Testing Exception and Error Cases Using Runtime Fault Injection

James A. Whittaker
Florida Institute of Technology
150 W. University Blvd.
Melbourne, Florida 32901, USA
++ 1 (321) 674 7638
jw@se.fit.edu

Florence E. Mottay
Florida Institute of Technology
150 W. University Blvd.
Melbourne, Florida 32901, USA
++ 1 (321) 674 7473
fmottay@se.fit.edu

Ibrahim K. El-Far
Florida Institute of Technology
150 W. University Blvd.
Melbourne, Florida 32901, USA
++ 1 (321) 674 7473
ielfar@acm.org

Florida Tech Computer Science Technical Report Number CS-2001-07

ABSTRACT

Fault injection deals with the insertion or simulation of faults in order to test the robustness and fault tolerance of a software application [8]. Such measures are generally performed on software that is mission critical, to the extent that failure could have significant negative ramifications. Actual injection of faults can be performed either at compile time, when additional code is inserted to force error conditions to evaluate to true, or at runtime during which faults are injected into the software's execution environment. This paper focuses on the latter type of fault injection and presents a new mechanism for inserting environment faults. In addition, insight is provided into fault selection based on an analysis of runtime behavior. This paper presents a methodology and tool for performing runtime fault injection, both of which are demonstrated on a commercial software product.

1. INTRODUCTION

For the purposes of our discussion of fault injection, code comes in two forms:

1. *Functional code* is code that accomplishes the mission of the software by implementing user requirements. In other words, it is the code that does the work necessary for users to fulfill their purpose in using the software.
2. *Error handling code* is code that keeps the functional code from failing. Examples include code to check inputs for validity and code that ensures stored data does not exceed its defined type and value range.

By its very definition, functional code is readily accessible through the software's interface(s) (be they graphical user interfaces or programming interfaces). In fact, testing functional code is a fairly well established discipline, though often imprecise [9]. On the other hand, exercising error-handling code is generally trickier and may require more extreme measures. Of course, some error conditions are easy to handle; for example, some conditions require only that certain input values be entered incorrectly to be satisfied. But other error handling code may require considerably complicated environmental circumstances to arise before it will execute [8].

Consider the case in which developers write code to guard against a full storage medium. The straightforward way to set up this anomaly is for testers to generate and maintain many large data files—files that are large enough to fill the capacity of the local storage device. Not only are such files hard to generate, but keeping them around means that the storage device can serve no other purpose (because it is full) than to house a single test case. And full media is only one case out of many faulty file system possibilities. We also need to consider file corruption, access privileges (read-only, etc.), file permissions and damage to the actual media, among other scenarios.

Indeed, the file system itself is only one possible part of the environment that developers write error code against. We must also consider memory calls, network APIs, databases, third party components and controls, etc. All of these environmental elements can fail in ways that an application must expect and guard against especially when its mission is compromised [9].

In this paper, we discuss injecting faults into an application's environment at runtime to effectively and accurately trigger failures in a manageable fashion. We begin by describing the runtime injection mechanism and provide specific examples for Microsoft's Windows® operating system. Next, we discuss in detail the types of faults that can be injected and the situations in which testers should use specific failure scenarios. Finally, we illustrate the technique by outlining results from a case study performed by the authors on a commercial software product.

2. A MECHANISM FOR RUNTIME FAULT INJECTION

Source-based fault injection can be complicated to achieve but is easy to explain: source statements are modified so that specific faulty behavior is attained [1],[2],[6]. When faults are injected to trigger exceptions, source statements are actually added so that internal data can be set to values which cause exception conditions to evaluate to true [3].

But source-based fault injection requires access to source code and, in most cases, the cooperation of the original developers, which is not always a given [8]. Release pressure is one reason that developers refuse to write such code. Further, it is often the case that many testers have no access to the source code. Either they are outsourcers or the build culture at their company does not support such involvement.

Regardless of the organizational factors that complicate source-based fault injection, runtime fault injection holds the benefit that the faults are more realistic. By inserting faults into the environment instead of the application, the latter, free of any additional code that may introduce unwanted behavior, is forced to react in exactly the same way as if the failures were real and not triggered by testers.

Environment faults can be forced either by reproducing the causal scenario or by simulation. For example, consider the case of the ubiquitous network through which many applications communicate with other applications or services. To test fault tolerance of network applications, one might physically damage the network by, say, unplugging the cable or by sabotaging the network adapter. Further, one could cause a busy network by generating large amounts of bogus traffic (say by sending constant command line pings from a few dozen machines).

But these same results, and many others, can be achieved by simulating the exact same circumstances so that the fault affects the application under test but not the rest of the system. The key is to realize that any environmental fault will manifest as failed system calls made by the application. For example, an application sees a network outage as a series of failed calls made to the local socket API. The application sees low memory as failed calls to the kernel. The application sees file corruption as CRC errors raised by the function `CreateFile`, and so forth.

Ultimately, there is the reality of a failure and the reality of what an application actually sees when the failure occurs. It is this latter entity that we can recreate and it is at the system-application boundary that faults can be injected. These faults will affect only the system under test, allowing the machine to be useful for other purposes.

