Testing with Hostile Data Streams

Alan A. Jorgensen
Computer Science Department
Florida Institute of Technology
150 W. University Blvd.
Melbourne, Florida 32901
aj@se.fit.edu

Abstract[1]

This note describes a method of testing software for response to malicious data streams. Systems that process data streams obtained from an external source such as the Internet are vulnerable to security issues if malicious data is not processed correctly. This note describes a testing method that creates malicious data streams, applies them to a software application and checks the appropriateness of the application response.

The note begins with a description of the problem: inadequate testing of software response to malicious data streams. I present a method of testing the response to malicious data streams and introduce the concepts of lexical, syntactic and semantic data stream deformation. I provide a description of a system that produces and applies such tests. This description divides the testing system into components and provides some detail about each component. This system applied to Adobe® Acrobat® Reader® version 5.0.1 provides a case study. The study applied 141,306 unique test cases and revealed 11 distinct indications of buffer overrun , numerous program lock-ups, and four steganographic possibilities.

Research is on-going in the following areas: generalized buffer overrun exploitation, maliciously testing protocols and testing with encoded or encrypted data streams.

Keywords: software testing, random testing, Internet security, Adobe® Acrobat Reader®, buffer overrun, buffer overflow, steganography

Introduction –

Security and Data Transmission

The recent concern over the possibility that system security might be broached by means of a data file [1], such as an image file, is not without merit. Though the example cited failed to prescribe a means of invading a system that had not been breached previously, the effort, once again, conveyed the possibility of steganographically conveying hostile information by means of an otherwise useful and harmless data file. Had that example also contained malformed data such that the processing software had failed to properly constrain a buffer range, a buffer overflow could have occurred with the attendant security risk. The hostile data in the stream could well have been hostile code to be executed as a result of a loss of program control initiated by the buffer overrun.

 Funding for this research has been provided by the U.S. Air Force Grant # F49620-01-1-0294, James A. Whittaker, Principal Investigator.

Failure to Constrain

Whittaker and Jorgensen document the failure of developers to properly constrain the software activities input, output, storage and computation [2, 3]. The failure of software testers to test for these failures to constrain has resulted in a plethora of security breaches initiated by buffer overruns [4]. (Search the CERT® website for "Buffer". The terms "buffer overrun" and "buffer overflow" are used interchangeably.)

This paper presents a method for testing a software application's response to corrupt and possibly hostile data streams.

Starting with a general description of the system, this paper presents a system block diagram and some details of the function of each component of the system. As a case study, that system was applied to Adobe® Acrobat® Reader®. I report the results of that study. There are several potential avenues of further research and some of them are presented.

The literature describes analytical reasons for not performing random testing because of the theoretically superior value of partition testing [5]. These theories generally assume the availability of partition knowledge and do not address general black-box testing issues and techniques [6].

Voas, et al, describes methods of corrupting program code at several levels during software fault injection such as randomly selecting the type of deformation or the deformation values themselves, [7].

Kaner mentions the need for a significantly larger volume of testing when using random parameter selection [8].

Hamlet notes the general difficulty of using random testing because of lack of oracles [9].

Data Stream Processing Constraint Failures

Buffer Overruns

A program storing data outside of the area reserved for that data creates a buffer overrun. Typically this involves storing a sequence of data greater in length than the storage area (buffer) reserved for that data. Storing data in an inappropriate place usually causes the software to enter states unanticipated by the developer and consequently the behavior of software after an arbitrary buffer overrun is unpredictable.

And steganography

Steganography is the embedding of a hidden message within another message. In the context of data transmitted over the Internet, data included within that transmission that serves a purpose other than the original purpose of the data transmission is steganographic data. An example of this kind of data would be

hostile code embedded in what would otherwise be informational (inactive) data.

Security Implications

Buffer overruns are a major source of security breaches for users (and providers) of the Internet [4]. A typical breach involves sending a carefully crafted overlong data string such that program control is appropriated and redirected to the data string itself. The data string is crafted to contain hostile code that performs undesirable actions such as, in the case of a computer virus, retransmitting the hostile data stream to other computers. Once hostile code has been executed, a large variety of insidious behaviors may take place.

Historically, buffer overruns have been located by examination of source code.

