Chapter 11
Local Fault Injection

This chapter focuses on the approach and techniques used to test the security of local applications. It begins by describing local resources and interprocess communication, which make up a local application's attack surface. After describing how to enumerate the local resources an application depends on, this chapter describes methods of testing several of those types of resources. It also describes how to test ActiveX objects, command-line programs, and applications' use of local files and shared memory.

Local Resources and Interprocess Communication

Modern operating systems offer a number of facilities for data input, sharing, and storage. An application's threat model (see Chapter 4, "Risk-Based Security Testing: Prioritizing Security Testing with Threat Modeling") must identify the local system resources that the application depends on and identify which of those may be controlled or affected by an attacker. We refer to this as the application's *local attack surface.* For example, every application depends on the executable file and shared libraries that make up the application's code. These files are typically protected by the operating system against modification by unprivileged users through file system permissions and access control lists (ACLs). However, if the application includes a directory that is writable by unprivileged users in its shared library load path, a local attack may be possible by creating a rogue shared library in this directory. See Chapter 8, "Web Applications: Common Issues," in the section "Uploading Executable Content (ASP/PHP/bat)," for more details on shared library load paths and this style of attack.

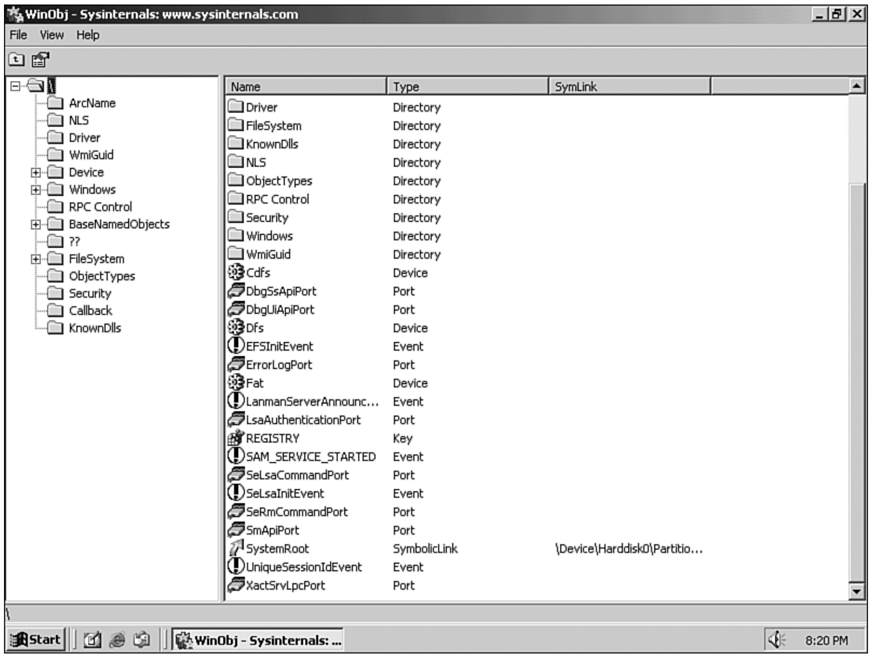
If the application depends on shared libraries that are loaded from outside the operating system's shared library directories, or if it uses the API functions `LoadLibrary` (Win32) or `dlopen` (UNIX), these directories and how the application loads libraries or plug-ins must be considered part of the application's local attack surface and must be tested. For example, the SAP DB database had a local privilege escalation vulnerability through this mechanism.[1] The database installed a Windows service configured to run as SYSTEM, launched from a directory that was writable by any user on the system. If a user placed a shared library named NETAPI32.DLL in this directory, it would be loaded into the privileged service, resulting in a privilege escalation vulnerability.

In the Internet age of rapid communication, files are seamlessly copied from Web servers or file servers to local files and are opened by local applications. This movement makes file parsing vulnerabilities much more serious when files that may have been corrupted by a malicious user are opened in applications not hardened against this sort of attack. For example, consider an application that processes files provided by remote users such as a Web browser or e-mail client. If the application in question opens potentially corrupted files provided by untrusted users, errors in the processing of these files may result in an application crash or, even worse, code execution. In this case, the application code that processes these files forms an important part of the application's local attack surface area and must be closely examined for vulnerabilities and tested using a technique such as file format fuzzing, as described later.

Windows NT Objects

A local application may also use a number of other resources, including command-line arguments, environment variables, or interprocess communication objects such as shared memory or semaphores. Windows NT-based operating systems, for example, support a number of potentially shareable local resources that are often called *objects*. These objects include files, devices, network sockets, shared memory segments, Registry keys, processes, threads, and more. Each of these objects may have a name so that other processes can refer to it and an ACL defining the object's access permissions to other users of the system.

The NT Kernel Object Manager maintains a file system-like namespace for these objects. A process may open an object by passing one of these names to a system function such as `CreateFile`. You can browse this namespace by making calls to the Windows NT Native API functions in NTDLL or by using a browsing utility such as WinObj. WinObj is a freeware utility from Sysinternals Freeware. It lists the NT Object Manager namespace and allows the user to browse the namespace and query object attributes and permissions. Figure 11-1 shows the objects and object directories in the namespace's root directory.



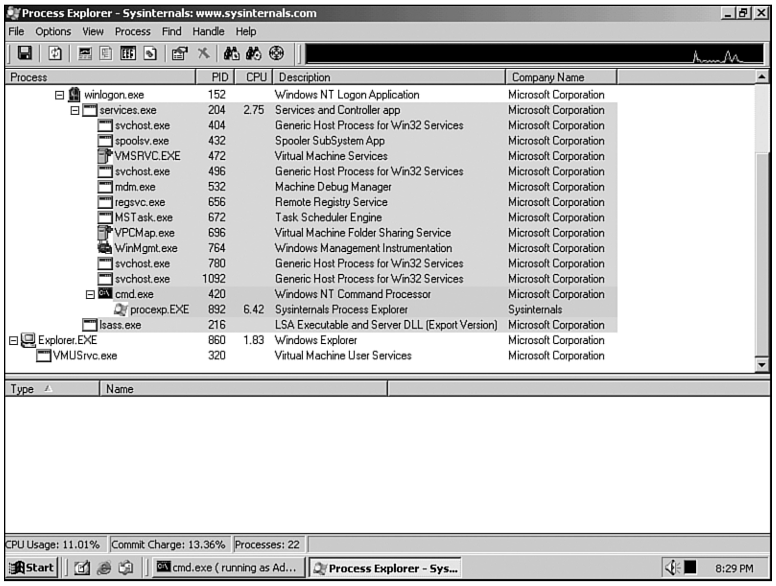
Name	Type	SymLink
Driver	Directory	
FileSystem	Directory	
KnownDlls	Directory	
NLS	Directory	
ObjectTypes	Directory	
RPC Control	Directory	
Security	Directory	
Windows	Directory	
WmiGuid	Directory	
Cdfs	Device	
DbgSsApiPort	Port	
DbgUiApiPort	Port	
Dfs	Device	
EFSInitEvent	Event	
ErrorLogPort	Port	
Fat	Device	
LanmanServerAnnounc...	Event	
LsaAuthenticationPort	Port	
REGISTRY	Key	
SAM_SERVICE_STARTED	Event	
SeLsaCommandPort	Port	
SeLsaInitEvent	Event	
SeRmCommandPort	Port	
SmApiPort	Port	
SystemRoot	SymbolicLink	\Device\Harddisk0\Partitio...
UniqueSessionIdEvent	Event	
XactSrvLpcPort	Port	

Figure 11-1 WinObj by Sysinternals

These objects are managed inside the NT kernel, and a user process may not manipulate them directly. The process may request indirect access, however, to a system object through Windows API functions such as `CreateFile`, `CreateProcess`, and `CreateEvent`. These functions return a 32-bit integer value that the process may pass as a parameter to other API functions to refer specifically to the corresponding NT object. This value, called an *object handle*, is unique within the process and refers to an open or active NT object. The handle is also *opaque*, meaning that it is not intended to be directly manipulated by the process and has meaning only to the facilities under the hood of the Windows API.

Handles and the objects they refer to are not necessarily private to the process that created them. As mentioned previously, NT objects have an ACL that defines what actions are allowed and by whom. At creation, the programmer may create an ACL for the object, or else a default one is used. The default ACL may be more permissive than necessary. Depending on how the application uses this object, this may present a security risk. For example, a shared memory segment could contain complex data structures involving arrays, indexes, pointers, and memory buffer lengths. If an attacker can modify these structures stored in a shared memory segment, she could crash or take over another process using this shared memory segment. See the section "Shared Memory" later in this chapter for a description of how to list, examine, and test shared memory segments.

Two tools from Sysinternals Freeware let you list the open handles on the system. Handle and Process Explorer, its graphical counterpart, list handles for a given process or for all processes on the system (see Figure 11-2). You can use these tools to identify NT object-based resources that an application depends on and view the ACLs for them. If an application uses or creates an object that other users may access, the application security architect must determine whether the access permissions can be tightened. For example, try to make all created objects accessible only to the user who created them. If the resource must remain accessible to other users, the application's use of it should be scrutinized and tested where possible.


