PREV NEXT

Fuzz Testing

Originally developed to find reliability bugs (Miller, Fredriksen, and So 1990; Miller et al. 1995; Miller 2005; Gallagher, Jeffries, and Landauer 2006), fuzz testing is an effective way to find certain classes of security bugs, too.

Fuzzing means creating malformed data and having the application under test consume the data to see how the application reacts. If the application fails unexpectedly, a bug has been found. The bug is a reliability bug and, possibly, also a security bug.

Note

At Microsoft, about 20 to 25 percent of security bugs are found through fuzzing a product before it is shipped. The vast majority of bugs are in old, pre-SDL code, however.

Fuzzing is aimed at exercising code that analyzes data structures, loosely referred to as parsers. There are three broad classes of parsers:

- **File format parsers** Examples include code that manipulates graphic images (JPEG, BMP, WMF, TIFF) or document and executable files (DOC, PDF, ELF, PE, SWF).
- **Network protocol parsers** Examples include SMB, TCP/IP, NFS, SSL/TLS, RPC, and AppleTalk. You can also fuzz the order of network operationsfor example, by performing a response before a request.
- **APIs and miscellaneous parsers** Examples include browser-pluggable protocol handlers (such as callto:).

Let's focus on each parser type in more detail.

Fuzzing File Formats

Fuzzing file formats means building malformed files to be consumed by your application. For example, if your application parses and displays TIFF files, you could build a malformed TIFF and have the application read the file. Of course, you don't create just one malformed file; SDL mandates that you create and test at least 100,000 malformed files for every file format and every parser you support.

Important

SDL requires that you test 100,000 malformed files for each file format your application supports. If you find a bug, the count resets to zero, and you must run 100,000 more iterations by using a different random seed so that you create different malformed files.

A generic file-fuzzing process

The process for fuzzing files is simple. It consists of the following steps:

1. Identify all the file formats your application supports.
2. Collect a library of valid files your application supports.
3. Malform (fuzz) a file.
4. Have the application consume the malformed file, and then observe the application.

The following paragraphs address each of these steps in detail.

Identify all valid file formats

The first step in the file-fuzzing process is to identify all file formats your application reads and handles. It's important to identify the formats your code handles. If the file data is handed off to a platform APIsuch as Microsoft Windows BitBlt or Graphics::DrawImage (Microsoft 2006a) or the open-source LibTIFF library (Still 2002)and the fuzzed file causes the application to fail, the bug is probably not in your code but rather in code developed by other parties. On failure, a stack trace will confirm this.

Tip

A useful list of extensions is the MIME-handler list file extension list. A MIME-handler is an application invoked when a file is double-clicked by a user. Such handlers must be fuzzed because they are a common social-engineering attack vector.

Your application should fail gracefully if faced with a file format it simply does not render or understand. Similarly, any component you have developed should fail gracefully and, just as important, bubble errors up to the next level of code.

Collect a library of valid files

You should gather as many valid files from as many trusted sources as possible, and the files should represent a broad spectrum of content. Aim for at least 100 files of each supported file type to get reasonable coverage. For example, if you manufacture digital photography equipment, you need representative files from every camera and scanner you build. Another way to look at this is to select a broad range of files likely to give you good code coverage.

Note that some file types are hard to fuzzfor example, files that are encrypted or have a digital signature associated with them. You should test with the signature checkingyou do check the signature, right?in your code enabled and disabled during fuzzing. In some cases, the procedure can be more complex than this if there are layers of code below your code that require decrypted data or a signature check before your code accesses the bits.

You should continue to build on this library over time as you define new formats or new format variants.

Malform a file

The work really starts when you begin malforming a file. You need to build or use a tool that chooses a file at random, malforms the file, and then passes the file to the software under test (van Sprundel 2005, Sutton and Greene 2005, Oehlert 2005).

The two broad classes of file fuzzing are smart fuzzing and dumb fuzzing. *Smart fuzzing* is when you know the data structure of the file format and you change specific values within the file. *Dumb fuzzing* is when you change the data at random. For example, PNG files start with a well-known signature followed by a series of blocks, named chunks (Milano 1999). The signature is 8 bytes long and must have the following value:

0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A

Each chunk has the following format:

- **4-byte length** The number of bytes in the data field
- **4-byte type** The name of the chunk (such as IHDR or IDAT)
- **n-byte data** The data, the format of which depends on the chunk type
- **4-byte CRC (cyclical redundancy check)** A CRC-32 calculated from the data

The IHDR chunk type always follows the signature, specifies image dimensions and color information, and has the following format:

- **4-byte width** Image width in pixels
- **4-byte height** Image height in pixels
- **1-byte bit depth** 1, 2, 4, 8, or 16 bits per pixel
- **1-byte color type** 0 for grayscale, 2 for RGB, 3 for palette, 4 for gray with alpha channel, and 6 for RGB and alpha
- **1-byte compression mode** Always zero
- **1-byte filter mode** Always zero
- **1-byte interlace mode** 0 for none and 1 for Adam-7 format

It's important to know also that PNG files structure multibyte integers with the most significant byte first. This structure is also called *big endian* (Cobas 2003).