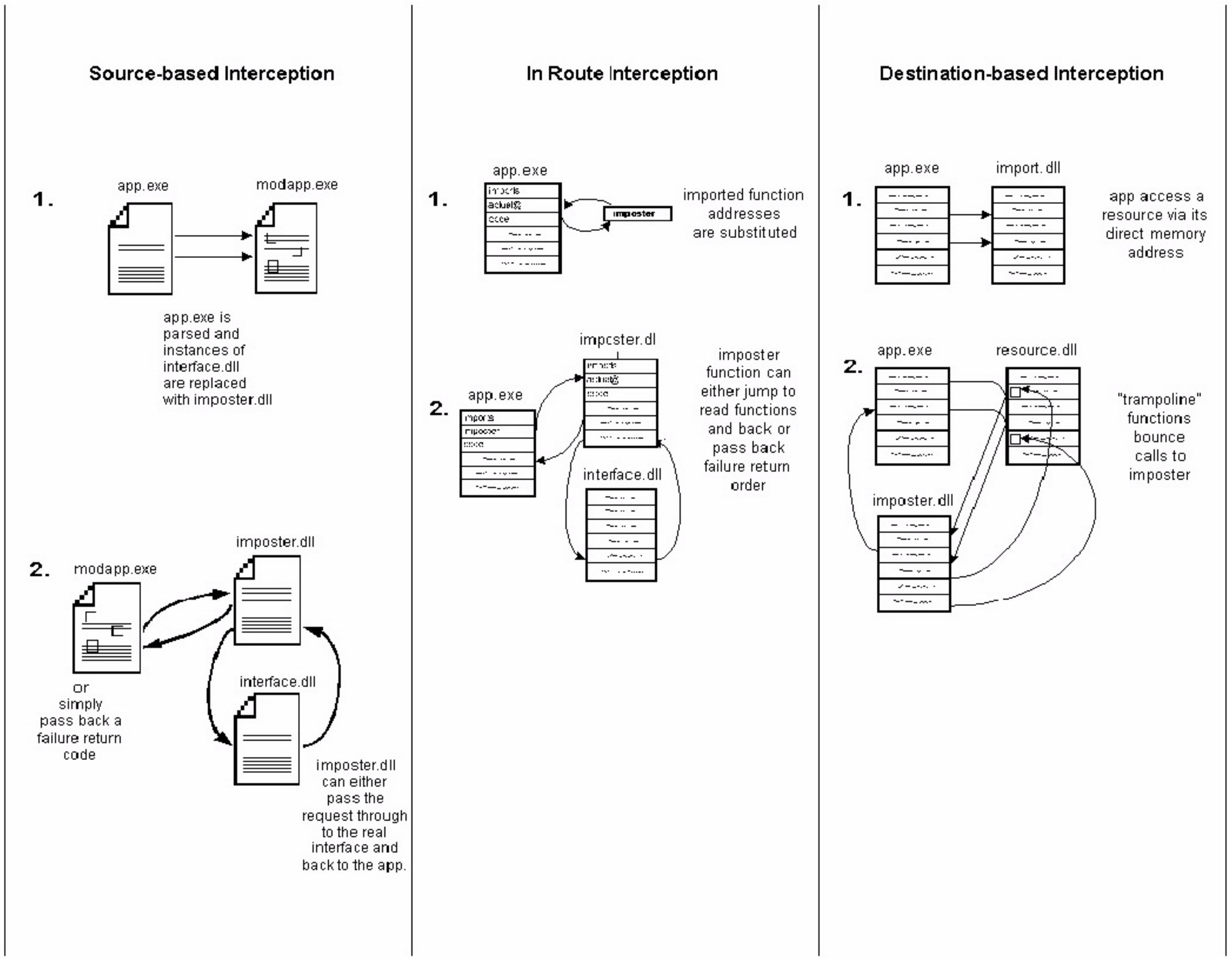
In order to understand how to interpret faults as failed system calls, we must first be able to capture system calls before they reach their destination. Then we must be able to record exactly how real faults manifest in the error codes and return values of these calls.

There are at least three ways to accomplish system call interception.

1. *Source-based interception* requires actual editing of a binary and replacing instances of the destination API with an imposter API. The imposter API then simply acts as a pass-through mechanism. For example, using a hex editor it is easy to search an executable for the string kernel32.dll and replace it with mykernel32.dll. We must then write imposter functions in mykernel32.dll with the same name as the functions in kernel32.dll that we want to fail. The imposter functions simply log the call and then call the real function in kernel32.dll. When kernel32.dll responds to mykernel32.dll, the imposter simply passes the error codes and return values back to the application.

2. *In route interception* can achieve the same effect as source-based interception without having to change an application's binary image on disk. Using techniques published in [7], one can modify addresses in function dispatch tables to divert calls to imposter functions. The imposters then act as pass-through mechanisms as above. Of course, in route interception only works when calls are routed through a centralized function dispatch mechanism like import address tables. Since these tables are stored in memory, the application's binary does not have to be modified on disk.

3. *Destination-based interception* requires inserting code into the function being called by the application. Unlike in route interception, which modifies memory addresses that are part of the application's code space, destination-based interception requires modifying the code space of the target function. In our implementation, we insert jump statements into the first few bytes of a function and copy those bytes to the imposter function. When the application makes the function call, the jump statement transfers control to the imposter function which will first executes its code and then transfer control back to the next executable memory location of the original function (i.e., past our inserted jump statement).