Process Explorer - Sysinternals: www.sysinternals.com
File
Options
View
Process
Find
Handle
Help
Start
Process Explorer - Sys...
8:29 PM

Process	PID	CPU	Description	Company Name
winlogon.exe	152		Windows NT Logon Application	Microsoft Corporation
services.exe	204	2.75	Services and Controller app	Microsoft Corporation
svchost.exe	404		Generic Host Process for Win32 Services	Microsoft Corporation
spoolsv.exe	432		Spooler SubSystem App	Microsoft Corporation
VMSRVC.EXE	472		Virtual Machine Services	Microsoft Corporation
svchost.exe	496		Generic Host Process for Win32 Services	Microsoft Corporation
mdm.exe	532		Machine Debug Manager	Microsoft Corporation
regsvc.exe	656		Remote Registry Service	Microsoft Corporation
MSTask.exe	672		Task Scheduler Engine	Microsoft Corporation
VPCMap.exe	696		Virtual Machine Folder Sharing Service	Microsoft Corporation
WinMgmt.exe	764		Windows Management Instrumentation	Microsoft Corporation
svchost.exe	780		Generic Host Process for Win32 Services	Microsoft Corporation
svchost.exe	1092		Generic Host Process for Win32 Services	Microsoft Corporation
cmd.exe	420		Windows NT Command Processor	Microsoft Corporation
procexp.EXE	892	6.42	Sysinternals Process Explorer	Sysinternals
lsass.exe	216		LSA Executable and Server DLL (Export Version)	Microsoft Corporation
Explorer.EXE	860	1.83	Windows Explorer	Microsoft Corporation
VMUSrvc.exe	320		Virtual Machine User Services	Microsoft Corporation

Type	Name

CPU Usage: 11.01% | Commit Charge: 13.36% | Processes: 22

cmd.exe (running as Ad...

Figure 11-2 Process Explorer listing running processes

UNIX set-user-id Processes and Interprocess Communication

The security of locally executed programs is especially important under UNIX-like operating systems where the executables can be marked to execute as *set-user-id* or *set-group-id*. When these permissions are assigned to an executable, the executable assumes the privilege of the owning user or group, respectively. They no longer run under the privileges of the user who invoked them. Instead, they run under the privileges of the user or group who owns the file. To allow unprivileged users to perform controlled actions on the system that would normally require privileged access, set-user-id (suid) *root* executables are commonly provided to perform that functionality on behalf of the unprivileged user. These executables have historically been the largest source of vulnerabilities in the UNIX family of operating systems.

The central insecurity in suid executables is that the executable runs with a higher privilege in an environment partially controlled by the unprivileged user. The user may control command-line arguments, environment variables, open file descriptors, and other resources that the privileged executable may inherit. If the privileged executable improperly trusts or uses these resources, it may result in a security vulnerability. Buffer overflows resulting from long command-line arguments and environment variables have been very common historically, but there have also been a number of subtle vulnerabilities resulting from the use of other inherited resources, such as file descriptors and file mode mask (*umask*).

Under UNIX, new programs are launched through the *fork/exec* model. The system call `fork()` creates a new process, and the system call `exec()` runs an executable in the current process. To launch a new program, the program calls `fork` to create a new process and then calls `exec` in the new (child) process to load and run the new executable. The new program inherits a number of attributes from the parent process:

- Environment variables
- Command-line arguments
- Open file descriptors (except for those marked close-on-exec)
- Process ID
- Parent process ID
- Process group ID
- Access groups
- Working directory
- Root directory

- Controlling terminal
- Resource usages
- Interval timers
- Resource limits
- File mode mask
- Signal mask

The most common type of local vulnerabilities under UNIX-like operating systems are buffer overflows caused by improperly handling command-line arguments and environment variables. The section "Command-Line Utility Fuzzing" describes how to test local applications for these sorts of vulnerabilities. If the application installs any set-user-id executables, it is crucial that they be tested for these vulnerabilities. Even if the executable is not marked as set-user-id, it may be launched with arguments crafted by a remote user and should be tested. This often occurs, for example, when a user's Web browser may automatically launch small helper programs to handle downloaded files.

Threat-Modeling Local Applications

Threat-modeling local applications involves three basic steps:

- Enumerate local resources used by the application
- Determine access permissions of shared or persistent resources
- Identify the exposed local attack surface area

The first step in threat-modeling a local application is enumerating the application's local resource dependencies. These may include command-line arguments used to launch the application, environment variables, files, Registry values, Windows stations and desktops, and synchronization primitives.

The best way to do this is to examine the application source code to identify each resource that may potentially be used. This information is extremely valuable to properly determine an application's attack surface. Unfortunately, this information is not often maintained, and the knowledge of the resources an application requires may be distributed among multiple developers, each responsible for different components of the application. In cases like these, or when application source code is unavailable, application behavior monitoring tools such as API Monitor, Process Explorer, lsof, and ktrace may be the best way to enumerate the local resources an application uses. API Monitor and lsof are discussed next. See

Chapter 4 for a more detailed discussion of how to use runtime monitoring tools to identify the resources an application uses and depends on.

Enumerating Windows Application Resources

As discussed, Process Explorer and Handle can be used to identify the resources used by Windows applications. Process Explorer and Handle list the resources currently in use by an application. Many applications, however, check for the existence of files and other resources or may use them periodically or briefly at startup. It is important to identify the dependence on these resources as well. Tools that monitor and log Windows API usage should be used to list all the resources an application uses over its lifetime. A tool such as API Monitor (http://www.rohitab.com/apimonitor/index.html) may be used for this.

API Monitor logs access to a large number of functions grouped into categories such as Processes and Threads, File I/O, Device Input and Output, and Handles and Objects. API Monitor can be used to log API calls from a running application or a newly launched one. Examining this log tells you what local resources the application uses and how. For example, you can find temporary files created by the application that may be susceptible to race-condition attacks or other vulnerabilities in file usage.

As soon as you have a list of the local resources being used, you must identify their access permissions. As discussed, most NT objects have an ACL that defines what access is permitted by whom. You can examine an object's permissions in WinObj or Process Explorer. You also can observe the API function call used to create this object and see if a security descriptor was provided for the object or whether NULL was supplied for this parameter and a default security descriptor assigned. Often the default security descriptor adequately protects the object, but not always.

Enumerating UNIX Application Resources

UNIX operating systems typically include a large number of small tools for process monitoring. For example, the tool lsof is included in most Linux distributions and MacOS X and is an optional installation for Sun Solaris. lsof, whose name means "list open files," can be used to list the files and network sockets that a running process is using. It is roughly analogous to Process Explorer for Windows. An example of lsof usage is shown in Listing 11-1.

Listing 11-1
`lsof` *Output of ATSServer on MacOSX 10.4*

```
% lsof
COMMAND      PID    USER      FD      TYPE      DEVICE  SIZE/OFF        NODE        NAME
ATSServer    104     ddz     cwd      VDIR        14,2      1156           2        /
ATSServer    104     ddz      0r      VCHR         3,2       0t0    51764740        /dev/null
ATSServer    104     ddz      1w      VCHR         3,2       0t0    51764740        /dev/null
ATSServer    104     ddz      2w      VCHR         3,2  0t231860    51764740        /dev/null
ATSServer    104     ddz      3r    PSXSHM  0x034269e4      4096                    obj=0x03511f70
ATSServer    104     ddz      4r    PSXSHM  0x03424924      4096                    obj=0x0347da28
ATSServer    104     ddz      5u      VREG        14,2    229376      393913
/Library/Caches/com.apple.ATS/502/filetoken.db
ATSServer    104     ddz      6u      VREG        14,2    126976      393915
/Library/Caches/com.apple.ATS/502/fonts.db
ATSServer    104     ddz      7u      VREG        14,2     40960      393916
/Library/Caches/com.apple.ATS/502/qdfams.db
ATSServer    104     ddz      8u      VREG        14,2     40960      393917
/Library/Caches/com.apple.ATS/502/annex.db
ATSServer    104     ddz      9u      VREG        14,2   5612836      393918
/Library/Caches/com.apple.ATS/502/annex_aux
ATSServer    104     ddz     10r      VREG        14,2   1135530        5452
/System/Library/Frameworks/ApplicationServices.framework/Versions/A/
Frameworks/ATS.framework/Versions/A/Resources/SynthDB.rsrc
```

From this lsof output, you can identify some of the local resources that ATSServer depends on. For example, the output lists two POSIX shared memory segments (type `PSXSHM` in the lsof output) in use and a number of files (type `VREG`). If a user other than the process owner can modify any of these resources, they should be tested using the methodologies described in the sections on file and shared memory fuzzing.

To identify the command-line arguments that an application expects, you can examine the call to the `getopt()` function:

```
int getopt(int argc, char * const argv[], const char *optstring);
```

To use `getopt`, the application passes a value, `optstring`, that encodes the command-line switches it expects and whether it expects an argument with that option. For example, rlogin calls `getopt` with an `optstring` of `8EKLde:l:`, which indicates that rlogin accepts boolean switches `-8`, `-E`, `-K`, `-L`, and `-d`, whereas options `-e` and `-l` require arguments. Command-line switches that accept arguments are typically the best options to fuzz, as demonstrated later. If you only have access to the application binary, you can take advantage of the fact that the `optstring` follows a known format. You can use the `strings` command to identify ASCII strings in a binary file and `grep` to search for output that matches what an `optstring` usually looks like:

```
% strings /usr/bin/rlogin ¦ egrep '^[A-Za-z0-9:]+$' ¦ grep ":"
8EKLde:l:
```

The local resources that we have enumerated and found to be accessible by an attacker form the application's local attack surface. The local attack surface should be documented in the application's threat model and minimized wherever possible. The rest of this chapter discusses tools and techniques for locating security vulnerabilities in local attack surface areas.