Knowing the basic PNG format and the IHDR chunk type, you can be very specific about how you corrupt a PNG file. We'll give file-corruption examples a little later.

Dumb fuzzing is a shotgun approach: you take a valid file and randomly corrupt it. It really is that simple.

On the CD

We have included on the companion disc a simple file fuzzer named MiniFuzz (written in C++), which demonstrates the malforming process. This application outlines in code the steps required to fuzz a file by using dumb fuzzing and, to a lesser extent, smart fuzzing.

You can smart fuzz or dumb fuzz a file in many ways, including these:

- Making the file smaller than normal
- Filling the entire file with random data
- Filling portions of the file with random data
- Searching for null-terminated strings (in ASCII and Unicode) and setting the trailing null to nonnull
- Setting numeric data types to negative values
- Exchanging adjacent bytes
- Setting numeric data types to zero
- Toggling, setting, or clearing high bits (0x80, 0x8000, and so on)
- Doing an exclusive OR (XOR) operation on all bits in a byte, one bit at a time
- Setting numeric data types to 2^N +/1

Looking back at the PNG format, you could be very specific and smart fuzz a file by using the following techniques:

- Set the chunk length to a bogus value.
- Create random chunk names. (They are case sensitive, and the case has specific meaning.)
- Build a file with no IHDR chunk.
- Build a file with more than one IHDR chunk.
- Set the width, height, or color depth to invalid values (0, negative values, 2^N +/1, little endian, and so on).
- Set invalid compression, filter, or interlace modes.
- Set an invalid color type.

In the PNG example, you would also need to build a valid CRC for each malformed file; otherwise, a CRC failure would prevent most of the parsing code from being exercised.

For some file formats, you should also consider locking files, escaping data (for example, HTML encoding), and so on.

Consume the file and observe the application

Finally, have the application consume the filefor example, simply via a command-line argument that includes the file name. As the application runs, monitor for failures such as access violations or core dumps, and watch for spiked CPU usage. In Microsoft Windows, you can control a process and monitor it as it executes, by using job objects (Microsoft 2006b). Or you can set the fuzzing tool to be a mini-debugger by using debugging APIs

(Robbins 2003), and then you can write failure information to a log file. The sample fuzzer, MiniFuzz, shows how to do this.

Best Practices

You must refuzz your application every time you update the product or parser.

If the application fails, perhaps because of a buffer overrun or a null-pointer dereference, you can reload the malformed file at any point and run the application under a debugger if you are not already doing so. When the application fails, determine the source of the failure and fix the code. This strategy assumes the application does not store state between invocations.

Best Practices

Keep all malformed files that cause your application to fail. They will be needed to reproduce the bug. You can also use them later to verify that code regressions have not reintroduced the bug.

If any security bug is found in your product, keep the test cases that verify the existence and removal of the bug, and rerun these tests every time you rebuild the product to verify the bug has not reentered the code base. You should also consider using publicly known exploit code as a test case. Obviously, you should run such code on computers that are not connected to your production network.

