feature

defect detection

# Software's Invisible Users

**James A. Whittaker,** *Florida Institute of Technology*

**Software systems receive input from several kinds of users, only one of which is human. The author discusses other kinds of users, often misunderstood and sometimes forgotten, and the problems that ignoring them can lead to.**

Software is deterministic. Given a starting state and a fixed series of inputs, software will produce the exact same output every single time those inputs are applied.[1] In fact, many of the technologies we apply during development (reviews, formal proofs, testing, and so forth) rely on this determinism—they would not work otherwise.

But if software is deterministic, why do weird things happen? Why is it that we can apply a sequence of inputs and observe a failure and then be unable to reproduce that failure? Why does software work on one machine and fail on another? How is it that you can return from lunch and find your Web browser has crashed when it wasn't being used?

The answer is, of course, that modern software processes an enormous number of inputs and only a small percentage of these inputs originate from human users. When testers can't reproduce failures, the reason is that they are only resubmitting the human input without sufficient regard to the operating system return codes and input from runtime libraries. When software fails on a new computer after running successfully on another, the reason can only be that the new system is supplying input that differs in content from the old one. And the browser that crashes when you are at lunch is responding to input from a nonhuman external resource.

## Where errors slip in

Testers and developers routinely ignore these invisible users or, even worse, do not realize they exist. But it is the mishandling of their inputs that causes hard-to-diagnose periodic system failure and, worse, opportunities for hackers who are all too familiar with such weaknesses in software development practices.[2]

The inputs from the software environment—the operating system kernel (the part of the OS that supplies memory, file handles, heap space, and so on), runtime libraries, external application programming interfaces (APIs), and the file system—are much the same as inputs from a human user: there are lots of them, many are invalid or produce error codes, and each one should be validated before being processed by application software. The danger in not



0740-7459/01/$10.00 © 2001 IEEE

## Typical Reactions to Low-Memory Situations

We wrote a tool to simulate low-memory behavior and tested a number of standard industry applications. All the applications failed to gracefully handle unexpected return values from the kernel. Here are some of the results.

- Scenario 1. We loaded a legitimate page in a Web browser and then denied further attempts to allocate global and local memory. Then we reloaded the same page using the browser's reload button. The browser returned an "invalid syntax" error message. Diagnosis: the syntax is obviously fine because we reloaded a previously displayed page; however, the developer wrote a global exception handler and the only thing he or she could think of that would cause the exception to be raised was that the user typed an invalid URL.
- Scenario 2. We selected the Open File dialog from a desktop application's menu and pointed it to a directory with lots of files. The files were listed correctly in the dialog box. Then we blocked access to local and virtual memory and redisplayed the same dialog. The files were not listed (even though they still resided in the same directory). Diagnosis: not enough memory was available to correctly populate the Open File dialog's display area, but the application had no facility to handle this situation. However, because we did not block access to creating the dialog window, the function completed without listing any files.

In scenario 1, the developer realized that the function might fail but did not fully consider all the ways it could fail. Thus, we received an error message that did not describe the real problem and would be useless for diagnosing the failure. In scenario 2, the developer didn't even consider failure. The function completed as though it were successful, so the user was left wondering what might have happened to the files.

doing so is the same risk we take when we let human users enter unvalidated input: corrupted data, buffer overruns, and invalid computation results.[3]

To demonstrate input from invisible users, my colleagues at the Center for SE Research and I constructed a software tool that watches Windows programs while they run and identifies events that cross external interfaces. We then executed a number of applications and let the tool log all the activity across their various interfaces. The results are eye opening. For example, Microsoft PowerPoint, a complex and large application for making presentations and slide shows, makes 59 calls to 29 different functions of the Windows kernel (excluding GetTickCount, which was called nearly 700 times) upon invocation. That means a single input from a human user (invoking the application) caused a flurry of undercover communication to be transferred to and from the OS kernel.

Certainly, invocation is a special input and requires a great deal of setup and data initialization. But other operations are also demanding on low-level resources: when PowerPoint opens a file, it calls 12 kernel functions a grand total of 73 times (once again excluding GetTickCount, which was called more than 500 times); when PowerPoint changes a font, it calls two kernel functions a total of 10 times.

And these are only calls to the OS kernel. PowerPoint also uses a number of other external resources such as dynamically linked libraries in the same manner as the kernel. Clearly, the amount of communication between an application and its "invisible" users dwarfs visible human input.

Perhaps we could account for many of the mysterious and hard-to-reproduce system anomalies if we treated invisible users the same as we treat human users. If any of those calls produce unexpected return codes, the system in question must handle these events and react appropriately.