Testing Scriptable ActiveX Object Interfaces

Attack Pattern: ActiveX Objects

- Identify application ActiveX objects that are safe for scripting.
- Enumerate the methods and properties exposed by the objects.
- Use COMbust or axfuzz to perform fault injection on object methods.

ActiveX is a Microsoft standard that allows software components to be written in and called from applications written in different programming languages. It is the successor to (and builds on) previous technologies such as OLE and COM and is used to implement Visual Basic controls and other plug-ins. ActiveX is also often used to implement rich, extended behavior in Web applications. Microsoft's Internet Explorer Web browser allows sites (as permitted by the zones security model) to supply a compiled ActiveX object to be downloaded and run within the browser. Because these components are compiled native code, they can do anything that an application can do to the user's machine. This results in both high levels of flexibility and a high risk of potential insecurity. An ActiveX *object* or *control* is a reusable software object that is implemented in an executable or shared library. Settings in the system Registry give the object a unique identifier (CLSID) and symbolic name (ProgID) so that other software may easily refer to and use the object. The object may implement one or more *interfaces*, which define which methods and properties the object supports. For more information on ActiveX, see Microsoft's online documentation in MSDN.[2]

An ActiveX control can be requested in an HTML page through the HTML `<OBJECT>` tag or from a `CreateObject()` call in JavaScript or VBScript. If ActiveX objects are allowed in the security zone assigned to the page, Internet Explorer proceeds to determine whether the object is considered "safe for scripting."[3] Internet Explorer first determines whether the object implements the `IObjectSafety` interface. The `IObjectSafety` interface contains a method called `SetInterfaceSafetyOptions` that allows an ActiveX container to query if the object is safe for scripting. If the object does not implement this interface, Internet Explorer checks the Registry under the object's CLSID's Implemented Categories section. Defined Registry keys may indicate that the object is safe for scripting. If the `IObjectSafety` interface is not implemented, if the control returns unsafe for any of these actions, or if the special Registry keys are not present, Internet Explorer does not let the object be used.

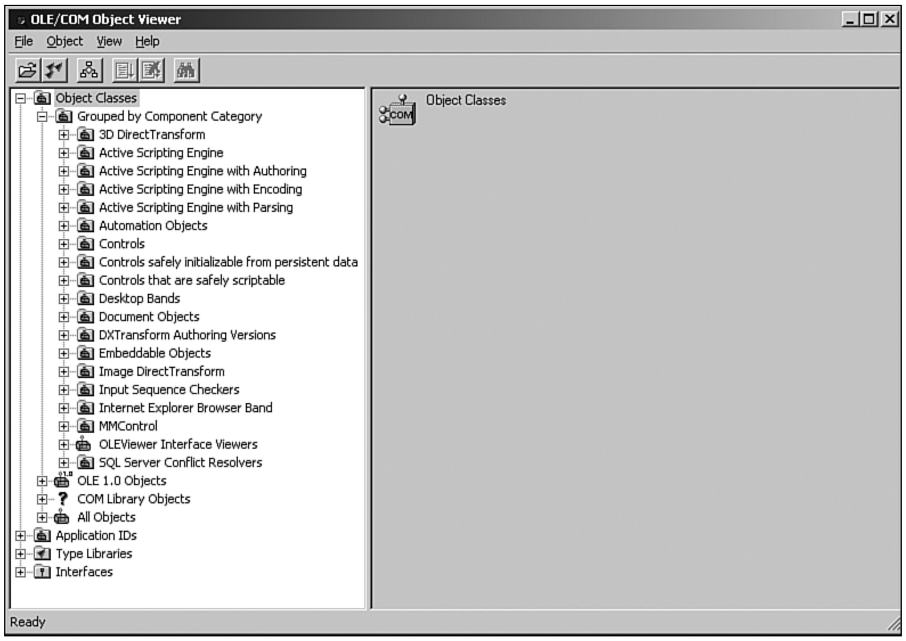
In the default configuration for Internet Explorer 6, Internet sites may initialize and script ActiveX objects that are already on the user's system and are marked safe for scripting. The user is prompted to download signed ActiveX objects, and the downloading of unsigned ActiveX objects is disabled.

To ensure that a scriptable ActiveX object installed by a trusted site or application cannot be abused by a hostile Web page, all its exposed methods must be tested and secured. This should be done for any custom scriptable ActiveX objects included with an application. The next section discusses identifying and testing ActiveX objects marked safe for scripting.

Identifying “Safe” Scriptable Objects

A number of tools are available for examining ActiveX controls and COM components. OLEView is a tool included with Microsoft Visual Studio that lists installed ActiveX, COM, and OLE objects and allows the user to view their properties and implemented interfaces. We use OLEView to determine whether a given ActiveX object is scriptable and marked “safe.”

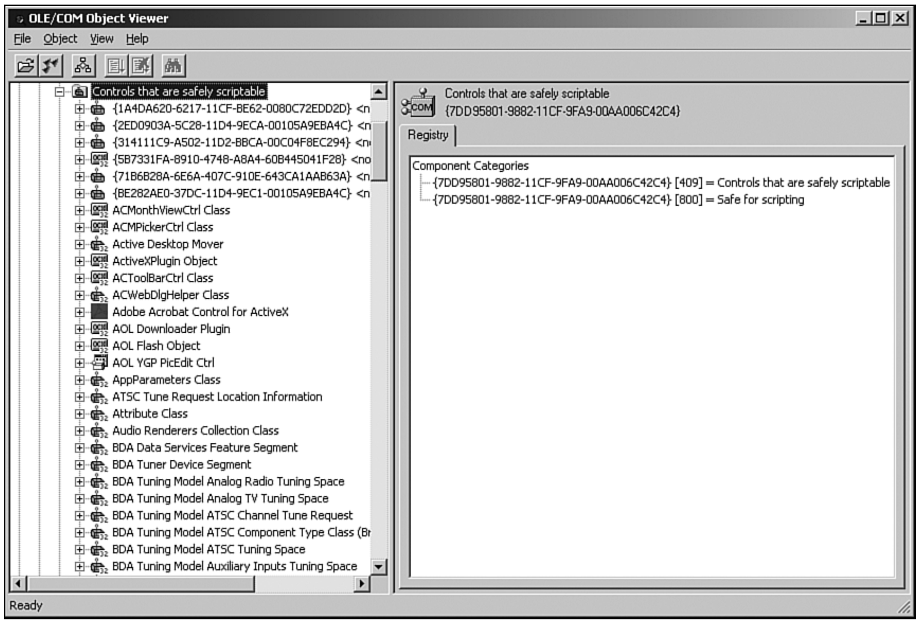
OLEView organizes all the OLE/COM objects on the system into a number of component categories, including .NET, Active Scripting, and Safely Scriptable (see Figure 11-3). We are most interested in the COM components marked safe for scripting because these are the ActiveX components that may be activated by a remote Web page. If there is a serious vulnerability in one of these ActiveX components, the remote page may be able to take full control of the user’s computer.



```
Object Classes
    Grouped by Component Category
        3D DirectTransform
        Active Scripting Engine
        Active Scripting Engine with Authoring
        Active Scripting Engine with Encoding
        Active Scripting Engine with Parsing
        Automation Objects
        Controls
        Controls safely initializable from persistent data
        Controls that are safely scriptable
        Desktop Bands
        Document Objects
        DXTransform Authoring Versions
        Embeddable Objects
        Image DirectTransform
        Input Sequence Checkers
        Internet Explorer Browser Band
        MMControl
        OLEViewer Interface Viewers
        SQL Server Conflict Resolvers
    OLE 1.0 Objects
    COM Library Objects
    All Objects
Application IDs
Type Libraries
Interfaces
```