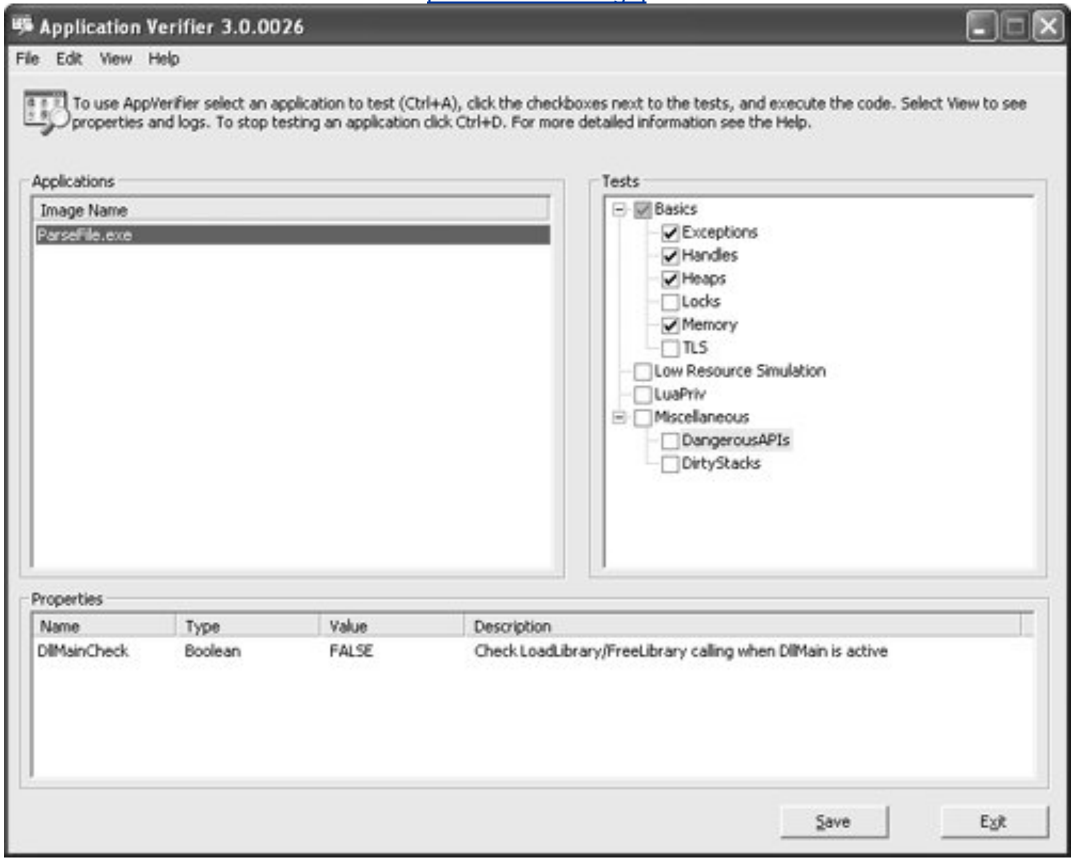
You should also watch for increased memory usage over time; this might indicate a memory leak. This leads to an important point: you should always run the application that you are testing under a debugger or similar tool. At Microsoft, we usually use a debugger such as cdb, or we run the code under Application Verifier (AppVerif) (Howard 2003). AppVerif is a test tool that is part of the Application Compatibility Toolset, which helps developers find bugs at run time. For example, AppVerif will catch the following classes of errors:

- Heap-based memory leaks and overwrites
- Uninitialized variables (dirty stacks)
- Dangerous APIs
- Thread local storage issues
- Lock usage

When fuzz testing an application, you must enable heap checking. It is recommended that you enable first-chance exceptions and handle checking, as shown in Figure 12-1.

Figure 12-1. AppVerif set to use the SDL required and recommended fuzz-testing settings.

[View full size image]



To use AppVerifier select an application to test (Ctrl+A), click the checkboxes next to the tests, and execute the code. Select View to see properties and logs. To stop testing an application click Ctrl+D. For more detailed information see the Help.

Applications

Image Name
ParseFile.exe

Tests

- ✓ Basics
  - ✓ Exceptions
  - ✓ Handles
  - ✓ Heaps
  - ☐ Locks
  - ✓ Memory
  - ☐ TLS
- ☐ Low Resource Simulation
- ☐ LuaPriv
- ☐ Miscellaneous
  - ☐ DangerousAPIs
  - ☐ DirtyStacks

Properties

Name	Type	Value	Description
DllMainCheck	Boolean	FALSE	Check LoadLibrary/FreeLibrary calling when DllMain is active

Note

AppVerif is a tool for unmanaged Microsoft Win32 code, not managed .NET code.

You inform AppVerif that your application is to be monitored, by using the following command line:

```
Appverif /verify myapp.exe
```

The /verify command-line argument will enable the following checks:

- HANDLE_CHECKS
- RPC_CHECKS

- COM_CHECKS
- LOCK_CHECKS
- FIRST_CHANCE_EXCEPTION_CHECKS
- FULL_PAGE_HEAP

Once the tests are complete, use the following command line to stop AppVerif from monitoring your application:

```
Appverif disable * -for myapp.exe
```

Next, you can read the log files by first exporting them from their native binary form to text, by using this:

```
Appverif export log for myapp.exe with to=c:\logs\myapp.log.txt
```

It is recommended that you also run the application using fault injection:

```
Appverif /verify myapp.exe /faults
```

There is a good chance that this option will exercise different code paths within the application as it runs.

Note

The fault injection option mimics certain common functions failingfor example, file I/O, memory allocations, and registry access.

Note that AppVerif does not test the application; it simply intercepts function calls from the application to the operating system when you run your test suite.