Source-based Interception
1.
app.exe
modapp.exe
app.exe is parsed and instances of interface.dll are replaced with imposter.dll
2.
modapp.exe
imposter.dll
interface.dll
or simply pass back a failure return code
imposter.dll can either pass the request through to the real interface and back to the app.
In Route Interception
1.
app.exe
imposter
imported function addresses are substituted
2.
impcster.dll
app.exe
interface.dll
imposter function can either jump to read functions and back or pass back failure return order
Destination-based Interception
1.
app.exe
import.dll
app access a resource via its direct memory address
2.
app.exe
resource.dll
imposter.dll
"trampoline" functions bounce calls to imposter

The above figure illustrates each of the three types of interception.

3. Fault Selection

We have employed two types of fault selection strategies and developed tools to help carry out each type. The first strategy consists of recording function calls made by an application and then systematically failing each call everywhere it is used in the application. We call this method systematic, call-based fault injection. For example, if we record that the kernel call `LocalLock` is used each time an application accesses a file, then we can cause `LocalLock` to fail and force the software through paths that have file opens, reads, writes, etc, so that the application sees the failure of `LocalLock`. Obviously, this is a time consuming and painstaking way to inject faults. The second strategy consists of staging a particular environment fault, recording the pattern of failed function calls caused by the fault, and then simulating that pattern in other parts of the application. We call this method *pattern-based fault injection*. For example, we might stage an unresponsive network by unplugging the Ethernet cable and record that the application sees failed return code from any number of socket APIs. We can then fail these same APIs as a simulated substitute for physically unplugging the Ethernet cable.

3.1 Pattern-based Fault Injection

The ultimate question for fault injection is: **What faults should be injected**? The answers to this question are varied:

The faults should collectively cause all of the error code to be executed and exceptions to be tripped.

This is a typical developer-centric answer. As desirable as it is to execute all the source code of an application under test, this is usually unachievable given today's relatively short development cycles and aggressive deadlines. Other difficulties include access to source code and the use of sophisticated code coverage tools. Again, in practice, not all testing teams have access to source code, development teams, or the required tools that would enable them to stage code-based fault injection.

Only the faults that can be readily staged in the testing lab should be selected.

This, on the other hand, is a typical tester-centric answer. It may be hard for some to imagine that testers are required to consider and run scenarios that cannot be accomplished outside the testing lab. However, testing labs are often not representative of the setup and environment of the real world. Users typically have more data, more machines, bigger networks, more software, and a wider variety of hardware, peripherals, and drivers than can be in a lab. Users, therefore, are a great source of realistic scenarios that may not have been anticipated by developers and that may cause unexpected failures.

Only faults that may realistically occur in the field need to be injected.

Users expect that software will work well in their uncontrolled, generally unpredictable environments. However, since such environments are difficult or impossible to stage in the testing lab, we have developed a tool to help simulate some of the more common faulty scenarios. We call the general principle the *Hostile Environment Application* Tester and the tool "Canned HEAT. The purpose of Canned HEAT is to stage some realistic problems in the environment in an easy-to-use way.

Canned HEAT works on a simple principle. Every faulty environment causes digital symptoms that the application recognizes and that we can recreate. Take a network that has gone down for example. This can be caused by an unplugged cable, a misconfigured adaptor, faulty network software, or network congestion. The application only recognizes the symptom that certain API calls, say to the network port, are failing. That is, instead of working as expected, they are returning failure codes to the application. Therefore, any number of actual faults may end up producing the same symptoms. Canned HEAT is a tool that reproduces these symptoms so that the application runs as if an actual failure has occurred.

3.1.1 Memory Faults

The amount of memory that an application uses varies according to the task it is performing. Some tasks require very little memory and are unlikely to cause memory to be depleted. Other tasks consume vast amounts of memory. Such tasks along with other applications simultaneously running may deprive the application of the amount of memory necessary for normal operation. In order to determine those features that are memory intensive, Canned HEAT is equipped with a monitor that keeps track of an application's use of memory. Once a list of these features is gathered, the first set of tests consists of continually lowering the available memory threshold to determine where (or if) the application begins to falter.

The next step is to run scenarios that will test the application's reactions to varying memory conditions. Canned HEAT can randomly vary the amount of memory available to an application. The intent is to simulate the real world scenario in which background applications access memory at sporadic times. The application is then run through its paces, concentrating on the memory-intensive features identified earlier.

The final test to perform is fault injection, and, with Canned HEAT, this is as simple as running the application and selecting a fault's check box at any time.

Consider the following example using Microsoft® Internet Explorer®.

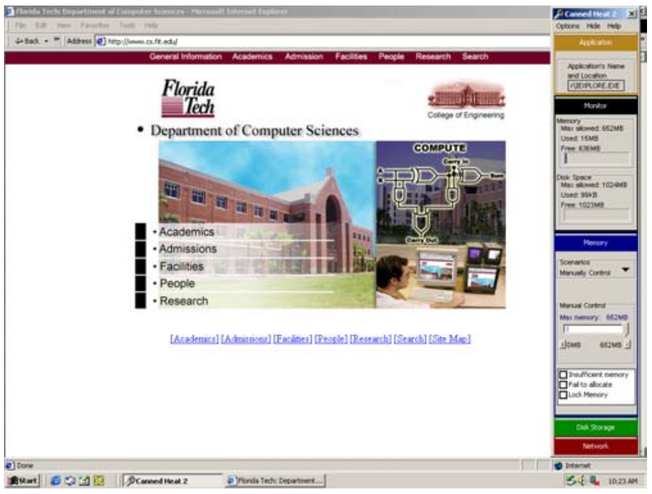
Step 1. Use the application and determine which features are memory intensive.