Figure 11-3 OLEView displaying component categories

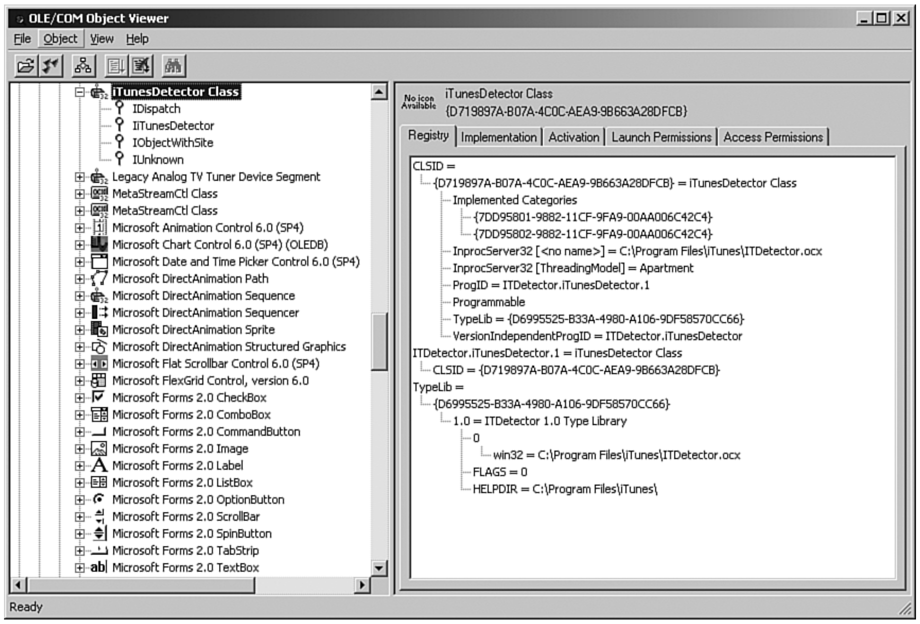
If you drill down into “Controls that are safely scriptable” (see Figure 11-4), you can easily enumerate all the components on the system that may be used by remote Web sites. Browsing through this list, you can examine individual controls and even list their interface and methods, looking for “interesting” methods or methods prone to security vulnerabilities (see Figure 11-5).



```
Controls that are safely scriptable
  {1A4DA620-6217-11CF-BE62-0080C72EDD2D} <n
  {2ED0903A-5C28-11D4-9ECA-00105A9EBA4C} <n
  {314111C9-A502-11D2-BBCA-00C04F8EC294} <n
  {5B7331FA-8910-4748-A8A4-60B445041F28} <no
  {71B6B28A-6E6A-407C-910E-643CA1AAB63A} <n
  {BE282AE0-37DC-11D4-9EC1-00105A9EBA4C} <n
  ACMonthViewCtrl Class
  ACMPickerCtrl Class
  Active Desktop Mover
  ActiveXPlugin Object
  ACToolBarCtrl Class
  ACWebDlgHelper Class
  Adobe Acrobat Control for ActiveX
  AOL Downloader Plugin
  AOL Flash Object
  AOL YGP PicEdit Ctrl
  AppParameters Class
  ATSC Tune Request Location Information
  Attribute Class
  Audio Renderers Collection Class
  BDA Data Services Feature Segment
  BDA Tuner Device Segment
  BDA Tuning Model Analog Radio Tuning Space
  BDA Tuning Model Analog TV Tuning Space
  BDA Tuning Model ATSC Channel Tune Request
  BDA Tuning Model ATSC Component Type Class (Br
  BDA Tuning Model ATSC Tuning Space
  BDA Tuning Model Auxiliary Inputs Tuning Space
```

```
Controls that are safely scriptable
{7DD95801-9882-11CF-9FA9-00AA006C42C4}

Component Categories
  {7DD95801-9882-11CF-9FA9-00AA006C42C4} [409] = Controls that are safely scriptable
  {7DD95801-9882-11CF-9FA9-00AA006C42C4} [800] = Safe for scripting
```

Figure 11-4 Listing of ActiveX controls marked safe for scripting



```
iTunesDetector Class
  IDispatch
  IITunesDetector
  IObjectWithSite
  IUnknown
Legacy Analog TV Tuner Device Segment
MetaStreamCtl Class
MetaStreamCtl Class
Microsoft Animation Control 6.0 (SP4)
Microsoft Chart Control 6.0 (SP4) (OLEDB)
Microsoft Date and Time Picker Control 6.0 (SP4)
Microsoft DirectAnimation Path
Microsoft DirectAnimation Sequence
Microsoft DirectAnimation Sequencer
Microsoft DirectAnimation Sprite
Microsoft DirectAnimation Structured Graphics
Microsoft Flat Scrollbar Control 6.0 (SP4)
Microsoft FlexGrid Control, version 6.0
Microsoft Forms 2.0 CheckBox
Microsoft Forms 2.0 ComboBox
Microsoft Forms 2.0 CommandButton
Microsoft Forms 2.0 Image
Microsoft Forms 2.0 Label
Microsoft Forms 2.0 ListBox
Microsoft Forms 2.0 OptionButton
Microsoft Forms 2.0 ScrollBar
Microsoft Forms 2.0 SpinButton
Microsoft Forms 2.0 TabStrip
Microsoft Forms 2.0 TextBox
```

```
iTunesDetector Class
{D719897A-B07A-4C0C-AEA9-9B663A28DFCB}

CLSID =
  {D719897A-B07A-4C0C-AEA9-9B663A28DFCB} = iTunesDetector Class
    Implemented Categories
      {7DD95801-9882-11CF-9FA9-00AA006C42C4}
      {7DD95802-9882-11CF-9FA9-00AA006C42C4}
    InprocServer32 [<no name>] = C:\Program Files\iTunes\ITDetector.ocx
    InprocServer32 [ThreadingModel] = Apartment
    ProgID = ITDetector.iTunesDetector.1
    Programmable
    TypeLib = {D6995525-B33A-4980-A106-9DF58570CC66}
    VersionIndependentProgID = ITDetector.iTunesDetector
ITDetector.iTunesDetector.1 = iTunesDetector Class
  CLSID = {D719897A-B07A-4C0C-AEA9-9B663A28DFCB}
TypeLib =
  {D6995525-B33A-4980-A106-9DF58570CC66}
    1.0 = ITDetector 1.0 Type Library
      0
        win32 = C:\Program Files\iTunes\ITDetector.ocx
      FLAGS = 0
      HELPDIR = C:\Program Files\iTunes\
```

Figure 11-5 Details of a third-party ActiveX component

A second tool you can use is axenum, part of the axfuzz project at http://axfuzz.sourceforge.net. By default, axenum enumerates all installed COM components on the system and queries them for the `IObjectSafety` interface and the "safe" Registry keys. The tool provides a quick way to automatically identify security-sensitive ActiveX controls. Listing 11-2 is a small sample of output from axenum. This output, for the `CrBlinds` object, shows the CLSID for the object, tells you whether the object is safe for scripting, and lists the object's methods and properties.

Listing 11-2
Sample of `axenum` *Output*

```
> CrBlinds
     {00C429C0-0BA9-11d2-A484-00C04F8EFB69}
     IObjectSafety:
     IO. Safe for scripting (IDispatch) set successfully
     IDispatch:GetInterfaceSafetyOptions Supported=1, Enabled=1
     ICrBlinds2:
        long Capabilities() propget
...
```

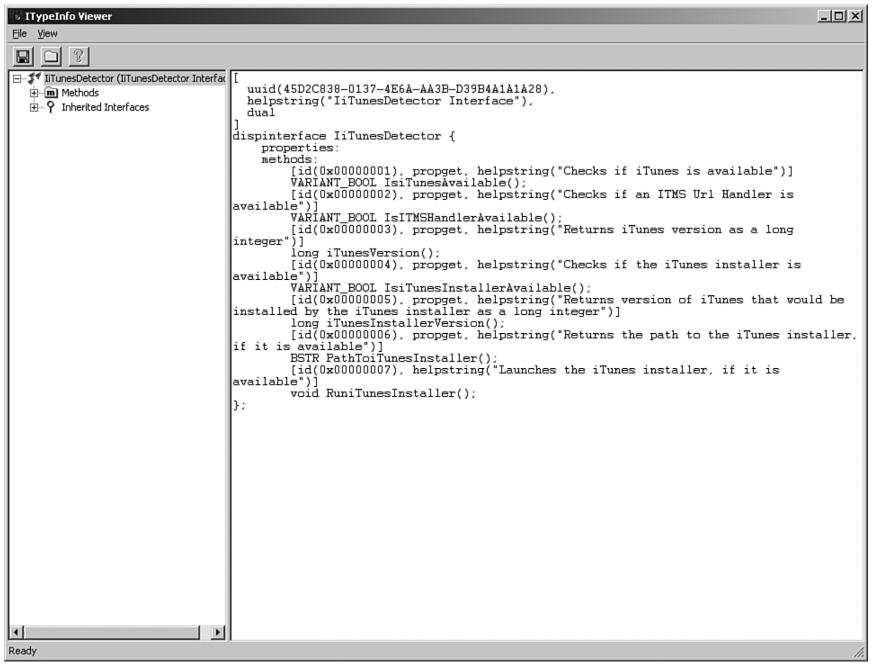
The other tool included in the project, axfuzz, enumerates all the objects safe for scripting and calls each available method with a 0 (NULL) value or long string for any BSTR values. Because the testing performed by axfuzz is rather simplistic, we recommend using something more thorough, such as COMbust (described in a moment).

Testing Object Interfaces

Manual Interface Testing

One of the best features of OLEview is its integrated interface browsing. If a typelib is available for an object, OLEview displays the type information in human-readable form. For example, Figure 11-6 shows the IDL type information for a third-party scriptable ActiveX component.