Best Practices

When fuzz testing an application, run the application under AppVerif to detect faults earlier.

Fuzzing Network Protocols

Fuzzing network connections is both similar to and different from fuzzing file formats. It is similar in that you use an application to create malformed data. But rather than creating files, you drop malformed packets on the network to attack a process listening on a network port such as a TCP (Transmission Control Protocol) or UDP

(User Datagram Protocol) socket (Nuwere and Varpiola 2005), an RPC endpoint, or a pipe. You should perform network fuzzing on a private subnet because otherwise you could accidentally attack production servers.

Note

The more message sequences a protocol uses, and hence the more state it stores or updates, the harder it is to reach the "deep" parts of the protocol.

There are three effective ways to fuzz network traffic:

- Create bogus packets.
- Record-fuzz-replay packets.
- Malform packets just before they are placed on the network or right after they are read from the network.

We'll look at each method in the following sections.

Create bogus packets

To create bogus packets, you must have a good understanding of the network protocol format because you build malformed packets based on the format. Let's look at an example. Microsoft SQL Server 2000 can listen on two ports: TCP/1433 and UDP/1434. The latter port is used by SQL Server to help determine which instances of SQL Server 2000 are available on a computer. A serious bug in the code listening on this port led to the Slammer worm (Boutin 2003). The data format is pretty straightforwardit's a one-byte "verb" followed by a short null-terminated string. The value of the first byte can be only 1 through 9.

The following code excerpt, written in Perl, shows how you can create bogus UDP packets and send them to a remote computer. The first byte is a random value from 1 through 9, and the string is just a series of random bytes from 1 to 2,048 bytes long.

```
use Strict;
use Socket;

die "Usage: <host> [port]\n" if !$ARGV[0];

my $server = $ARGV[0];
my $port = 1434;
print "Connecting to $server:$port\n";

srand 31337; # to make it easy to repro tests
while (1) {
    # create socket
    my $sock;
    socket(sock, PF_INET, SOCK_DGRAM, getprotobyname('udp')) || die "$!";
    my $iaddr = gethostbyname($server);
    my $sin = sockaddr_in($port, $iaddr);

    # build packets
    # format of packet is 1-byte verb (1..9) and n-byte string
    my $packetsize = int(rand(2048));
    my @chars= ('a'..'z','A'..'Z',0..9,qw(! @ # $ % ^ & * - _ = +));
    my $junk = join("",@chars[map{rand @chars} (1 .. $packetsize)]);
    my $verb = 1 + int(rand(9));
    $junk = pack("ca" . $packetsize,$verb,$junk);

    # lob the grenade
    print "Sending $packetsize bytes, verb $verb\r";
    send(sock, $junk, 0, $sin);
}
```

This code will crash a computer running an unpatched SQL Server 2000 in an average of three seconds. Obviously, some data formats are more complex than this format used by SQL Server 2000, but building fuzzers for complex protocols is more important than building fuzzers for simple protocols because complex protocols are potentially more buggy.

Record-fuzz-replay packets

In this testing scenario, you capture valid network packets using a network packet sniffer, fuzz the packets of interest to you, and then replay them. For example, you would perform the following steps if you wanted to test a Web application:

1. Enable a packet sniffer on the subnet.
2. Collect a few thousand (or more) HTTP packets.
3. Fuzz the HTTP packets in the recorded file.
4. Replay the fuzzed HTTP packets.

Tools such as Cenzic's Hailstorm can automate much of this process (Cenzic 2006).

Malforming packets on the fly

Another effective technique for fuzzing network traffic is to tweak packets within the application before sending the packets to the destination computer. This is a classic man-in-the-middle attack. To do this, you can place stub code in your network code to tweak the data at random. The following example is from a WinSock client application:

```
int SendData(SOCKET socket, const char *pBuf, size_t cbBuf) {
    ...
    int nRet = send(socket,
                    pBuf,
                    cbBuf,
                    0);
```

As you can see, this code simply sends string data to another socket. But with a small change, the code could be turned into fuzz code designed to stress-test the destination application code.

```
#ifdef __FUZZ__
void Fuzz(char *pBuf, size_t *pcbBuf) {
    const size_t FUZZ_THRESHOLD = 2; // corrupt 2% of packets
    if (rand() % 100 <= FUZZ_THRESHOLD) {
        // Fuzz data
    }
}
#endif // __FUZZ__

int SendData(SOCKET socket, char *pBuf, size_t cbBuf) {
    ...

#ifdef __FUZZ__
Fuzz(pBuf,&cbBuf);
#endif // __FUZZ__

    int nRet = send(socket,
                    pBuf,
                    cbBuf,
                    0);
```

Of course, make sure you disable the fuzzing code before you ship your software to customers.