This step can actually be performed while the application is being tested under ordinary circumstances. Simply launch the application under Canned HEAT and pay close attention to the memory monitor while you are using the application. Make a note of which features use the most memory. Obviously, disk-intensive operations like reading and writing files will cause memory to be used, but also, loading rich images, processing large files and performing any computationally intense function will require memory usage.

Step 2. Determine the application's lower bound threshold of tolerance to low memory.

It is now the time to see how the application fares with restricted memory resources. This can be accomplished by forcing the features to be exercised and simultaneously restricting the application's access to memory resources using Canned HEAT.

Canned HEAT has a convenient slider bar under the memory tab for this purpose. By simply sliding the bar to the left, the available main memory is decreased (see figure below).


Application's Name and Location: /IEXPLORE.EXE

Memory
Max allowed: 652MB
Used: 15MB
Free: 636MB

Disk Space
Max allowed: 1024MB
Used: 99KB
Free: 1023MB

Scenarios: Manually Control

Manual Control
Max memory: 652MB
0MB 652MB

☐ Insufficient memory
☐ Fail to allocate
☐ Lock Memory

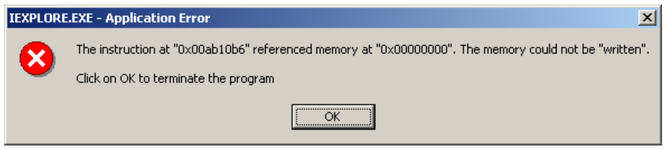
Note the slider bar is all the way to the right, allowing the application access to all available memory. However, if we move it to the left and continue to use Internet Explorer's memory intensive features, we note that around 35MB, things begin to slow down tremendously. Further, if we take away all but about 15MB, Internet Explorer ceases to work at all.

Step 3. Run Canned HEAT's scenarios that randomly vary available memory.

Once this is determined and reported to development, the next set of tests to run concerns the application's ability to perform well under tremendously varying memory conditions.

Selecting the varying memory scenario will make the memory control slider unavailable, meaning that Canned HEAT has assumed control of deciding when and how much memory will be available to any given request made by the application under test.

Using this scenario will often crash applications even when the human user is not working with them. This is because any given memory call may result in an artificial failure being injected by Canned HEAT. Such is the case with Internet Explorer as shown below.



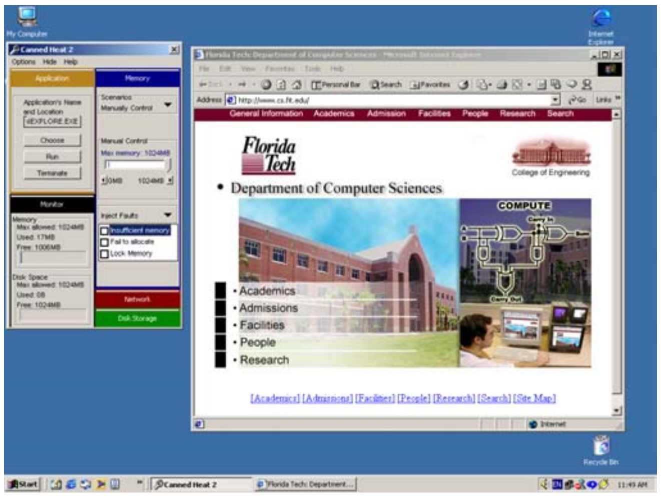
IEXPLORE.EXE - Application Error

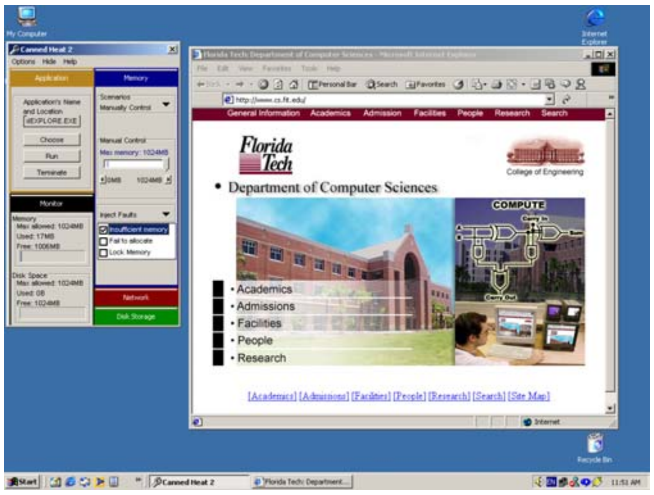
The instruction at "0x00ab10b6" referenced memory at "0x00000000". The memory could not be "written".

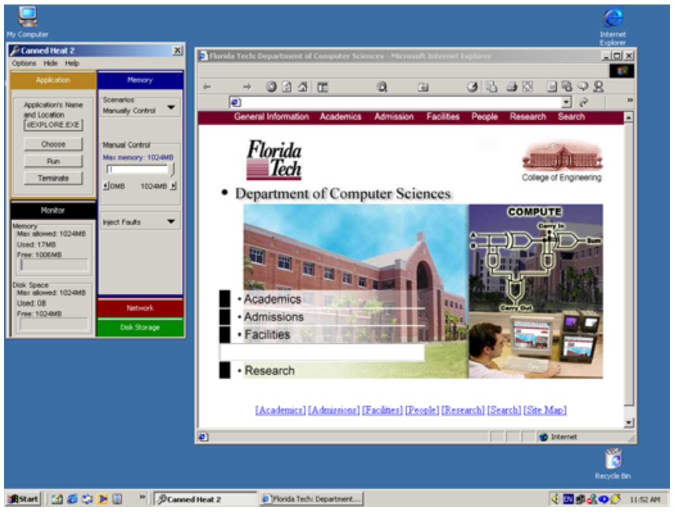
Click on OK to terminate the program

Step 4. Inject faults at runtime during memory use.

