Chapter 40
Classic Fuzz Testing: GUI Studies

Revision 3.0, December 2025.

Objectives

- Learn about what was tested in three graphic user interface (GUI) fuzz studies.
- Understand the type and quality of errors that were found in these studies.
- Learn about the programming practices that led to these errors.

40.1 Introduction

In 1995, the world was moving away from a command-line-only world to one where GUI-based applications were becoming increasingly important. That led to the challenge of how to feed random input to these applications.

There are a couple of major technical challenges that we had to overcome. First, we have to be able to identify the geometry and placement of the application windows. Second, we need to generate random keyboard and mouse events and have a way to feed them to the application.

Making this more complex is that each window system is quite different from the others. And designing the test jigs to work with these systems requires detailed knowledge of how they work.

Our first platform, tested in 1995, was the UNIX X-Windows system. X-Windows runs as a server and the application program connects to that server to send and receive messages to do output and input.

To intercept the messages and modify them, we created a test jig that consisted of a web proxy process called xwinjig. The application process connects to xwinjig thinking that it is the actual X-Window server, and then xwinjig connects to the window server. xwinjig queries the X server to learn about window positions and sizes. This structure is illustrated in Figure 1.

By communicating with xwinjig, we can inject keyboard and mouse events to be sent to the application process, either totally random events or well-formed ones.

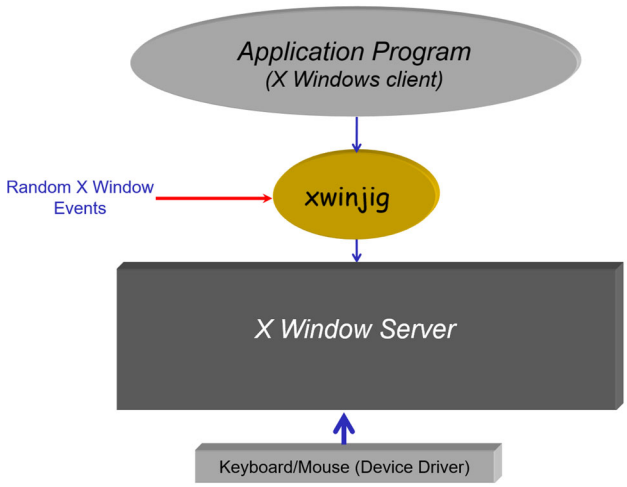


Figure 1: Intercepting Messages Between X-Windows Application and Server

In 2000, we investigated Microsoft Windows Win32 applications. The logical picture for testing Microsoft Windows applications looks similar to UNIX, but structurally is quite different. This structure is illustrated in Figure 2. The Windows operating system uses event queues for communications, and the Win32 window system uses these event queues to communicate with a GUI-based application. There are a few places where Windows allows you to insert events into these queues, either as raw events or well-formed keyboard and mouse events.

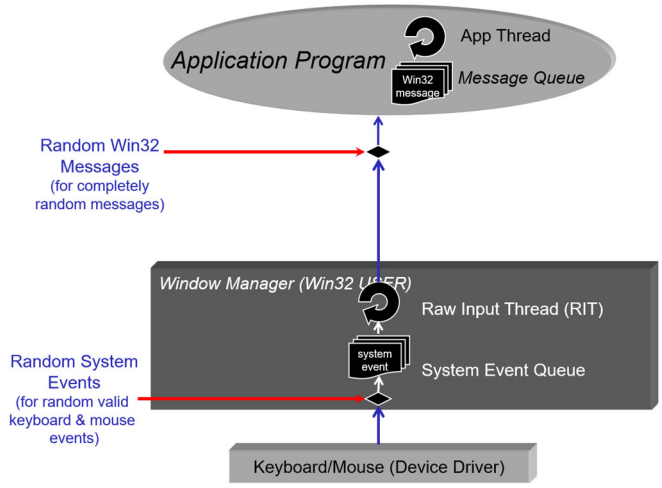


Figure 2: Intercepting Events Between Windows Win32 System and Application

In 2006, we investigated MacOS X. The MacOS Aqua window system also uses event queues to communicate with the application. And, as in Win32,

there are mechanisms for inserting raw and valid keyboard and mouse events to be sent to the application. This structure is illustrated in Figure 3.