As we mentioned before, you can fuzz the data in numerous ways, such as by flipping high bits, writing random data, truncating the data stream, setting null-terminators or other sentinel characters to invalid characters, and so on.

Best Practices

When performing network fuzz testing, don't fuzz only data that goes from the client to the server; build a rogue server to fuzz data going from the server to the client.

Some formats require that you be methodical while fuzzing. Take HTTP as an exampleyou should fuzz all the subcomponents of a valid HTTP payload. For example, you could:

- Add invalid headers (headers that are too long or too short or that have invalid characters or names, and so on).
- Change header fields (delete them, make them too long or too short, tweak characters, or add invalid characterssuch as by setting the Content-Length header to a negative value [SecurityTracker 2006]).

Tip

Setting a value to 1 is often a very effective way to change header fields because many server implementations add 1 to this value to accommodate for a trailing null. Server implementations then use this value to allocate heap memory.

- Duplicate valid headers.
- Remove or tweak sentinel characters such as the CR/LF combinations.
- Change the HTML content (fill it with random data, randomly truncate it, tweak bytes at random, and so on).

Be wary when fuzzing XML payloads, such as SOAP traffic, because you might end up testing the XML library's schema validation code and not your code. The same caveat applies to RPC code.

One of the advantages of file and network fuzzing is that once the test harness is constructed and operational, little human effort is required to keep the process running. Code reviews, on the other hand, require constant and expensive human attention. Of course, fuzzing does not replace code reviews, but it does effectively augment the security process.

Miscellaneous Fuzzing

Any code that consumes, manipulates, or analyzes data structure that comes from untrusted sources must be parsed. Good examples are arguments to ActiveX controls or any mobile code such as Java, Macromedia Flash, or .NET code that could be hosted in a browser. Identify all methods and properties to the code, and fuzz them all systematically. A simple way to do this is to build an HTML page that instantiates the mobile code, and then use JavaScript to fuzz all the methods and properties owned by the mobile code. Even the URL that hosts the mobile control can be fuzzed. For example, make the hosting URL very long (longer than 1,000 characters) and see if the mobile code can handle that if it checks to see which URL it was loaded from.

Fixing Bugs Found Through Fuzz Testing

Table 12-1 and Table 12-2 outline conservative triage bars for bugs found through fuzz testing. These outlines are likely to evolve as new bug classes are discovered, so please treat them as *minimum* bars and err on the side of caution.

Table 12-1. Client Code Bug Bar

Category	Errors
Must Fix	• Write Access Violation • Read Access Violation on extended instruction pointer (EIP) register
Must Investigate (Fix is probably needed.)	• Large memory allocations • Integer Overflow • Custom Exceptions • Other system-level exceptions that could lead to an exploitable crash • Read Access Violation using a REP (repeat string operation) instruction where ECX is large (on Intel CPUs) • Read Access Violation using a MOV (Move) where ESI, EDI, and ECX registers are later used in a REP instruction (on Intel CPUs)
Security issues unlikely (Investigate and resolve as a potential reliability issue according to your own triage process.)	• Other Read Access Violations not covered by other code areas • Stack Overflow exception (This is stack-space exhaustion, not a stack-based buffer overrun.) • DivideByZero • Null dereference

Table 12-2. Server Code Bug Bar

Category	Errors
Must Fix	• All errors leading to an elevation of privilege (EoP) or a significant denial of service (DoS). This is generally everything in all sections of the client fix bar, with the exception of large memory allocations (potentially).
Must Investigate (Fix is probably needed.)	• Large memory allocations.
Security issues unlikely (Investigate and resolve as potential reliability issue according to your own triage process.)	• None.

It is important to understand that every crash or unexpected error is an indication of an implementation issue, so you should always investigate such abnormalities. After you understand the problem, draw security-related conclusions. *Never* underestimate your enemyif you don't test your code, somebody else will do it for you and with harsher consequences.

If you hit a large number of bugs in the code by fuzz testing, stop what you're doing and start a deep code review. Code reviewing and fuzzing work very well together. If you find no bugs even after 100,000 iterations, we recommend that you make sure your fuzzing tool is creating invalid files. If you have access to code coverage tools, verify that the fuzzing is exercising large code areas. If all looks well, congratulations are in order!

Finally, never lose sight of the value a small number of computers has to fuzz testing. Iterating 100,000 or more files can be time consuming, so dedicating a small number of machines to the fuzzing process is beneficial.

Let's now turn our attention to other forms of security testing required by SDL.

PREV NEXT