Finally, the last set of tests involves injecting specific faults. Whereas the last two steps simply fail memory calls according to the amount of available memory, our tool allows individual faults to be injected regardless of the amount of memory available. The following series of figures shows an example of this in Internet Explorer. We use Canned HEAT to simulate an "insufficient memory" fault and watch as IE's controls simply disappear due to lack of adequate memory resources. Eventually, IE will hang.







3.1.2 Network Faults

We test network faults in the same four-stage process we have just demonstrated for memory faults. First, we will use the application and determine which features cause network activity. Second, we will use Canned HEAT's slider bar to slow the network until our application is unacceptably slow or until it crashes or hangs. Third, we will run scenarios that will vary the network speed over time, concentrating on those features that cause the most network activity. Fourth, we will inject specific faults, one at a time, and monitor the application's resulting behavior.

Let's consider an example.

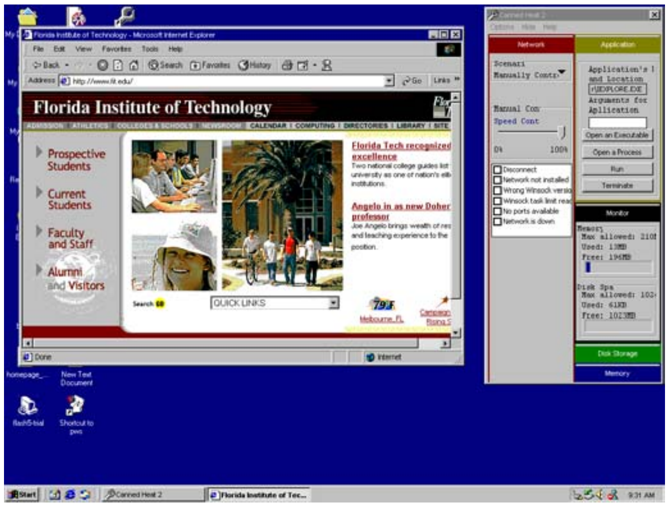
Step 1. Use the application and determine when it hits the network port.

The browser's most intense use of the network port occurs during file downloads and when web pages are served to it.

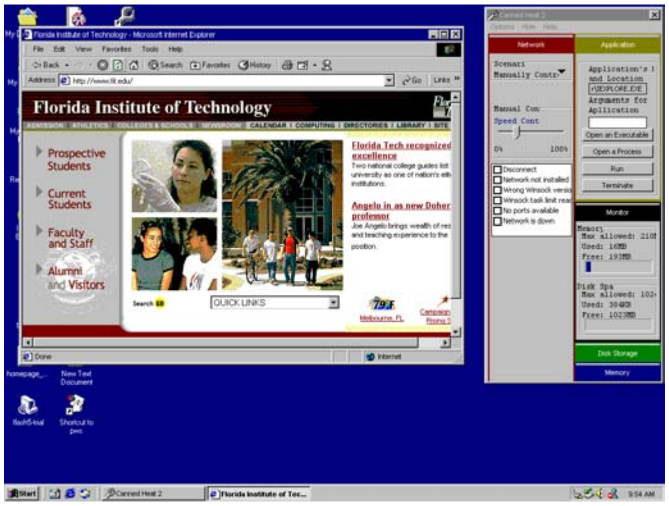
Step 2. Determine the application's lower bound threshold of tolerance to a slow network.

Once you have determined when the application hits the network port, it is interesting to find out the behavior of the application by reducing network speed. This can be achieved by using the application while manually reducing the network speed using Canned HEAT's network speed slider bar.

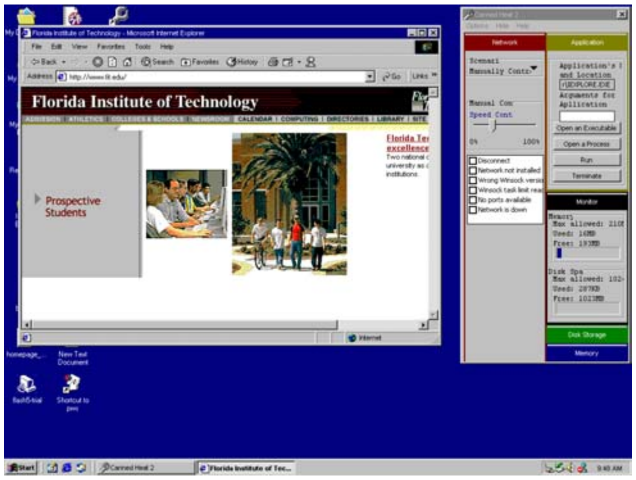
Canned HEAT allows the user to manually control the network speed the same way it does for available memory. Using the slider bar, the user can easily adjust the network speed to the desired percentage of the full capacity of the machine, on which the application is being tested. (see figure below)



The next two screenshots demonstrate the use of the network slider bar. The first one shows a perfectly loaded page while the network speed is 33% of its maximum. This shows that the loading of a page similar to this one does not require more than 33% of network speed.