This class of security failure can be avoided by the identification and elimination of the defective code that performs buffer overruns.

Constraint Testing Technique

Though code inspection is a useful and cost effective way to locate buffer overrun coding defects, in today's development environment of large, complex programs and libraries, the source code may not be available for review. This paper presents a "black box" testing method for identifying buffer overruns. This testing technique is to apply randomly deformed data streams to the application under test. This technique also provides a broader testing capability, however, and includes the ability to detect steganographic possibilities (places in data streams where information can be hidden without detection by the application processing that data stream).

Random data stream deformation

Random data stream deformation is the process of taking a valid data stream, deforming that data stream in a manner such that the data stream is no longer valid. I define three (3) types of file deformation: lexical, syntactic, and semantic. These three categories generally overlap but I define them loosely as:

Lexical – Changing a valid lexical element to an invalid lexical element. A lexical deformation could be replacing a printable character with a non-printable character. In practice, the method is extended to include any character replacement.

Syntactic – Changing valid syntactic elements to lexically correct but invalid syntactic elements. An example would be to replace a left parenthesis with a space character. In practice, the definition is extended to include long string insertions.

Semantic – Changing valid semantic elements to syntactically correct but semantically invalid elements. For example, changing the representation of a number to the representation of a different number that is invalid in that context would constitute a semantic deformation. Another would be changing a defined identifier to an undefined identifier.

Malformed Data Stream Testing System

I have developed a system for testing applications with randomly malformed data streams.

The System

Figure 1 is a block diagram of the hostile data stream test system.

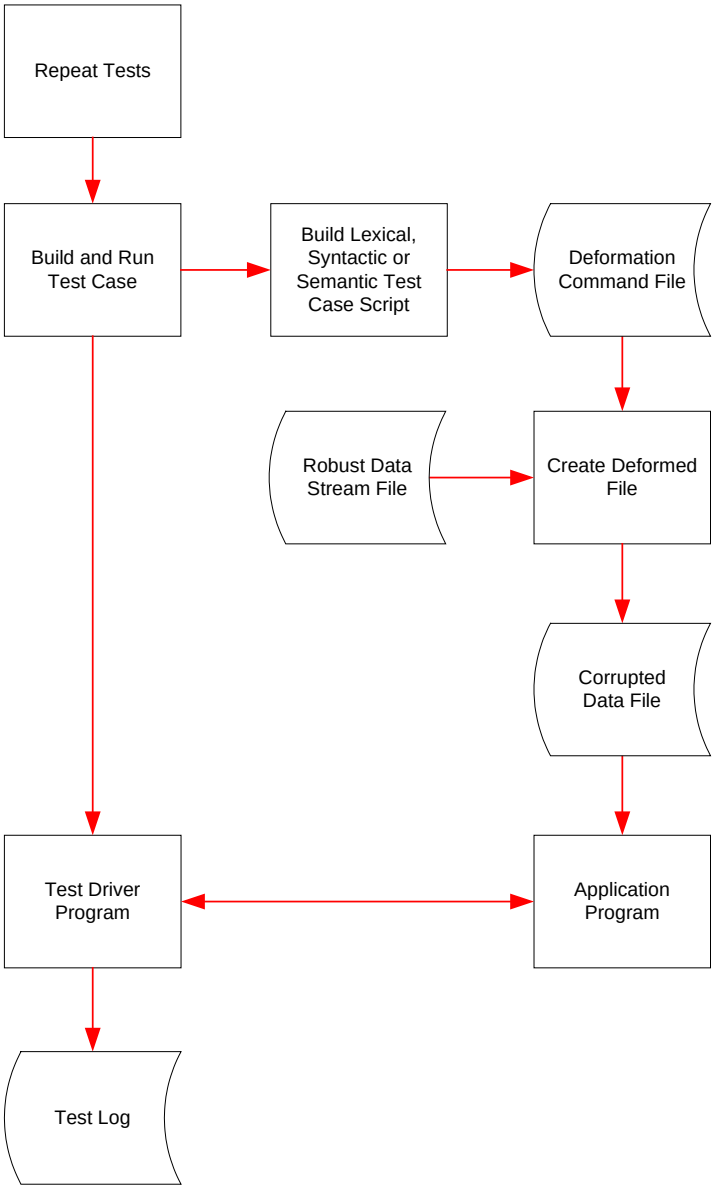

Figure 1 Hostile Data Stream Test System

The Tools

Repeat Test – Continuous repeated testing

The *Repeat Test* function continuously executes the *Build and Run Test Case* function until halted.