As we have discussed, testing a GUI-based application has a more complex structure than for testing command line applications. The communication between the window system and the application is typically through messages. These messages have a fixed message format and applications use a library to construct, send, and receive these messages. This is the first application of structured fuzz testing.

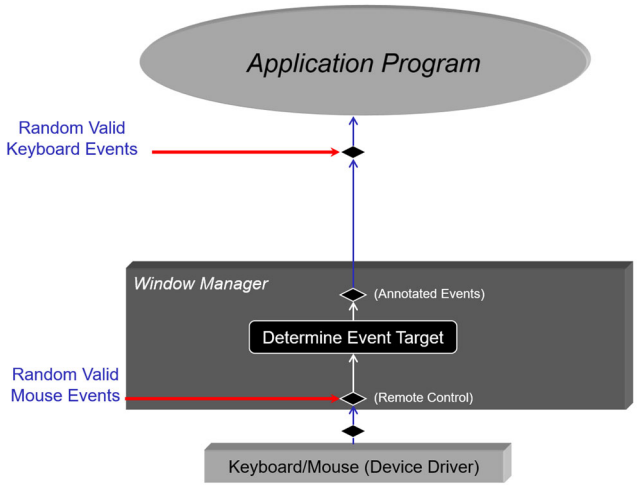


Figure 3: Intercepting Events Between MacOS X Aqua Window System and Application

We experimented with three kinds of testing on GUI-based applications:

Completely unstructured input, meaning completely random bytes: This kind of testing mostly exercises the library's protocol processing code and would rarely test any of the application's logic.

Input with valid message structure: In one case, we used mutation testing, taking valid messages and randomly modifying them, with the possibility of resulting in either valid or invalid message. This approach will often generate invalid input but can also generate valid inputs that will test deeper into the logic of the application code.

Input that conforms in all ways to valid keyboard and mouse events: This means that the messages will be for valid operations and make sense. For example, a key down event for a given key is always matched with the key up event, and X-Y coordinates for the mouse events are always within the bounds of a valid window.

Almost all GUI-based programs use libraries to interact with the window system, and those libraries construct valid event messages, so it is unlikely

in real life to see problems from the first two categories. Results from the third category are most representative of what a real program would do.

40.2 Results from the GUI Tests

We will discuss results from the valid keyboard and mouse event studies (the third category). If you want to learn more about the results from the other two categories, you can refer to the original research papers cited in Chapter 39.

The quantitative results from the GUI studies appear in Figure *4*. In 1995, we see that we could crash more than a quarter of the X Windows applications on UNIX. In 2000, we could crash around half of the Win32 applications on Windows. And in 2006, we could crash almost three quarters of the GUI-based applications on MacOS. Clearly a sad state of affairs.

In retrospect, it would have been great to have comparable studies done on each platform in each year, but unfortunately we do not have that data. A good project would be to do such a comparison study now to see both the absolute and relative numbers for current applications.

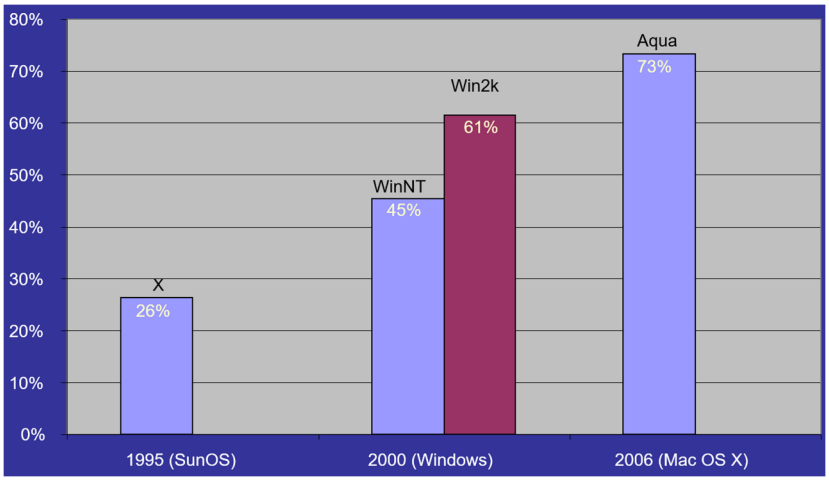


Figure 4: Crash/Hang Rates for the GUI Studies

Each of the GUI-based studies crashed a variety of programs. In Figure *5*, we see the list of utilities tested on X Windows; the ones in **bold red** are the ones that failed. Since this study was done quite a while ago, the names of the programs are likely to be unfamiliar. However, we see some interesting ones like a mail client (xmh), text editor (xedit), Postscript (predecessor to PDF) previewer (ghostview), and a couple of video games (xminesweep and xneko).