To test applications' capabilities of handling unexpected input, we perturbed some of these inputs so that the application in question received legitimate but unexpected return codes and data from its environment. For example, when an application allocated memory, we passed back an unsuccessful return code; when the application opened a file, we told it there were no more file handles; and so forth. Every application we tested this way failed within seconds (see the "Typical Reactions to Low-Memory Situations" sidebar). The diagnosis: many software applications are unaware of the diversity and complexity of their environment. We seem to expect that nothing will go wrong, networks won't get congested, memory will never run out, and virus writers are just pranksters who can cause no real harm. (See elsewhere for another case study detailing the failure of applications to handle unexpected system call return codes.[4])

How important is it to handle these situations? Skeptics will argue that inputs usually come in as expected, so that the cost of checking them is not money well spent (for more on checking inputs, see the "Validat-

## Validating User Input

The most straightforward way to validate user input is to follow the Get Input command with a selection statement (`if`, `case`, and so on) that checks validity before continuing. But you must then encapsulate the input routines in a separate module to avoid littering the body of the code with `ifs`. Remember too that all this checking will slow down your code.

If the input is to be immediately stored, it is prudent to hide the storage structure behind a firewall of access routines (that is, make it an object). The access routines are then responsible for not allowing bad data into the structure.

A popular but unsafe option is to simply raise an exception, trapping any errors. Beware of the side effects of exception handling, though. Programs tend to lose state information when exceptions occur because the instruction pointer has changed abruptly. When the instruction pointer is restored after the error routine executes, files could still be open and data might or might not be initialized, creating traps for unwary programmers.

Preventing the input from ever getting to the application in the first place is possible only at the human user interface. GUIs are specifically designed to do just that. Specialized controls that will only allow integers (or other data types or formats) to be typed are a great way to filter unwanted input without having to write error routines.

ing User Input" sidebar). However, hackers are not so forgiving of our software's weaknesses. Common attacks such as packet storms that overwhelm memory and stress resources are among a number of ways to cause denial of service by exploiting an application's lack of attention to its environment. Often, external events such as a specific time (which is a return value of a kernel call and, therefore, an input) trigger logic bombs. How can we possibly test for such things without treating memory and system-call return values as inputs?

This is indeed the bottom line for software developers and testers: you must consider every single input from every external resource to have confidence in your software's ability to gracefully and safely handle malicious attacks and unanticipated operating environments. Deciding which inputs to trust and which to validate is a constant balancing act.

Where do these inputs come from? How should developers and testers handle them? Figure 1 depicts a typical software application and its operating environment. The inner circle represents the application in question. Note that its boundaries are completely contained within the OS. Indeed, modern software communicates with nothing but the OS—all communication with any other external resource must go through it first.

Beyond the OS lie four classes of user (which is the term I will adopt for any external resource with which software can communicate). These are human users, OS users, API users, and file system users. Let's look at each of these users, the challenges of dealing with them, and the types of inputs they can apply.

### The human user

Human users often do not understand that their input does not actually touch the applications they use. But programmers must understand this or risk inheriting bugs from the programs that really do process human input.

Consider keyboard input, for example. First, a keyboard device driver interprets the keystrokes, and then the operating system passes that information through layers of OS libraries and APIs until it finally generates and then passes an event to the target application. The event is all the application sees; the many layers of OS libraries take care of interpreting the inputs from the human and translating them into something the application can deal with.

Can things go wrong with this process? Of course—software is involved, after all. Bugs in the OS or code for GUI controls can cause all types of problems for an application expecting only a successful event. For example, what happens when you put too many entries in a list box? The list box will fill up and then either crash, taking your application with it, or generate an error code that gets passed back to your application. If your application is not equipped to handle the error, it will crash—all on its own this time. It is imperative that programmers understand this process and learn which interface controls and APIs are trustworthy and exactly how and when they return error codes when they fail.

When developers do not handle GUI control error codes, they are risking denial-of-service attacks that exploit such holes by finding ways to force the interface error to occur. Even worse, developers might explicitly trust data passed from a GUI control; this opens the door for potential buffer overruns when the user supplies too much data. Do not trust interface controls; their input must be carefully constrained.

Compounding this situation are applications that let developers become users by ex-

posing functionality that another program can call. In this case, programmers must consider two issues. First, has the calling program supplied valid and meaningful parameters? Obviously, developers should check parameter validity just as they would variable input through a GUI control. Second, are there side effects to calls that preclude other calls from being executed properly? One common shortcoming is making the same call twice in a row. Once a call has opened a file, the software expects that file to be read, not reopened (it was never closed), so a second call causes the software to fail. Because there is no GUI to conveniently protect the application from such invalid inputs, developers must be responsible for it.

Human users, whether the typical variety using an application through a GUI or developers using an application through a programmable interface, pose special but familiar problems to software developers and testers. But other types of users are not so familiar, and our handling of them can ensure the robustness of our applications or make them vulnerable.

## The operating system user

As discussed earlier, the OS is the intermediary between all physical users and the application. It also interacts directly with an application by supplying memory, file handles, heap space, and so on. We call this latter part of the OS the kernel.