You can use this interface information to manually test the methods to identify their purpose and functionality. A simple way to do this is to create a Web page with some VBScript that creates the object and invokes the object's methods or retrieves properties. Listing 11-3 tests an object called `iTunesDetector`. This object can be used from a Web page to identify what version of iTunes (if any) is installed on the computer.



```
[
  uuid(45D2C838-0137-4E6A-AA3B-D39B4A1A1A28),
  helpstring("IiTunesDetector Interface"),
  dual
]
dispinterface IiTunesDetector {
    properties:
    methods:
        [id(0x00000001), propget, helpstring("Checks if iTunes is available")]
        VARIANT_BOOL IsiTunesAvailable();
        [id(0x00000002), propget, helpstring("Checks if an ITMS Url Handler is
available")]
        VARIANT_BOOL IsITMSHandlerAvailable();
        [id(0x00000003), propget, helpstring("Returns iTunes version as a long
integer")]
        long iTunesVersion();
        [id(0x00000004), propget, helpstring("Checks if the iTunes installer is
available")]
        VARIANT_BOOL IsiTunesInstallerAvailable();
        [id(0x00000005), propget, helpstring("Returns version of iTunes that would be
installed by the iTunes installer as a long integer")]
        long iTunesInstallerVersion();
        [id(0x00000006), propget, helpstring("Returns the path to the iTunes installer,
if it is available")]
        BSTR PathToiTunesInstaller();
        [id(0x00000007), helpstring("Launches the iTunes installer, if it is
available")]
        void RuniTunesInstaller();
};
```

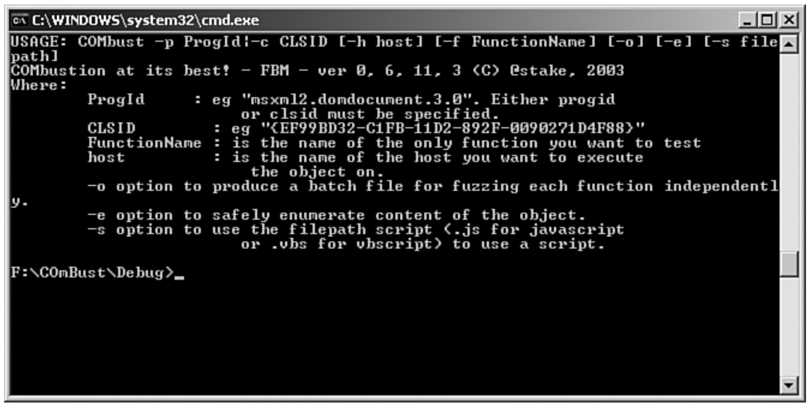
Figure 11-6 Interface of a third-party ActiveX component

Listing 11-3
Simple Web Page Demonstrating Manual Invocation of ActiveX Methods

```
<html>
<head>
<title>iTunesDetector Test</title>
</head>

<body>
<script language="VBScript">
<!—
  Dim itunes
  Set itunes = CreateObject("ITDetector.iTunesDetector.1")

  document.write("IsiTunesAvailableAvailable: " &
itunes.IsiTunesAvailable & "<BR>")
  document.write("IsITMSHandlerAvailable: " &
itunes.IsITMSHandlerAvailable & "<BR>")
  document.write("iTunesVersion: " & itunes.iTunesVersion & "<BR>")
  document.write("IsiTunesInstallerAvailable: " &
      itunes.IsiTunesInstallerAvailable & "<BR>")
—>
</script>
</body>
```

Automated ActiveX Interface Testing

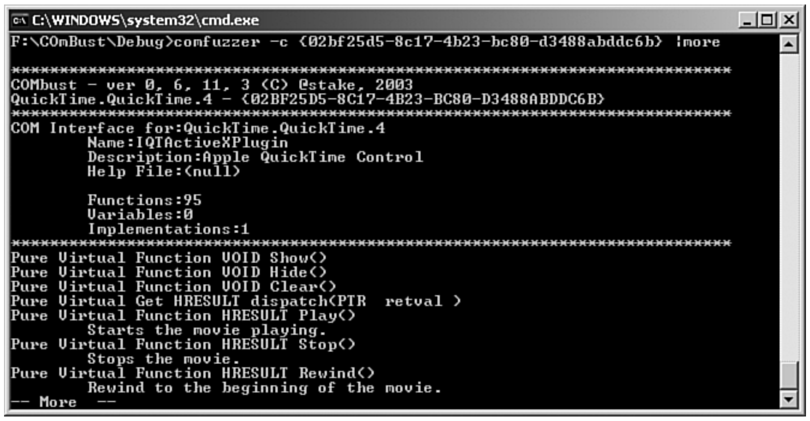
COMbust, released by Frederic Bret-Mounet at the BlackHat Briefings 2003, is a tool for automatically enumerating and fuzzing an ActiveX object's interface (see Figure 11-7). It requires that the object be scriptable, which is often the case because we are most interested in testing the security of objects that an untrusted user may interact with.



```
C:\WINDOWS\system32\cmd.exe
USAGE: COMbust -p ProgId|-c CLSID [-h host] [-f FunctionName] [-o] [-e] [-s file
path]
COMbustion at its best! - FBM - ver 0, 6, 11, 3 (C) @stake, 2003
Where:
        ProgId      : eg "msxml2.domdocument.3.0". Either progid
                        or clsid must be specified.
        CLSID        : eg "{EF99BD32-C1FB-11D2-892F-0090271D4F88}"
        FunctionName : is the name of the only function you want to test
        host         : is the name of the host you want to execute
                         the object on.
        -o option to produce a batch file for fuzzing each function independentl
y.
        -e option to safely enumerate content of the object.
        -s option to use the filepath script (.js for javascript
                        or .vbs for vbscript) to use a script.

F:\COmBust\Debug>_
```

Figure 11-7 COMbust usage

COMbust requires either the ClassID or ProgId specifying the object to test. For testing, you should use the CLSID or ProgId discovered in OLEview or axenum. Without any other arguments, COMbust automatically enumerates the interface to the object and begins testing each function provided by the object until it causes a crash. For this reason, using the `-o` option is recommended. This option causes COMbust to produce a batch file that runs COMbust on each function independently. That way, if a crash is caused in one function, the remaining functions are still tested. COMbust stops testing a function at the first crash it causes in it. Figure 11-8 shows COMbust being run against a QuickTime component installed with Apple's iTunes software. This component was chosen because it has a large number of methods and properties. This run of COMbust did not find any crashes. When COMbust does find a crash, the line `CRASH!!!` is displayed in the output, and COMbust stops further testing.



```
C:\WINDOWS\system32\cmd.exe
F:\COmBust\Debug>comfuzzer -c {02bf25d5-8c17-4b23-bc80-d3488abddc6b} |more

****************************************************************************
COMbust - ver 0, 6, 11, 3 (C) @stake, 2003
QuickTime.QuickTime.4 - {02BF25D5-8C17-4B23-BC80-D3488ABDDC6B}
****************************************************************************
COM Interface for:QuickTime.QuickTime.4
        Name:IQTActiveXPlugin
        Description:Apple QuickTime Control
        Help File:(null)

        Functions:95
        Variables:0
        Implementations:1
****************************************************************************
Pure Virtual Function VOID Show()
Pure Virtual Function VOID Hide()
Pure Virtual Function VOID Clear()
Pure Virtual Get HRESULT dispatch(PTR  retval )
Pure Virtual Function HRESULT Play()
        Starts the movie playing.
Pure Virtual Function HRESULT Stop()
        Stops the movie.
Pure Virtual Function HRESULT Rewind()
        Rewind to the beginning of the movie.
-- More  --
```

Figure 11-8 Running COMbust

Evaluating Crashes

COMbust may turn up crashes that result from a number of conditions: null pointer dereferences, buffer overflows, object initialization and state errors. Some of these are more serious security vulnerabilities than others. For example, addressing a potentially exploitable buffer overflow that would allow code execution should be prioritized over a simple null pointer dereference that causes only denial of service. Chapter 12, "Determining Exploitability," discusses in depth how to evaluate how exploitable a program crash may be.

Fuzzing File Formats

Applications such as Web browsers, image viewers, and media players regularly process files provided by untrusted remote users. The formats and encoding of these files, especially those used for compressed images, video, and audio, are quite complex and thus are difficult to parse securely. It is therefore essential that the applications' processing of these files be properly scrutinized and tested.

As an example of a common file format vulnerability, consider the following code fragment. It is an example of a style of code commonly seen parsing binary file formats. The file format may consist of a file header and a number of sections, each with section headers. Each section header contains a section size field that describes how many bytes of data are contained within that section. If the file format parsing code uses these values unchecked in a memory allocation request size or as an offset into the file, a denial-of-service or memory trespass vulnerability may be likely. The following code does not check the section size field read from the file section header. It reads file data into a heap-allocated data buffer without validating the size or checking the return value of `HeapAlloc`. This presents several problems (see Listing 11-4).