This next screenshot though, shows an incomplete page (some pictures and menu titles are missing) with a network speed around 30%.

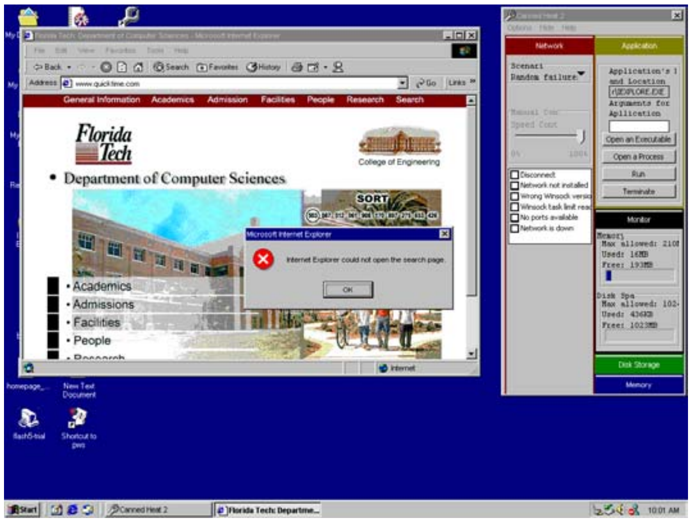

Florida Institute of Technology - Microsoft Internet Explorer
Florida Institute of Technology
Prospective Students
Network
Application
0%
100%
Monitor
Disk Storage
Memory

We thus determined the network speed threshold for which Internet Explorer can load accurately a page similar to the Florida Tech homepage, with respect to the number of graphics, animation etc. If we further lower the network speed, IE will not be able to load a page anymore.

Step 3. Run Canned HEAT's scenarios that randomly vary network speed.

The next step is to run tests using scenarios. Canned HEAT is programmed to simulate random network failures while the application is running. Examples of such failures are disabled network connection, unresponsive network port or failure of socket API's.

The following screenshot demonstrates the use of the Canned HEAT's random failures scenario. The network slider bar is unavailable when running random scenarios, as was the memory slider bar.



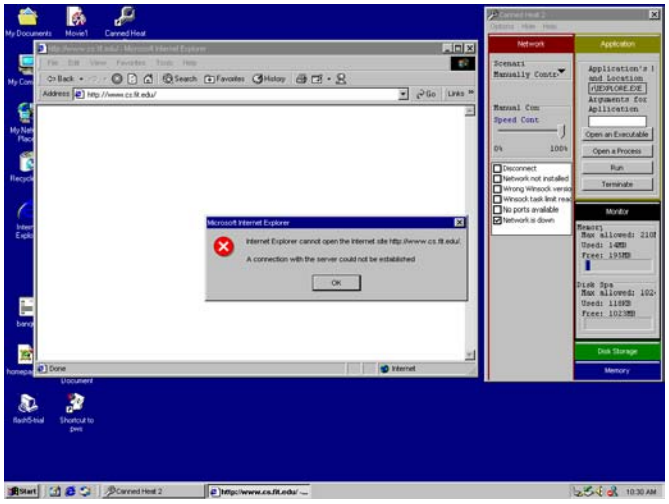
Microsoft Internet Explorer

Internet Explorer could not open the search page.

Step 4. Inject faults at runtime during network use.

The last tests to perform consist of injecting faults at runtime. The previous step demonstrated the use of the random failures scenarios. What we want to accomplish here is to study the behavior of the application when inserting specific faults.

Canned HEAT allows for inserting a number of faults including the "network is down" fault. When this fault is inserted, we can watch Internet Explorer's reaction to a network that has become unresponsive.



Microsoft Internet Explorer

Internet Explorer cannot open the Internet site http://www.cs.fit.edu/.

A connection with the server could not be established.

3.2 Systematic Call-based Fault Injection

Canned HEAT is an easy to use tool to inject course-grained faults into an application's environment at runtime. It is not suitable for use when a more fine-grained, surgical approach is needed.

Canned HEAT works by failing sets of API calls either all of the time, most of the time or some of the time. However, the tester is given no ability to be more choosey than Canned HEAT's interface will allow. Sometimes, testers may want to fail a specific call only once. Or they may want to fail a call only in very specific contexts. In other words, they need a tool to observe APIs being called and have the ability to intervene on a call-by-call basis. This type of fault injection is called *observe-and-fail*.

There are many companies that have in house tools to do such fault injection and they seldom release their tools to the general market. I will attempt to explain how to use these tools using a prototype we have developed at Florida Tech called Holodeck. Any *Star Trek*® fan will immediately recognize the Holodeck as the virtual reality grid where holograms are indistinguishable from real people. Holodeck is our code name for a tool that makes faked software environments indistinguishable from real environments from the software's point of view.

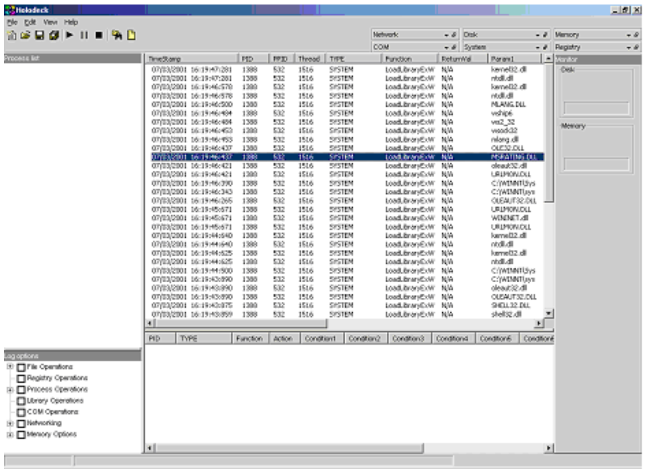
Here's how it works:

Similar to Canned HEAT, Holodeck is able to intercept API calls. Holodeck logs these calls so that testers can observe an application's activity and decide where to inject faults. Holodeck is equipped with filters that allow testers to narrow their search to very specific types of APIs.

