Black Box DEBUGGING

Modern software development practices build applications as a collection of collaborating components. Unlike older practices that linked compiled components into a single monolithic application, modern executables are made up of any number of executable components that exist as separate binary files.

This design means that as an application component needs resources from another component, calls are made to transfer control or data from one component to another. Thus, we can observe externally visible application behaviors by watching the activity that occurs across the boundaries of the application's constituent components.

In addition to its own components, an application can import third-party components such as COM (component object model) or CORBA (common object request broker architecture). The application also has a fundamental dependence on other core components such as the operating system kernel and file system

**It's all about what takes
place at the boundary of
an application.**

JAMES A. WHITTAKER, FLORIDA INSTITUTE OF TECHNOLOGY
HERBERT H. THOMPSON, SECURITY INNOVATION

Black Box Debugging

and communicates with them almost continuously during execution. Each of these interactions can be observed without reference to the application's source code or compiler symbols.

Debugging is most common when source code or compiler symbols are available. In such situations, debuggers can be attached to an application and runtime information collected. The source and symbols allow information, like the values of internal variables, to be tracked so that engineers can discover some aspect of internal behavior. As is typical in commercial or legacy software, however, either the source or symbols may be unavailable for such purposes.

In some cases, the information that crosses the boundaries of application components can be used for testing and debugging, among other things. Our interest here is in diagnosing bugs that may involve the interactions between the core application and a component or service with which it interfaces. We also want to study how information passes through an application's boundary and then be able to manipulate this data to force the application down certain execution paths.

Software developers, testers, and users are not always aware of the complex relationships that are created when functionality comes from an external component. From a software quality and security point of view, an application inherits the problems of the external components on which it relies. If a bug exists in either the external component or the documentation describing its programmatic interface, an enormous amount of time can be wasted trying to isolate a bug. Whenever a failure manifests in an application, we automatically think that the bug is in that application's code. Usually this leads to a painstaking and time-consuming line-by-line inspection of the code to track down the offending commands. In reality, software is often buggy, which means that its components are frequently riddled with errors.

Another issue is that application developers often expect a discrete set of values to be returned from an external function call. In many cases, developers assume that these calls will succeed and fail to check the return values of calls to the functions. Observing the data passed to these third-party components and the return values of function calls can dramatically reduce the time needed to track down application bugs originating from an external component.

This is where black box debugging comes in. Black box debugging collects useful debug information by monitoring only those behaviors external to an application. By observing the data that passes between the application

and its dependencies, developers can better understand relationships and both diagnose and precipitate a wide range of failures.

Consider the following example:

Microsoft Notepad is a text-editing program that ships with many versions of Windows. If we launch Notepad in Windows XP, its appearance matches the Windows XP default theme (shown as the upper left window in figure

FIG 1

Microsoft Notepad Fails to Load a Library

FIG 2

Diagnosing the Notepad Failure

If Notepad fails to load the `uxtheme.dll` library in Windows XP, it reverts to the window style of previous versions of the operating system (bottom right). What causes such unusual behavior? In figure 2, Security Innovation's Holodeck tool shows that, in the second instance, the `uxtheme.dll` library failed to load. This library is responsible for applying the style or theme to a window, so when it failed to load, it forced the application to display in a default state. In practice, this difference in behavior could have been caused by a number of environmental inputs, the most likely being a lack of memory that resulted in a failure of the Windows `LoadLibrary` system call.

1). With the same user-input sequence, namely double-clicking on the Notepad executable, the application sometimes appears as shown in the lower right of figure 1. Note the differences between the two windows. These were produced using the same user-input sequence on the same machine, but obviously a portion of input the application received from its environment was different. Black box debugging can be used to help track down the offending input (See figure 2).

In the following section we will identify black box behaviors and discuss how they can be intercepted. Then, using examples of real software, we will show how black box debugging can be used to diagnose software defects.

BLACK-BOX BEHAVIOR

We all accept the premise that software accepts inputs through any number of interfaces and, through internal computation and data manipulation, produces outputs that are then sent to final destinations through one of those interfaces. For example, a user may send an input through a keyboard interface, which causes an application to send data through a network interface—say, to fetch a Web page. Many such inputs and outputs make up an application's black box behavior.

Black box behavior exists at the interfaces of an application without taking internal operations into account. When collecting black box information, we are concerned only with inputs that enter through an interface and outputs that are generated as a result of those inputs. Exactly how the inputs get transformed into outputs is the domain of traditional debugging procedures.

This concentration on external behavior makes it important to identify all possible interfaces to a given application so that the behavior crossing those interfaces can be collected and presented as debug information.

We begin by defining general classes of software interfaces. It is important to understand these interfaces, as they will determine the information that we can collect for black box debugging.

Imagine an application's behavior while it is executing. At the boundaries of the application, six major categories of behavior occur:

- 1. I/O devices** including the keyboard, mouse, and monitor are the most recognizable interfaces.
- 2. The operating system kernel** supplies input to the application by allocating memory for the application to store data. The application received these inputs by making calls to the kernel API (application programming interface). Since these calls leave the application destined for the kernel, they are outputs.