Build and Run Test Case – Create a deformed file and run it.

The *Build and Run Test Case* function selects a Lexical, Syntactic, or Semantic deformation algorithm, and executes that function to create a deformed file. The deformation type is selected from a list in a file using the *Select Random Record* function. The frequency distribution of a particular deformation is controlled by the percentage of occurrence of that function name in a file listing the functions. Then *Build and Run Test Case* executes the *Test Driver* function to execute the test case.

Create a Deformed File

The *Create Deformed File* function accepts file deformation commands that produce a copy of a file with specified modifications.

Available commands are:

- Input - Closes prior input stream and opens a new one
- Output - Closes prior output stream and opens new output stream
- Close - Closes the current input and output streams
- Copy - Number of bytes from input to output
- Copy all - Copies the remainder of the input to the output
- Copy to byte number - Copies all input bytes until an input byte position
- Insert - String into output
- Overlay - Inserts string into output and skips same number of input stream bytes
- Fill - Cyclically inserts a specified number bytes from a string
- Overfill - Cyclically inserts a specified number bytes from a string and skips the same number of bytes of input
- Skip - Skips the specified number of bytes of the input.
- System - Performs the specified system level command
- Comment - Permits annotation of a deformation script
- Quit or Exit - Terminates execution the *Create Deformed File* function.

Numbers may be represented either hexadecimally or decimally. A zero (0) precedes the representation of hexadecimal numbers.

Characters or characters in a string are represented by their printable character or by their numeric values (preceded by a '/' character).

There are three deformation functions, *Lexical Deformation (Lex), Syntactic Deformation (Syn),* and *Semantic Deformation (Sem).* Each function creates a file deformation script for the *Create Deformed File* and then executes that function to create a deformed data stream file. Each function passes the specifics of the test case to the *Test Driver* for recording in the *Test Log*.

Lexical Deformation – Create a Lexically deformed file

The *Lexical Deformation (Lex)* function selects a lexical element in the robust source file for replacement with an invalid lexical element. This is accomplished by creating a file of the locations and types of lexical elements using the *General Parse* function. *Lex* selects a particular lexical element from the parsed output using the *Select Random Record* function. With the element location and size obtained from the parse, *Lex* generates the deformation commands to copy the source file up to that element, insert the erroneous lexical element, and copy the rest of the file.

Syntactic Deformaion – Create a Syntactically deformed file

The *Syntactic Deformation (Syn)* function creates a syntactically deformed file. There are many possible techniques for this, but the favored one is to create long string attacks in an attempt to force a buffer overrun. This was accomplished by selecting a random position in the file for the deformation to occur and inserting or overlaying a randomly long sequence of some randomly selected character(s). To accomplish this *Syn* generates a deformation command file that will copy some characters from the source file, fill or overfill a string of characters, and copy the remainder of the source file.

Semantic Deformation – Create a Semantically deformed file

The *Semantic Deformation (Sem)* function creates a semantically deformed file by selecting a specific token from the parsed version of the source file. *Sem* selects a different token of the same type from the same file as a replacement. The *Sem* function creates the script to copy the source until the selected element, overlay with the replacement element, and copy the remainder of the file. This creates a file that is syntactically correct, but (probably) semantically invalid.

General Parse – Parse a file into identified lexical elements

The *General Parse* function is a general-purpose parser that identifies lexical elements of a file. The output consists of one text line per lexical element. Each line contains the character position of the lexical element in the file, a single character code for the element type, the length of the element, and the element itself. Non-displaying characters in the lexical element are presented as numeric values proceeded by a solidus ('/'). This nomenclature is consistent with the representation of non-displaying characters in the *Create Deformed File* function.

Select Random Record – Randomly select a file record

Given the name a of text file, the Select Random Record function outputs one record randomly selected from that file. Each line of the input file has an equal probability of being selected.