The Windows kernel, for example, exports over 800 different functions, each of which has at least two return values (for success and error conditions). This is indeed a challenge for developers who all too often trust the kernel explicitly. When they allocate memory, they expect memory, not a return code saying "Sorry, it's all gone." Every time the program allocates memory, it has to check the return code for success before continuing its task. If it does not, it will behave unexpectedly (or perhaps fail) when low-memory conditions occur.

The experiment cited in the "Typical Reactions to Low-Memory Situations" sidebar is good evidence that developers put too much trust in their system calls always behaving as expected. Certainly this trust is often well founded, but when an OS is suffering a malicious attack such as a packet storm, the application cannot and should not trust them, especially if the application is expected to work safely and securely. But most applications go blissfully unaware of such problems or react by failing.

## The API user

Similar to the OS user, APIs are external libraries that an application can use to store data and perform various tasks. For example, an application can make SQL queries to an external relational database or use APIs for programming sockets, performing matrix algebra, manipulating strings, and doing any number of other commonly reused functions.

These functions usually provide reliable service, but they are also software and can contain bugs that the programmer must consider. Functions can also provide unsuccessful return values that the program must handle properly. For example, a database could be offline or contain corrupted data; a socket connection could drop packets or fail to acknowledge a transmission due to network congestion. Developers must be aware of the possibilities, and testers must figure out how to test them.

Trusting other software components leaves our own programs vulnerable to their bugs. Not fully understanding how components behave when they fail subjects our software to unexpected (but legitimate) return codes.

## The file system user

Files (binary or text) are users, and their contents are inputs—so they can also be invalid. Files, even binary ones, are easy to change outside the confines of the applications that create and process them. What happens, for example, when a privileged

**Deciding which inputs to trust and which to validate is a constant balancing act.**

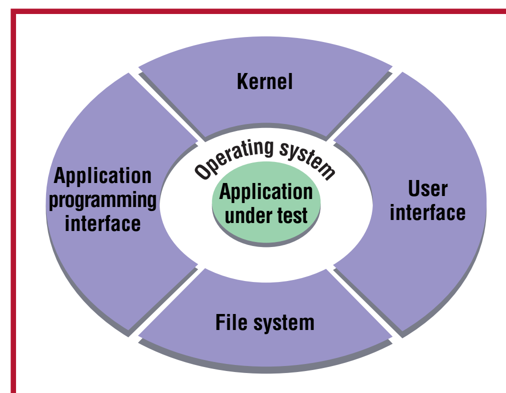


**Figure 1. The execution environment for application software. All communication must go through the operating system to reach the application.**

**Data is the lifeblood of software; when it is corrupt, the software is as good as dead.**

user changes the permissions on a file that another user is editing?

An application's defenses against corrupted files are usually weak.[5] As long as the file extension is correct, the "magic string" is at the top of the file (as an identifier), and the field delimiters are in place, the application often reads the contents without checking them.

I cannot overstate the danger of this. Reading files without validating content means that you are introducing unknown data to your software to be used for internal computation and storage. Data is the lifeblood of software; when it is corrupt, the software is as good as dead.

Software applications are controlled by their environment. Unless the system checks the validity of every single input from every single user, the software can fail.

But such overzealous defensive programming is probably not feasible for most software vendors. Indeed, based on my experience, many vendors are probably unaware of the extent to which their software applications depend on "invisible" users. Still, some precautions are in order:

- During software design, programmers must continually realize when they are going outside the application's boundaries to get input from users—and not just the human variety. They must understand that, ultimately, all users are software and realize that users can be buggy, so they must not trust their input. They must decide which inputs to verify and which inputs to trust with full knowledge of the consequences.
- Testers must accept the challenge to simulate as many anomalous inputs as possible that are realizable and pose a threat to the application and the system. This will mean new tools, new techniques, and purposeful consideration of all software users, even the ones we can't see.

For systems that execute in a networked environment or protect sensitive data, understanding invisible users is as important as understanding human users. Software developers beware: you never know with whom your software is communicating.

## References

1. R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.
2. J. Richter, *Programming Applications for Microsoft Windows*, Microsoft Press, Redmond, Wash., 1997.
3. B. Miller, L. Fredrikson, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Comm. ACM*, vol. 33, no. 12, Dec. 1990, pp. 32–44.
4. A. Ghosh and M. Schmid, "An Approach to Testing COTS Software for Robustness to Operating System Exceptions and Errors," *Proc. 10th Int'l Symp. Software Reliability Eng.*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 166–174.
5. J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley & Sons, New York, 1998.

## About the Author

**James A. Whittaker** is an associate professor of computer science at the Florida Institute of Technology, Melbourne. He founded the Center for Software Engineering Research with grants from a number of industry and government funding agencies. The Center performs contract software testing and research focusing on the reasons why software fails and what can be done to make software safer and more reliable. Contact him at the Dept. of Computer Sciences, Florida Tech, 150 W. University Blvd., Melbourne, FL 32901; jw@se.fit.edu.