3. Process operations are a special type of service provided by the operating system that allow one program to spawn another program. The call to create the process is now part of the black box behavior of the original program, so we can either ignore the spawned program or decide to include its behavior as part of the larger black box behavior of the combined processes.

4. External libraries are accessed via their published APIs in a similar manner to kernel accesses.

5. File operations occur when a program creates, reads, writes, or modifies data on storage devices. These files exist outside the black-box boundaries of the program and are identifiable as API calls to the file system and return values/error codes that are passed back to the program from the file system.

6. Network activity occurs through external libraries that control sockets and ports on the computer where the program resides.

All of these categories of behaviors are identifiable at the application's boundaries—that is, outside its black box. This is the information that is available for black-box debugging.

We will use this information to learn about aspects of program behavior when the source code is unavailable or not readily usable (e.g., if we cannot control the build environment). It can be used for interpreting behavior, finding bugs and analyzing their root causes—and many things related to either the behavior or implementation of a compiled binary.

One area in which this can help is diagnosing bugs that result from failures in an application's environment. Failures that occur in one environment and cannot be reproduced in another are the bane of software testers. The typical process for describing a failure is to document a sequence of user actions that cause the failure to manifest, and then to pass this description to an application developer who then reproduces the failure on another machine to track down the source of the problem. If the failure is not reproducible, the chances of the bug being fixed are small.

Why does the same software that fails on one machine work correctly on another with the identical sequence of clicks and keystrokes? We all know that software is deterministic—given the same inputs we expect the same result; but the issue is that we rarely take into account the nonuser-generated inputs that an application receives through its interfaces.

Figure 3 shows how we typically “perceive” an application responding to input. This is a fairly accurate representation of what occurred in the earlier Notepad

Black Box DEBUGGING

example. We see the same set of user inputs delivered to the identical application running on two different machines. When this input sequence is applied to machine 1, the software fails. When applied to machine 2, the application responds correctly. At first blush, the different behaviors, given the same input sequence, seem to violate the deterministic nature of software.

Figure 3 shows another set of inputs to the application that we initially failed to consider: the “actual” interaction. On machine 1 there was a failure in loading a library, which ultimately resulted in the application’s failure. To truly understand software behavior requires knowing its inputs and outputs through all of its interfaces; this is where black box debugging can help.

In order to observe these interactions and interpret the data that moves through the application’s boundary, we need specialized tools. A number of observation tools are available both commercially and as freeware. A freeware suite of tools suitable for this purpose is available from Sysinternals,¹ which produces Regmon, Filemon, and ListDLLs to monitor registry, file-system, and library interactions, respectively. Two commercial offerings in this space are Appsignt by Identify Software² and Holodeck

from Security Innovation.³

FAULT INJECTION

Monitoring data that passes through the application’s interfaces can help diagnose a failure once it has occurred. This approach, however, is *reactive*; we first wait for a failure to occur and then collect debug information to isolate the source of the failure. From a black box perspective we have far more control over the application than user input. With the proper tools, we can control interactions through all of the application’s interfaces. By manipulating inputs from the six sources identified earlier, we can force the application down specific execution paths. Among the possible inputs that can be fed to the application through its interfaces, perhaps the most interesting are those that simulate failure conditions in the application’s environment. Failures can take the form of a failed library load, insufficient memory, write-protection errors on disk, and more. This process of simulating environmental failures is referred to as *runtime fault injection*.

Error conditions are interesting because when extraordinary conditions occur as a result of stress, any error-

handling routines are executed. These are pathways through the application that do not add to its functionality but are designed to keep the functional code from failing. These error-handling routines are notoriously subjected to far less testing than the functional code they are created to protect. With such limited exposure to testing, these code paths are fertile breeding grounds for many types of defects. It is therefore important to force environmental failures during debugging. This is where runtime fault injection can help.

Another area where runtime fault injection can help is with nonreproducing bugs. These failures are often caused by unexpected input through one of the application's hidden interfaces. This value could be the result of contention for a resource with another process running on the machine or possibly an intermittent failure in memory or the disk. These types of anomalies, even if recorded using black box debugging, are difficult to reproduce manually.

To demonstrate, consider the "Save As" dialog box for the Notepad text-editing program that ships with Windows XP (see figure 4a). To get to this dialog box, the following sequence of inputs can be applied:

1. Launch Notepad
2. Click the Save As option from the file menu.
3. Observe the failure: The Save As dialog box is missing the identifiers Save and Cancel on the appropriate buttons, as depicted in figure 4a. When a valid file name is

entered and the Save button is clicked, the dialog box closes normally, without saving the document.