X-Windows: 10 Failures of 38 GUI Programs Tested					
bitmap	emacs	ghostview	idraw	mosaic	mxrn
netscape	puzzle	rxvt	xboard	**xcalc**	xclipboard
xclock	**xconsole**	xcutsel	xditview	xdvi	**xedit**
xev	xfig	**xfontsel**	**xgas**	xgc	xgdb
xmag	**xman**	**xmh**	**xminesweep**	**xneko**	xpaint
xpbiff	xposit	xspread	xsnow	xterm	xtv
xv	xweather				

Figure 5: X-Windows GUI Programs Tested with Crashed Programs in **Red/Bold**

And Figure *6* shows the same type of list for the Microsoft Windows Win32 study. You see some pretty interesting programs that failed, like the Netscape web browser (very important at the time), Code Warrior IDE, Eudora email client, SecureCRT SSH client, and Microsoft Word, PowerPoint, Internet Explorer, Wordpad, Access, and Visual C++. Quite a list!

Windows Win32: 16 Failures of 33 GUI Programs Tested				
Access 97	**Access 2000**	Acrobat Reader 2000	Calculator 4.0	CD Player 4.0
Codewarrior Pro 3.3	Command Anti-Virus 4.54	**Eudora Pro 3.0.5**	Excel 97	Excel 2000
Framemaker 5.5	FreeCell 4.0	Ghostscript 5.50	Ghostview 2.7	GNU Emacs 20.3.1
Internet Explorer 4.0	Internet Explorer 5.0	**Java Workshop 2.0a**	**Netscape Comm. 4.7**	Notepad 4.0
Paint 4.0	**Paint Shop Pro 5.03**	**PowerPoint 97**	**PowerPoint 2000**	**Secure CRT 2.4**
Solitaire 4.0	Telnet 5 for Windows	**Visual C++ 6.0**	Winamp 2.5c	**Word 97**
Word 2000	**WordPad 4.0**	**WS_FTP LE 4.50**		

Figure 6: Windows Win32 GUI Programs Tested with Crash Programs in **Red/Bold**

On MacOS, in Figure 7, we see an even bigger collection of programs that failed, including Adobe Acrobat; Mozilla Firefox and Thunderbird; Apple's iTunes, Opera, Finder, and Mail; and many of Microsoft's programs. It is all pretty distressing.

MacOS X: 22 failed of 30 GUI Programs Tested				
Acrobat Reader 7.0.5	**Adium X 0.87**	**Aquamacs Emacs 0.9.7**	Calculator 10.4.3	**Camino 0.8.4**
Dictionary 10.4.3	**Excel 11.2.0**	**Finder 10.4.3**	**Firefox 1.5**	GarageBand 2.0.2
iCal 10.4.3	**iChat 10.4.3**	iDVD 5.0.1	iMovie 5.0.2	**Internet Explorer 5.2.3**
iPhoto 5.0.4	**iTunes 6.0.1**	**Keynote 2.0.2**	**Mail 10.4.3**	**OmniWeb 5.1.13**
Opera 8.51.2182	**Pages 1.0.2**	**PowerPoint 11.2.0**	Preview 10.4.3	Safari 10.4.3
Sherlock 10.4.3	TextEdit 10.4.3	**Thunderbird 1.5**	**Word 11.2.0**	**Xcode 2.2**

Figure 7: MacOS X GUI Programs Tested with Crash Programs in **Red/Bold**

40.3 Analysis of a Couple of the Coding Errors Found

We now look at a couple of the programming practices that led to these failures. Not surprisingly, these practices were not too different from the ones we found in the command line studies described in Chapter 39.

Note that since the list of failed GUI programs, especially those from the recent studies, includes many commercial products, we did not always have access to the source code to do full debugging to identify the cause of the failure.