Listing 11-4
A Common Binary File Format Parsing Vulnerability

```
FILE_HEADER fh;
SECTION_HEADER *sh;

ReadFile(hFile, &fh, sizeof(FILE_HEADER));
sh = HeapAlloc(fh.dwSectionSize + SIZEOF(SECTION_HEADER));
ReadFile(hFile, sectionData, fh.dwSectionSize);
```

Consider the case in which the value of the section size field read in the file header is very large. If the allocation fails and a buffer cannot be allocated, `HeapAlloc` returns NULL. When the application calls `ReadFile` with a nonzero size and a NULL buffer pointer, the application crashes with an access violation. This causes an exception to be generated that the application might catch and handle. If the application doesn't handle it, an application crash occurs, indicating a possible denial-of-service vulnerability. However, if the section size field is set to be equal to 0 minus the size of the section header, the `HeapAlloc` call allocates a 0-byte length buffer due to the integer arithmetic overflowing and wrapping around 0. The subsequent call to `ReadFile` below it attempts to write a large amount of data to the 0-length heap block, causing a heap overflow. An attacker may exploit this vulnerability to achieve arbitrary code execution.

An application's file format handling should be tested against improper and malformed files. The test methodology should generate a series of malformed files by mutating properly formatted files, generating random garbage files, and creating files likely to trigger errors handling boundary conditions. The application should be tested against each file to ensure that it properly handles each one without crashing or causing unexpected behavior. The next section describes automated file corruption testing and some freely available file format testing tools.

File Corruption Testing

File corruption testing is a form of input fuzzing targeted at applications and interfaces operating on binary input files. Common applications of file corruption testing include testing image, font, and archive file format parsing.

Testing an application's handling of a binary file format may be performed at several different levels. A straightforward yet labor-intensive approach is to manually create a series of files that have been corrupted in different ways and proceed to attempt to use the file in the application being tested. This approach requires little or no programming; the file corruption can be performed manually with a hex editor or by a small Python script. With this approach, however, several issues arise. For example, there may be a large number of test cases, and manually corrupting the files and testing them may take too long—not to mention being mind-numbingly boring.

Some level of automation can speed up this process of file creation and testing to free the application penetration tester to do other, more interesting things.

Automated File Corruption

Binary file formats can be complicated, involving a large number of structures with type, option, size, and offset fields that may have intricate interdependencies. They may also contain file sections possibly involving compression or encryption. Manually creating test cases requires in-depth knowledge of the file format. It also may require "borrowing" a good deal of code from the application to be tested to properly compress or pack data into the file format. Although this sort of code reuse may save time, it may make some bugs difficult to find because the same incorrect assumptions made in file parsing would be assumed in the file creation. Luckily, by deliberately avoiding intimate knowledge of the file format, we can sidestep this pitfall and create a generic binary file corruption test harness that can uncover a good number of vulnerabilities quickly.

A simple tool to perform quick file format testing is Ilja van Sprundel's Mangle.[4] Mangle overwrites random bytes in a binary file format's header with random values, slightly biased toward large and negative numbers. Mangle requires a template file to mangle and the size in bytes of the file header. As an example, let's mangle a JPG file.

First, you need a sample JPG file. This example starts with a simple JPG file, Test.jpg. First, you must copy the template file to a new file to be mangled, Test0.jpg. Then you run Mangle on Test0.jpg to corrupt some bytes in the header. You give Mangle two command-line arguments: the name of the file to mangle, and the number of bytes in the file header. Mangle mangles only bytes in what it thinks is the file header. You specify 256 so that Mangle corrupts only some of the first 256 bytes in the file. Last, you use the MacOS X command-line `open` command to open the JPG in Preview.app. As shown in Figure 11-9, Preview.app is unable to open the mangled file.

```
% cp Test.jpg Test0.jpg
(Copy the template file to a new file)
% ./mangle Test0.jpg 256
(Mangle the new file)
% open Test0.jpg
(Try opening the mangled file)
```

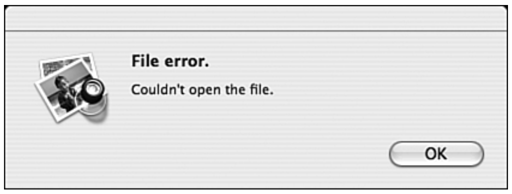

File error.
Couldn't open the file.

Figure 11-9 Error dialog from Preview.app

In a real-world security testing scenario, this procedure would be automated to generate a large number of test cases and minimize human interaction. Such testing frameworks are often application-specific and are beyond the scope of this book. Some tools are available to assist in automating file fuzzing and launching, such as FileFuzz[5] for Windows and SPIKEFile[6] for UNIX.

Command-Line Utility Fuzzing

By performing command-line fuzzing, we focus on an area where there are most likely to be exploitable vulnerabilities: command-line arguments and environment variables on UNIX-like operating systems. These vulnerabilities are typically used to attack local set-user-id root executables on UNIX operating systems. set-user-id UNIX executables were described earlier in this chapter.

Earlier in this chapter, we described how to identify the command-line arguments and environment variables an application uses. You will now use this knowledge to test how applications handle invalid input in them.

Immunity ShareFuzz

ShareFuzz, written by Dave Aitel at Immunity, Inc., was one of the first released local fuzzers. ShareFuzz uses shared library interposition to replace the definition of `getenv` (the UNIX function to retrieve the value of an environment variable) in the target process with its own implementation. This is done by creating a `shim` shared library and specifying it in the LD_PRELOAD environment variable to cause it to be loaded in the target process. The ShareFuzz getenv returns a long string of A characters for each variable, except for DISPLAY so as not to interfere with the operation of X11-based applications.

One of the primary benefits of ShareFuzz is that it does not require any knowledge of which environment variables an application uses because it automatically detects when the application queries the environment for a variable and returns false data. Another benefit of ShareFuzz is that it uses a shell script to automate the process of locating set-user-id root executables and testing each of them in succession. This can be used to quickly locate easily exploitable vulnerabilities on a commercial UNIX operating system.

First, you copy all the set-user-id root executables into a local directory:

```
% ./pullfiles.sh
Pulling all files into suid directory
find: cannot read dir /lost+found: Permission denied
find: cannot read dir /export/lost+found: Permission denied
...
```

Then you build the sharefuzz shared library:

```
% make
gcc -c -fPIC localeshared.c
echo linking
linking
ld   -G -z text -o libd.so.1 localeshared.o
```

Now you can use it to fuzz executables in *suid/*:

```
% env LD_PRELOAD=./libd.so.1 suid/dtappgather
shared library loader working
...
GETENV: DTUSERSESSION
...
MakeDirectory: /var/dt/appconfig/appmanager/AAAAAAAAA... : File name
too long
```

From this sample ShareFuzz session, you can see that dtappgather uses a particular environment variable to construct the name of a directory it attempts to create. It also shows that the application identified the pathname as being too long. This means that the application can properly handle long strings in this environment variable. This gives you some information on how the application uses the environment variable. You may begin testing how it treats file path elements such as directory separators and parent directory paths (/ and ..).

Brute-Force Binary Tester

Mike Heffner's Brute Force Binary Tester[7] is a fast local fuzzer meant to quickly test a large number of executables for low-hanging vulnerabilities. It tests each executable with a long string command-line argument, multiple long argument strings, a long environment variable string, and a long string as input. Executed programs are run in parallel, making fuzzing a large number of binaries fairly quick. However, because the options tested are hard-coded, this tool may not uncover deeper vulnerabilities in code that can be reached only with valid command-line options.

Listing 11-5
Example `bfbtester` *Session on* `/usr/lib/sa/sadc`

```
% ./bfbtester -a /usr/lib/sa/sadc
=> /usr/lib/sa/sadc
 (setuid: 0)
   * Single argument testing
   * Multiple arguments testing
   * Environment variable testing
Cleaning up...might take a few seconds
```

Listing 11-5 ran successfully without finding any vulnerabilities. If it had, the output would have included *** Crash *** and a description of the input that caused the crash.

CLI Fuzz

CLI Fuzz[8] is a custom local fuzzer built to address some of the limitations of other available tools. For example, other tools do not support "surgical fuzzing," or the inserting of fuzz strings into other fixed input. CLI Fuzz focuses on thoroughly testing an executable's command-line interface, but it requires the user to provide the knowledge of what arguments and variables the executable uses. It also has a number of features for testing an executable when the variables or commands it expects are unknown (see Listing 11-6).

Listing 11-6
CLI Fuzz Usage Summary

```
Command Line Fuzzer
Dino Dai Zovi <ddz@theta44.org>, 20050709
usage: ./clifuzz [ -0 ¦ -i <file> ] [var=val ...] [exec [arg ...]]
options:
 -0   Fuzz argv[0]
 -e   Fuzz currently defined environment variables as well
 -i   Pipe <file> as standard input to target executable
 -o   Show child standard output
 -O   Show child standard error output
```

To support controlled surgical fuzz input generation, a concise language was devised to specify generated input test cases. This language is modeled after the format string language used in the `printf` family of functions in the C programming language standard library. It also takes some syntactical elements from POSIX regular expressions.

The fuzz rule is a character string composed of plain characters, escape sequences, and fuzz generator specifications. Plain characters are inserted as-is into the output. Escape sequences are used to specify special characters such as nonprintable characters or binary data. The escape sequences are identical to those used in the C programming language. They are presented in Table 11-1.