This reproduction sequence is brief and accurately represents the sequence of clicks and keystrokes that caused Notepad to fail. If you follow these steps on your Windows XP machine, however, Notepad will likely respond correctly (see figure 4b.). Why? The reason is that the reproduction steps did not consider the hidden environmental inputs that Notepad receives. The failure manifests only when system memory is low, which results in errors being returned to Notepad's system calls to the operating system kernel to allocate space in memory. Bugs like these can cause hours of frustration for testers who try desperately to track down the keystroke or click they may have missed to reproduce a behavior. Black box debugging techniques would drastically reduce the time required to isolate the true failure-producing inputs.

In addition to identifying these inputs, it is useful to have a convenient way of supplying environmental inputs to an application in a black box manner. For example, consider the challenge of reproducing the Notepad failure. How could you do it? One option would be to start many background processes and create contention for memory. Another is to write a program that allocates memory until all of the memory on the system is used.

These approaches have a few problems. The first is that the failure in the memory allocation call by Notepad may or may not happen because of the almost random

Black Box DEBUGGING

nature by which memory is being consumed. The second problem is that any analysis tools you may want to use on Notepad will probably not function well, or at all, under tight memory constraints. A third problem is that consuming all of the memory on a system may also cause critical operating system functions to fail and thus hang or crash the system.

What we really need are tools to control inputs to the application's hidden interfaces that will affect only the application under test. There is hope in this area. White box fault-injection approaches (those in which the internal structure and coding are known) have been used in the industry to simulate environmental error conditions by hard-coding return values of system calls. There have also been attempts on the Unix platform to inject environmental failures in a black box fashion at runtime.⁴ We have previously demonstrated how this could be accomplished on the Windows platform.^{5, 6, 7} Security Innovation's Holodeck tool can be used to control all data that passes through an application's boundary at runtime without modifying application code. Such tools bring the power to force the application down specific execution paths and correctly identify inputs to software that produce a failure.

Challenges remain. How do we decide when to inject faults? Which faults are meaningful? Which faults are likely to be bug-revealing? Which failures are more important—and what are the risks of failure, such as security? These are all important questions that practitioners face when applying these techniques during the testing and debugging process. The software engineering and testing community is now only beginning to answer them, but the general rule of thumb is to induce stress when an application is most in need of a resource. Limiting memory during intense computation or simulating network faults during remote authentication are both examples of this targeted insertion. As more tools begin to surface, these techniques will likely find a welcome home in most testing arsenals.

Software developers, testers, and users have access to and control over lots of data that is exchanged between an application and its environment. This data is accessible and interpretable without the benefit of source code or symbols. Using the tools and techniques discussed, users can unlock valuable debug information from compiled binaries by looking at all behaviors, not just those visible through the user interface.□

REFERENCES

1. Sysinternals: see <http://www.sysinternals.com/>.
2. Identify Software: see <http://www.identify.com>.
3. Security Innovation: see <http://www.sisecure.com>.
4. Kao, W. I., Iyer, R. K., and Tang, D. FINE: A fault injection and monitoring environment for tracing the Unix system behavior under faults. *IEEE Transactions on Computer Science* 19, 11 (Nov. 1993), 1,105–1,118.
5. Thompson, H. Why security testing is hard, *IEEE Security and Privacy* (July/Aug. 2003), 83–86.
6. Thompson, H., Whittaker, J. A., and Mottay, F. Software security vulnerability testing in hostile environments, *Proceedings of ACM SAC* (2002), 260–264.
7. Thompson, H., and Whittaker, J. A. Testing for Software Security, *Dr. Dobb's Journal* (Nov. 2002), 24–34.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

JAMES A. WHITTAKER is a professor of computer science at the Florida Institute of Technology. He is also a member of Microsoft's Trusted Computing Academic Advisory Board and is editor of the "Application Security" column for *IEEE Security and Privacy* magazine. His research interests are software testing and reliability, software design methods, and computer security. He is the author of *How to Break Software* (Pearson Addison Wesley, 2002) and coauthor with Herbert H. Thompson of *How to Break Software Security* (Pearson Addison Wesley, 2003). Whittaker has a Ph.D. in computer science from the University of Tennessee and is a member of ACM and IEEE.

HERBERT H. THOMPSON is director of security technology at Security Innovation. Thompson has worked extensively with James A. Whittaker in researching software security and anti-cyber warfare. He has also worked as a software test engineer for Microsoft and is a frequent speaker and writer on software security. He has spoken at IEEE's Software Reliability Conference (ISSRE2001), numerous software testing conferences, ACM SAC and ACSAC, among others, and was the general track chair for software engineering at the ACM SAC 2003 conference. He has published articles in academic journals and trade magazines including *Software Test and Quality Engineering* and *Dr. Dobb's Journal*. He is a certified information systems security professional (CISSP) and is pursuing his actuarial license. Thompson worked with Whittaker on *How to Break Software Security* (Pearson Addison Wesley, 2003). He holds a Ph.D. in mathematics from the Florida Institute of Technology.

© 2003 ACM 1542-7730/03/1200 \$5.00