The popular editor `emacs` had a pointer dereferencing error on Windows caused by not checking a pointer before using it. The function shown in Figure 8 is a standard Win32 callback function that handles an input event. The third and fourth parameters, `wParam` and `lParam`, are pointer values and come from the input. On line 6, the function copies the fourth parameter and then on line 9, dereferences that pointer, never checking the point for validity before using it. Of course, with an invalid pointer, a crash was not a surprising result.

```
00  LRESULT CALLBACK
01  w32_wnd_proc (hwnd, msg, wParam, lParam)
03  {
04    ...
05    POINT *pos;
06    pos = (POINT *)lParam;
07    ...
08    if (TrackPopupMenu((HMENU)wParam, flags,
09        pos->x, pos->y, 0, hwnd, NULL))
10      ...
11  }
```

Figure 8: Unchecked Pointer Bug (`emacs`, 2000)

Figure 9 shows another crash from an unchecked pointer, this time in the `Firefox` web browser. The first line of code is a little complex to understand because it is a macro. This statement is an assignment statement using a macro that does a type cast. The value from `mObservers.ElementAt(i)` is assigned to the variable `observer`, cast as a pointer to type `nsIDocumentObserver`. `observer` is then dereferenced without checking if the pointer was valid.

```
nsIDocumentObserver* observer =
    NS_STATIC_CAST(nsIDocumentObserver*, observers.ElementAt(i));
observer->ContentAppended(this, aContainer, aNewIndexInContainer);
```

Figure 9: Unchecked Pointer Bug (`Firefox`, 2006)

We were not able to fully debug many of the commercial applications that we crashed because the code was proprietary. However, we were able to get some idea where the crashes occurred by identifying in what libraries and functions they happened. Several crashes happened in MacOS system libraries, including:

`libobjc`, the Objective-C runtime library;

`WebCore`, Apple's HTML and JavaScript engine;

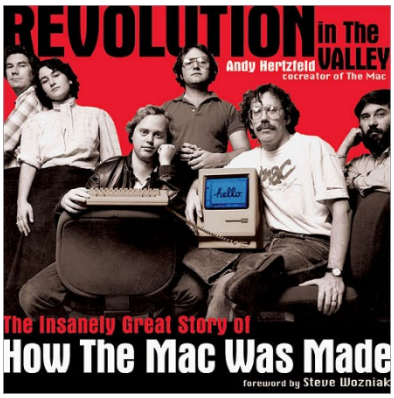
`libsystem`, the collection of MacOS system libraries like `libc`, `libm` and `libpthread`; and `com.apple`, another MacOS collection of system libraries.

These crash results shows you how fragile are the interfaces to many of these system libraries. You can decide whether the blame sits more on the user code that provided dangerous values to the library or the library itself that provided weak interfaces and insufficient parameter checking. In general, across different operating systems, we see system libraries as having weak error checking.

Application	Function	Library
iChat	objc_msgSend_rtp	libobjc
Pages		
Xcode		
Keynote	objc_msgSend	
Mail	khtml::maxRangeOffset	com.apple.WebCore
OmniWeb	khtml::RenderTable::layout*	
Sherlock	khtml::RenderFlow::detach	
Adium X	mach_msg_trap	libSystem
Adobe Reader	szone_calloc	
Camino	SetGWorld	com.apple.QD
Finder	MDQueryDisableUpdates	com.apple.Metadata
Terminal	__bigcopy	commpage

Figure 10: Location of MacOS GUI Application Crashes (2006)

40.4 Apple's Foray with Random Testing on the Mac



As you might guess, random testing was not totally new at the time that we did our first tests. From a book written in 2004 by Andy Hertzfeld[1], who was one of the of the original members of the Macintosh software test, we see an example of such testing. The founders of Apple used a form of random testing in 1983 to test the original Mac computer. This program was called "The Monkey" and it apparently was pretty effective at finding hard-to-replicate bugs. It is not clear why Apple stopped using this form of testing.

40.5 Summary

In this chapter, we have learned about the GUI-based application fuzz studies that were based on the classic fuzz testing technique. As we noted, this was

[1] Andy Hertzfeld, **Revolution in the Valley: The Insanely Great Story of How the Mac Was Made**, O'Reily, 2004.

the first application of structure fuzz testing. This testing produced a dramatic number of crashes and that is definitely a concern.

40.6 Exercises

1. The older GUI fuzz studies (from 1995 and 2000) tested many applications that are not familiar today. Choose a few examples from Figure 5 and Figure 6, investigate from where these applications came and what they did.
2. (Advanced) How would you extend fuzz testing of GUI or phone applications to include camera and microphone inputs? What kind of test harness would you need to accomplish this kind of testing?