutable válido, más que brincarnos una validación, podemos hacer que el programa ejecute código arbitrario

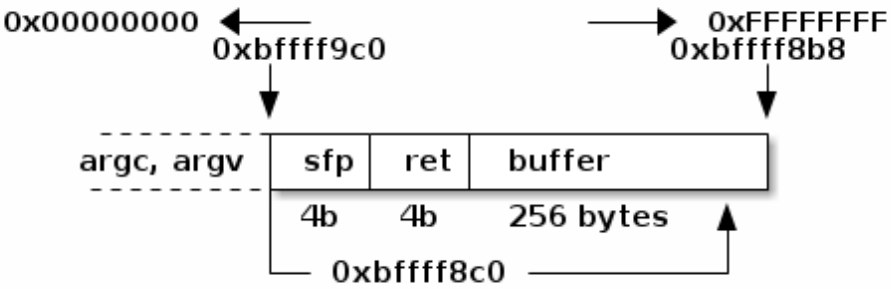


Figura: Ejecutando el código arbitrario inyectado al buffer

- El flujo de ejecución del proceso está comprometido
- Tenemos 256 bytes para inyectar el *shellcode* y ejecutar código arbitrario con los privilegios del proceso víctima

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Técnicas de mitigación

- Claro está, desde hace años se han desarrollado maneras de mitigar el daño causado
- En primer lugar, cultura de desarrollo seguro
  - No emplear funciones que no hagan *verificación de límites*
  - . . . Pero requieren conciencia por parte de todos los programadores $\rightarrow$ imposible. . .
  - Y son sensiblemente más lentas
- Segundo, dificultar al atacante inyectar datos arbitrarios
  - Aleatorización de direcciones
  - Canarios
  - Memoria no ejecutable

Aleatorización de direcciones

- Incluir espacio arbitrario entre SFP, RET y el buffer
- Cambiante a cada ejecución
- Dificulta al atacante insertar una cadena de las dimensiones requeridas
- Desventajas
  - Desperdicia espacio
  - No siempre puede ser aplicado

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Canario

- Insertar un valor aleatorio entre las variables locales y los registros del frame
- Verificarlo al volver de la función
  - Terminar el programa si no concuerda
- Desventajas
  - Debe verificarse a cada llamada a función
  - Rendimiento disminuído

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Memoria no ejecutable

- Bits de control en el MMU impidiendo que se ejecute la memoria en el stack
- Imposibilitan ejecutar código en el stack
  - Pero no regresar a otras direcciones