Table 11-1

Fuzz Rule Escape Sequences

Escape Sequence	Description
`\e`	Writes an escape character
`\a`	Writes a bell character
`\b`	Writes a backspace character
`\f`	Writes a form-feed character
`\n`	Outputs a newline character
`\r`	Outputs a carriage return character
`\t`	Outputs a tab character
`\v`	Outputs a vertical tab character
`\'`	Outputs a single-quote character
`\\`	Outputs a backslash character
`\num`	Outputs the byte whose ASCII value is given in the three-digit octal number *num*

`\xhex`	Outputs the byte whose ASCII value is given in the two-digit hexa decimal number *hex*

Fuzz generators are specified with a `%` (percent) character followed by a character indicating which generator to use. These characters and their meanings are defined in Table 11-2. An optional decimal number between the percent and the fuzz generator specification indicates a *repeat count.* An optional literal sequence between curly braces ({ and }) defines a *default value* that is used when that fuzz generator is inactive. If a default value is not specified, a somewhat reasonable default is used. Finally, parentheses may be used to specify *blocks*. *Block length references* can be used to output the length of those groups in binary or as an ASCII decimal or hexadecimal string.

Table 11-2

Fuzz Generator Specifiers

Character	Meaning
`A`	ASCII "fuzz" string
`C`	ASCII character
`D`	Digit (0 to 9)
`N`	32-bit number as a string
`X`	32-bit number as a hex string
`L`	Binary LONG (4 bytes)
`S`	Binary SHORT (2 bytes)
`B`	Binary BYTE

The fuzz rule language is perhaps best explained by some examples of using it to construct input test cases for some well-known syntaxes. Let's look at a common ASCII-based protocol, HTTP. Here is the most basic HTTP 1.1 `GET` request to retrieve the file index.html from a Web server:

```
GET /index.html HTTP/1.0<carriage return><newline>
Host: www.example.com<carriage return><newline>
```

You now go through each token and specify a fuzz generator to test it. You will replace literal tokens and variables with ASCII fuzz string generators (`%A`) and replace the digits in `HTTP/1.1` with ASCII digit generators. You will also use escape sequences to specify the unprintable carriage return and newline characters. When you do this, you get this fuzz rule:

```
%A{GET} %C{/}%A{index}.%A{html} %A{HTTP}/%D{1}.%D{1}\r\n
%A{Host}: %A{localhost}\r\n\r\n
```

A second example demonstrates using our rule language to generate binary input test cases. Consider a simple binary file format consisting of a magic number to identify files of its type, a file length, a file offset where data begins, and finally the file's contents. The magic number, length, and offset are all 32-bit LONG values. Our fictional file format is specified in Table 11-3.

Table 11-3

Simple Binary File Format

File Offset (Hex)	**Field**	**Description**
00	Magic	Magic number; must be 0xdeadbeef
04	Length	Length (in bytes) of data in data portion
08	Offset	Offset in file of data portion
Offset	Data	Data, of length *length* beginning at file offset *offset*

The following is a fuzz rule specifying an input file containing 16 data bytes, starting at file offset 12 (right after the header):

```
%L{\xde\xad\xbe\xef}%L{$L1}%L{\x00\x00\x00\x0c}(%16B{A})
```

Notice how parentheses are used to group the 16-byte data portion as a block and output the length of that block as a binary LONG value earlier in the string.

As an example of its usage, consider the following simple invocation of CLI Fuzz:

```
% clifuzz -0e SOMEVAR=%A vulnprogram –%A{h} %A
Results:
Critical: 2
High:      1
Medium:    0
Low:     780
Normal:    0
```

This example tests a fictional program, vulnprogram, fuzzing `argv[0]` (usually the name of the program being executed), each currently defined environment variable, an additional environment variable `SOMEVAR`, and command-line argument parsing.

CLI Fuzz evaluates the exit status of the target program for each execution and records them by severity, optionally logging or printing each result above a certain severity. The severities are determined by the child process's exit status. As shown in Table 11-4, they range from normal severity, indicating a clean exit status, to critical, indicating that the child process terminated with a bus error.

Table 11-4

CLI Fuzz Run Status Severities

Severity	**Description**
Normal	The child exited cleanly with status 0 (success).
Low	The child exited with a nonzero status.
Medium	The child terminated with a signal (besides SIGSEGV or SIGBUS).
High	The child terminated with signal SIGSEGV (segmentation fault).
Critical	The child terminated with signal SIGBUS (bus error).

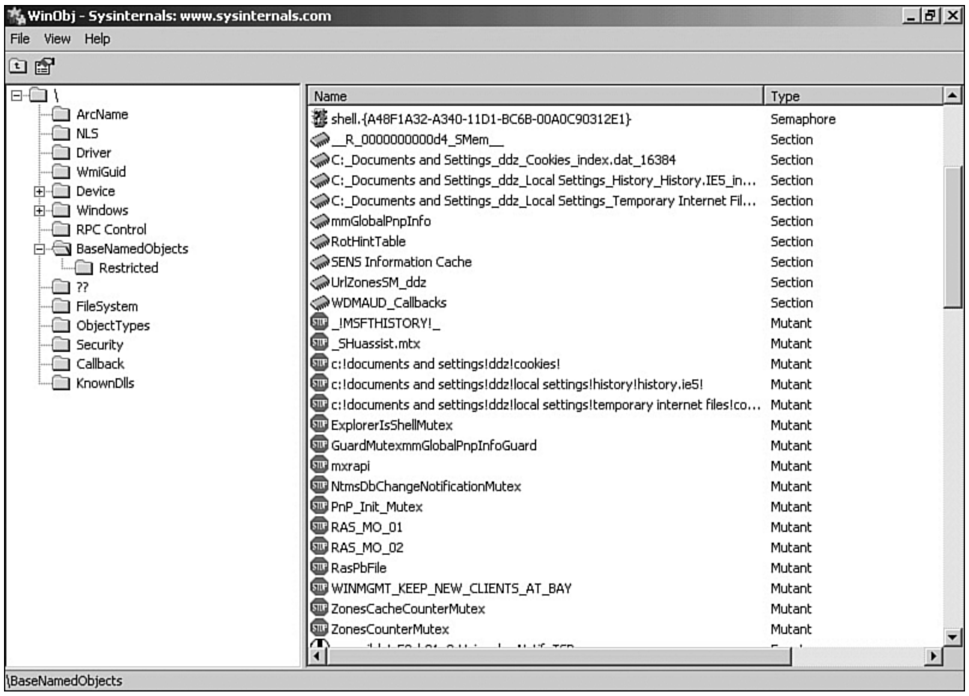
At the end of the session, CLI Fuzz prints out a count of the number of executions that resulted in each severity.

Shared Memory

Shared memory is a form of interprocess communication supported on almost all operating systems, including Windows and UNIX-like operating systems. Shared memory facilities allow a range of memory addresses, the *shared memory segment*, in one process to be visible to another. For one process to make data available in another process on the same machine, all it has to do is write the data to the shared memory segment. Any other process that has the shared memory segment mapped will be able to use it immediately.

Under Windows NT-based operating systems, shared memory segments are generally called *shared sections*. Shared sections are one type of Windows kernel executive object (described earlier). You can identify them and their permissions using WinObj (see Figures 11-10 and 11-11). Shared sections are listed under the \BaseNamedObjects\ directory and can be grouped by sorting on the Type column in WinObj.

Some tools to test shared sections were released by Cesar Cerrudo of Argeniss Information Security at Black Hat Europe 2005.[9] His command-line tools include ListSS to list local shared sections (see Figure 11-12), DumpSS to output the contents of the shared section memory, and TestSS to overwrite the memory in a shared section. Many applications use shared sections for interprocess communication. If the application being tested uses a shared section, overwriting the section with DumpSS may cause the application to crash. Data



WinObj - Sysinternals: www.sysinternals.com

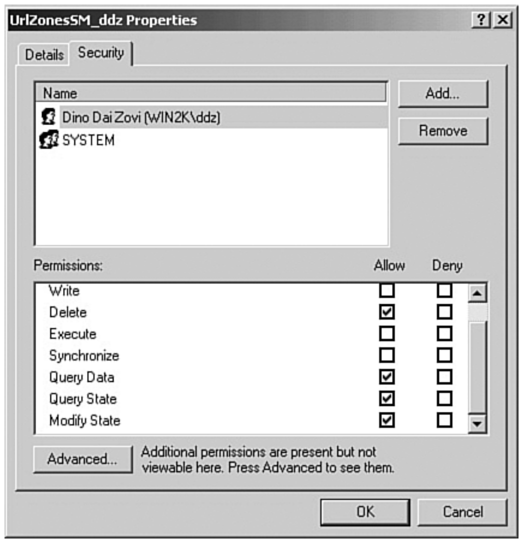
File View Help

```
\
  ArcName
  NLS
  Driver
  WmiGuid
  Device
  Windows
  RPC Control
  BaseNamedObjects
    Restricted
  ??
  FileSystem
  ObjectTypes
  Security
  Callback
  KnownDlls
```