Consider the following example (which represents a nice security exploit against the world's favorite web browser).

We begin the example by first using the target application and observing the system calls it makes (using Holodeck to view them). Most of these calls are mundane from a testing point of view but some are not and these can alert astute testers to possible attacks.

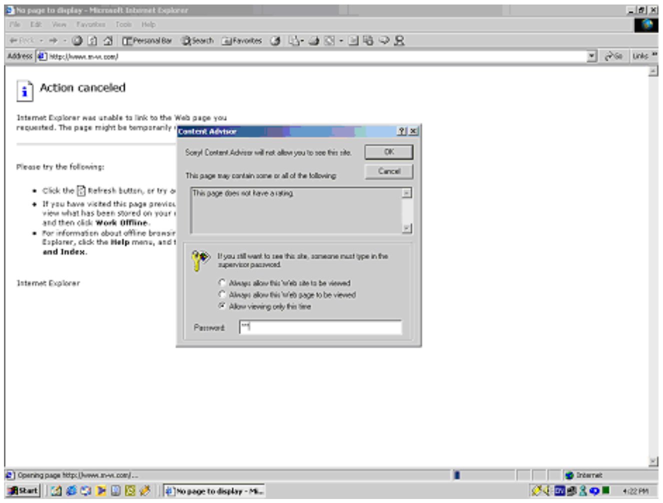
One such call is `LoadLibraryExW`. This call causes external code libraries to be loaded for use. One particularly suspicious such library is MSRATING.DLL as shown below.



```
TimeStamp                PID   PPID  Thread  TYPE     Function        ReturnVal  Param1
07/03/2001 16:19:47:281  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        kernel32.dll
07/03/2001 16:19:47:281  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        ntdll.dll
07/03/2001 16:19:46:578  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        kernel32.dll
07/03/2001 16:19:46:578  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        ntdll.dll
07/03/2001 16:19:46:500  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        MLANG.DLL
07/03/2001 16:19:46:484  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        wshipé
07/03/2001 16:19:46:484  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        ws2_32
07/03/2001 16:19:46:453  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        wsock32
07/03/2001 16:19:46:453  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        mlang.dll
07/03/2001 16:19:46:437  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        OLE32.DLL
07/03/2001 16:19:46:437  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        MSRATING.DLL
07/03/2001 16:19:46:421  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        oleaut32.dll
07/03/2001 16:19:46:421  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        URLMON.DLL
07/03/2001 16:19:46:390  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        C:\WINNT\sys
07/03/2001 16:19:46:343  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        C:\WINNT\sys
07/03/2001 16:19:46:265  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        OLEAUT32.DLL
07/03/2001 16:19:45:671  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        URLMON.DLL
07/03/2001 16:19:45:671  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        WININET.dll
07/03/2001 16:19:45:671  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        URLMON.DLL
07/03/2001 16:19:44:640  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        kernel32.dll
07/03/2001 16:19:44:640  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        ntdll.dll
07/03/2001 16:19:44:625  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        kernel32.dll
07/03/2001 16:19:44:625  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        ntdll.dll
07/03/2001 16:19:44:500  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        C:\WINNT\sys
07/03/2001 16:19:43:890  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        C:\WINNT\sys
07/03/2001 16:19:43:890  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        oleaut32.dll
07/03/2001 16:19:43:890  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        OLEAUT32.DLL
07/03/2001 16:19:43:875  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        SHELL32.DLL
07/03/2001 16:19:43:859  1388  532   1516    SYSTEM   LoadLibraryExW  N/A        shell32.dll
```

As a tester we are now alerted to the fact that this DLL is providing services to our application under test.

The desired behavior of the browser's rating system is to allow, say, a parent, to set up passwords for sites so that, say, their children cannot access them. When the browser is pointed to such a site, it will prompt for the password as shown below.


No page to display - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Personal Bar Search Favorites
Address
OK
Cancel
Password

Action canceled

Internet Explorer was unable to link to the Web page you requested. The page might be temporarily

Please try the following:

- Click the Refresh button, or try
- If you have visited this page previo view what has been stored on your and then click **Work Offline**.
- For information about offline browsi Explorer, click the **Help** menu, and **and Index**.

Internet Explorer

Content Advisor

Sorry! Content Advisor will not allow you to see this site.

This page may contain some or all of the following:

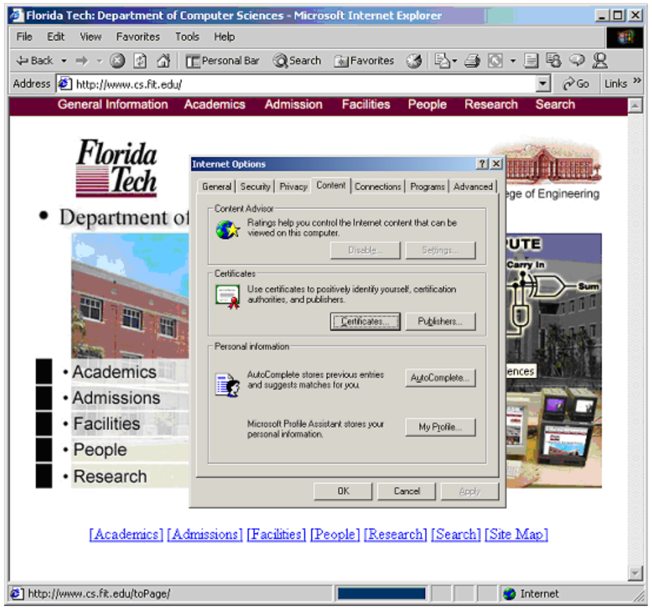
This page does not have a rating.

If you still want to see this site, someone must type in the supervisor password.

( ) Always allow this Web site to be viewed
( ) Always allow this Web page to be viewed
(✓) Allow viewing only this time

Now that the target system call has been identified, we can use Holodeck to inject a fault in the same manner as we used Canned HEAT. In this case, we will simply return a value indicating that the file MSRATING.DLL cannot be opened.

But failing the call to `LoadLibraryExW` causes the feature to be disabled, allowing anyone to surf anywhere they want. Note in the screen shot below, the blocked web site loads and the rating options is unavailable, as indicated by the inaccessibility of its icons.


Florida Tech: Department of Computer Sciences - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://www.cs.fit.edu/
Go Links
General Information Academics Admission Facilities People Research Search
Florida Tech
Department of
College of Engineering
Academics
Admissions
Facilities
People
Research
[Academics] [Admissions] [Facilities] [People] [Research] [Search] [Site Map]
http://www.cs.fit.edu/toPage/
Internet

Internet Options

General | Security | Privacy | Content | Connections | Programs | Advanced

Content Advisor
Ratings help you control the Internet content that can be viewed on this computer.
Disable... Settings...

Certificates
Use certificates to positively identify yourself, certification authorities, and publishers.
Certificates... Publishers...

Personal information
AutoComplete stores previous entries and suggests matches for you. AutoComplete...
Microsoft Profile Assistant stores your personal information. My Profile...

OK Cancel Apply

4. CONCLUSIONS

Triggering exceptions can be very difficult at runtime. Creating scenarios that cause exception handlers to execute often involves a faulty environment that is not easy to stage in a laboratory. Thus, software is released without ever executing some exceptions or error handling code. Since user environments represent more diverse usage than is easily reproduced in testing labs, these exceptions are more likely in the field. This predicament is risky for software publishers who must release untested exception handlers, particularly publishers who release mission or safety critical applications.

Runtime software fault injection allows faulty environments to be simulated in the testing laboratory. Performed judiciously, software testers can increase coverage of error handling code and gain more confidence in their software's ability to perform robustly in an unstable environment.

The tool and methodology presented in this paper allow runtime fault injection to be performed without access to or modification of source code. By exposing system interfaces to interrogation, testers can reason about behaviors that may lead to exception handlers being executed. By modifying system-call return values and error codes dynamically, faults can be simulated so that the exact environment fault is presented to the application under test in a realistic manner. This mechanism is completely general, allowing most any type of stressed environment to be accurately simulated in a laboratory environment. As a result, the benefits can range from increased code coverage to a higher degree of confidence in the robustness of the application when it is deployed.

5. ACKNOWLEDGMENTS

This work was supported in part by separate grants from Microsoft Research and Rational Software Corporation. We thank the remaining members of the HEAT and Canned HEAT development teams, which include Rahul Chaturvedi, Andres De Vivanco, Aditya Kakrania, Terry Lentz and John Brown. In addition, many thanks go to the testers at Microsoft and Rational for their insights into useful ways to fault inject. Special appreciation goes to Harry Robinson of Microsoft and Sam Guckenheimer of Rational for their input into Canned HEAT's user interface and fault selection methods.

6. REFERENCES

[1] Agrawal, H., et al, Design of mutant operators for the C programming language, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, (March 1989).

[2] Bowser, J., Reference manual for Ada mutant operators, Technical Report GIT-SERC-88/02,

Department of Computer Science, Georgia Institute of Technology, Atlanta, (February 1988).

[3] Friedman, M. and Voas, J., Software assessment: reliability, safety, and testability, Wiley, (1995).

[4] Ghosh, A. and Schmid, M., An approach to testing COTS software for robustness to operating system exceptions and errors. In Proceedings of 10th Int'l Symposium on. Software Reliability Eng., (Los Alamitos, CA, 1999) IEEE Computer Society Press, 166-174.

[5] Houlihan, P. Targeted software fault insertion, Proceedings of STAR EAST 2001 (Software Testing Analysis and Review), (Orlando FL, 2001), Software Quality Engineering.

[6] King, K. and Offut, A.J. A Fortran language system for mutation-based software testing, Software Practice and Experience, 21 7, (July 1991), 685-718.

[7] Richter, J. Programming applications for Microsoft windows, Microsoft Press, (1997).

[8] Voas, J. and McGraw, G. Software fault injection: Inoculating programs against errors, Wiley, NY, (1998).

[9] Whittaker, J. What is software testing. And why is it so hard. IEEE Software, 17, 1, (2000), 70-79.

[10] Whittaker, J. Software's invisible users. IEEE Software, 18, 3, (2001) 84-88.