Name	Type
shell.{A48F1A32-A340-11D1-BC6B-00A0C90312E1}	Semaphore
__R_0000000000d4_SMem__	Section
C:_Documents and Settings_ddz_Cookies_index.dat_16384	Section
C:_Documents and Settings_ddz_Local Settings_History_History.IE5_in...	Section
C:_Documents and Settings_ddz_Local Settings_Temporary Internet Fil...	Section
mmGlobalPnpInfo	Section
RotHintTable	Section
SENS Information Cache	Section
UrlZonesSM_ddz	Section
WDMAUD_Callbacks	Section
!MSFTHISTORY!	Mutant
_SHuassist.mtx	Mutant
c:!documents and settings!ddz!cookies!	Mutant
c:!documents and settings!ddz!local settings!history!history.ie5!	Mutant
c:!documents and settings!ddz!local settings!temporary internet files!co...	Mutant
ExplorerIsShellMutex	Mutant
GuardMutexmmGlobalPnpInfoGuard	Mutant
mxrapi	Mutant
NtmsDbChangeNotificationMutex	Mutant
PnP_Init_Mutex	Mutant
RAS_MO_01	Mutant
RAS_MO_02	Mutant
RasPbFile	Mutant
WINMGMT_KEEP_NEW_CLIENTS_AT_BAY	Mutant
ZonesCacheCounterMutex	Mutant
ZonesCounterMutex	Mutant

\BaseNamedObjects

Figure 11-10 WinObj listing active sections



UrlZonesSM_ddz Properties

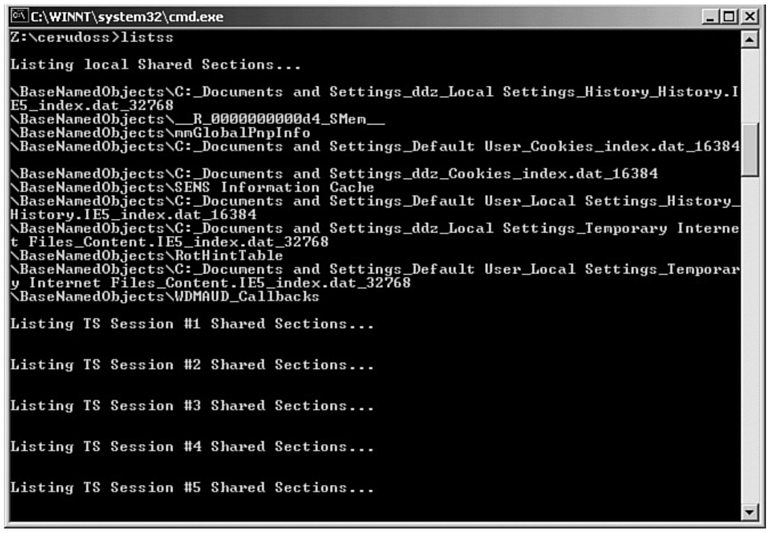
Details | Security

Name:
- Dino Dai Zovi (WIN2K\ddz)
- SYSTEM

Permissions:	Allow	Deny
Write	☐	☐
Delete	☑	☐
Execute	☐	☐
Synchronize	☐	☐
Query Data	☑	☐
Query State	☑	☐
Modify State	☑	☐

Additional permissions are present but not viewable here. Press Advanced to see them.

Figure 11-11 WinObj section ACL



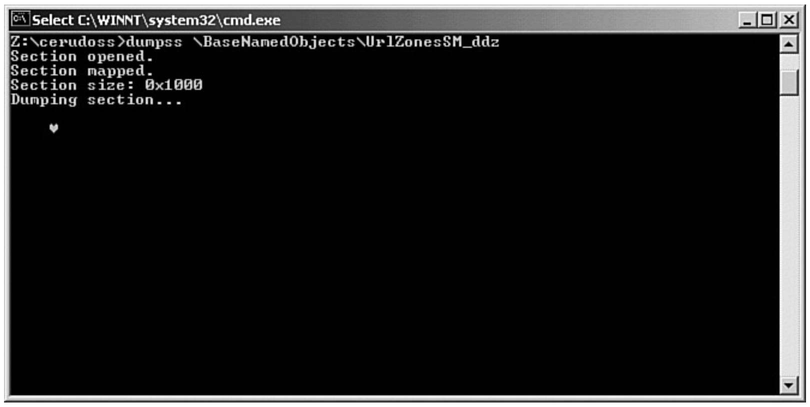
```
C:\WINNT\system32\cmd.exe
Z:\cerudoss>listss

Listing local Shared Sections...

\BaseNamedObjects\C:_Documents and Settings_ddz_Local Settings_History_History.I
E5_index.dat_32768
\BaseNamedObjects\__R_0000000000d4_SMem__
\BaseNamedObjects\mmGlobalPnpInfo
\BaseNamedObjects\C:_Documents and Settings_Default User_Cookies_index.dat_16384

\BaseNamedObjects\C:_Documents and Settings_ddz_Cookies_index.dat_16384
\BaseNamedObjects\SENS Information Cache
\BaseNamedObjects\C:_Documents and Settings_Default User_Local Settings_History_
History.IE5_index.dat_16384
\BaseNamedObjects\C:_Documents and Settings_ddz_Local Settings_Temporary Interne
t Files_Content.IE5_index.dat_32768
\BaseNamedObjects\RotHintTable
\BaseNamedObjects\C:_Documents and Settings_Default User_Local Settings_Temporar
y Internet Files_Content.IE5_index.dat_32768
\BaseNamedObjects\WDMAUD_Callbacks

Listing TS Session #1 Shared Sections...


Listing TS Session #2 Shared Sections...


Listing TS Session #3 Shared Sections...


Listing TS Session #4 Shared Sections...


Listing TS Session #5 Shared Sections...
```

Figure 11-12 ListSS listing shared sections

stored in shared sections may often include pointers and other complex data structures. If an attacker can overwrite this data, he might be able to take control of the application.

Figure 11-12 shows that Windows 2000 has a number of shared sections used internally by applications such as Internet Explorer. Any shared sections used by third-party applications also appear in this listing.

As an example of testing a shared section, examine the section \BaseNamedObjects\UrlZonesSM_ddz. First, dump the contents of the shared section with DumpSS, as

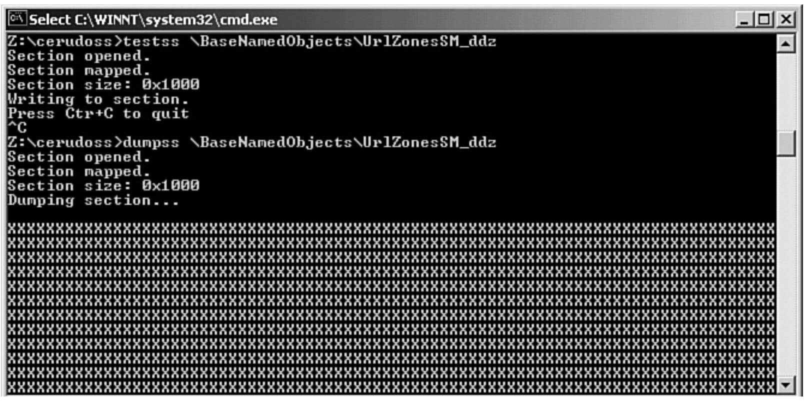


```
Select C:\WINNT\system32\cmd.exe
Z:\cerudoss>dumpss \BaseNamedObjects\UrlZonesSM_ddz
Section opened.
Section mapped.
Section size: 0x1000
Dumping section...

     ♥
```

Figure 11-13 DumpSS dumps contents of shared section memory

shown in Figure 11-13. No ASCII data is visible in the section, so if you need further information on the section contents, you should examine the output of DumpSS as a hexadecimal byte dump.

Proceed by using TestSS to overwrite the entire shared section. As shown in Figure 11-14, after you run TestSS, the contents of the shared section are replaced



```
Z:\cerudoss>testss \BaseNamedObjects\UrlZonesSM_ddz
Section opened.
Section mapped.
Section size: 0x1000
Writing to section.
Press Ctr+C to quit
^C
Z:\cerudoss>dumpss \BaseNamedObjects\UrlZonesSM_ddz
Section opened.
Section mapped.
Section size: 0x1000
Dumping section...

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Figure 11-14 TestSS overwrites the shared section, and DumpSS shows that it has changed

with the ASCII X character (hexadecimal byte 0x58). If any application uses this corrupted shared section, it may crash. If something crashes, you should examine the crash and determine exploitability, as described in the next chapter.

Summary

This chapter discussed how to test local applications with fault injection, including methodologies for testing ActiveX objects, file formats, command-line executables, and shared memory segments. This chapter also described useful tools to help with and automate this process.

The testing performed in this and previous chapters may have uncovered a number of vulnerabilities. The next chapter discusses how to evaluate a vulnerability's exploitability and construct a proof-of-concept exploit to reproduce it.

Endnotes

1. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0938.
2. "Introduction to ActiveX Controls," http://msdn.microsoft.com/workshop/components/activex/intro.asp.

3. "How Internet Explorer Determines if ActiveX Controls Are Safe," http://support.microsoft.com/kb/q216434/.
4. http://www.digitaldwarf.be/products/mangle.c.
5. http://labs.idefense.com/labs-software.php?show=3.
6. http://labs.idefense.com/labs-software.php?show=14.
7. http://bfbtester.sourceforge.net.
8. CLI Fuzz is available for free download on this book's Web site, http://www.softwaresecuritytesting.com.
9. "Hacking Windows Internals," http://blackhat.com/presentations/bh-europe-05/BH_EU_05-Cerrudo/BH_EU_05_Cerrudo.pdf.