Robust Data Stream

A significant element of the system is the use of a robust data stream. A robust data stream is an efficient stream of data that exercises a large number of features of the application under test. An efficient stream should short in length and preferably quickly processed by the application under test. Such a data stream provides a good, quick check of positive application functionality (not exercising error handling or exceptions) and should not produce error messages.

Test Driver -- Run the application with the corrupted data

A *Test Driver* function must be created for each application to be tested. This driver must perform the following functions:

1) Invoke the application
2) Apply the data stream
3) Detect response: The responses of interest here are: Acceptance (Undetected Corruption), Rejection (Corruption detected in some manner), Catastrophic Failure (Crash, General Protection Fault, Failure to Respond, Infinite loop)
4) Detect completion of stream processing
5) Terminate application
6) Record test case information. This includes sufficient information to reproduce the test case and includes recording the response to test case as well as other information such as the date and time of the execution of the test case and the test case duration.

Case Study, Adobe® Acrobat Reader®

Adobe® Acrobat® Reader® is free software provided by Adobe Systems, Incorporated and is available from [10]. Adobe® Acrobat® Reader® (AAR) is trusted software and its use is ubiquitous on the Internet and as such, makes a representative example for the testing concept presented in this note. AAR

processes data in Portable Document File format.

Portable Document Files (PDF)
"Adobe® Portable Document Format (PDF) is the open de facto standard for electronic document distribution worldwide." [11] PDF files permit the accurate communication of documents including lexical and graphical properties.

Robust data file selection
FastIO Systems [12] provides a library of software functions to create PDF files. The function library includes examples of use including programs that produce library test files. This case study included the files atest-1.pdf, atest-2.pdf and arctest.pdf that can be generated using these library test programs. Modifications of the programs that generated these files, testpdf.c and arc.c, generated special versions of these files, atest-1a.pdf, atest-2a.pdf and arctesta.pdf that inhibited some of the file encoding so that data would appear as clear text. Though by no means complete, these files were representative of robust test cases for AAR. Adobe® Acrobat® Reader® version 5.0.1 processes each of these files without error.

Test Driver
The test driver for AAR was a Rational® Visual Test® (VT) script. The Reader driver provided the functions described above. The recording mode of VT provided script elements such as those recognizing and recording the contents of the expected pop-up error windows.

Test environment
Up to five (5) processors ran the test cases for the case study. Each processor ran the same version of the Microsoft® Windows 2000® operating system (Version 5.0, build 2195, service pack 1) and Adobe® Acrobat® Reader® 5.0.1 3/27/2001. Progressive development and improvement of the test system occurred over the period of the case study. Verification of intermediate test results occurred during the testing but duplicate test results are not included in the test results.

Example Test Case
The following test case example illustrates the operation of the system with examples of portions of files produced within the test system.

Robust File – A parsed section of atest-2.pdf
Following is a portion from a test file source after parsing:

```
02000 S 6 stream
02006 B 1 /00A
02007 ? 1 %
02008 ? 1 !
02009 I 16 PS-AdobeFont-1.0
02019 ? 1 :
0201A b 1 /020
0201B I 10 UtopiaBold
02025 B 1 /00A
```

The selection starts from character position 0x02000 in the file. The four fields are 1) location, 2) token type code, 3) token length, and 4) the token itself (non-printing characters shown as hex numbers).

Deformation Command File
Following is a random file deformation command sequence generated by the syntactic deformation function.

```
'  atest-2.pdf
'  Fill from 0201A 0BFF characters of /80

Input atest-2.pdf
Output t.pdf
Copy 0201A
Overfill 0BFF /080
Copy all
Close
System echo log("atest-2.pdf ") > log.inc
System echo log("Fill from 0201A 0BFF characters
of /80") >> log.inc
```

This example has a comment that explains that the source for the test is the file "atest-2.pdf" and describes the nature of the deformation. The deformation commands specifies "atest-2.pdf" as the input file, "t.pdf" as the output file, copies 0x201A characters from the input to the output, inserts 0xBFF bytes of 0x80 into the output, skips 0xBFF bytes of the input, copies the remainder of the input to the output and closes the output file. The properties of the test case are stored in the file "log.inc" for use by the driver to identify this test case in the log.

Deformed File - A parsed section of t.pdf
Here is the corresponding portion of the parsed version of the resulting deformed file:

```
02000 S 6 stream
02006 B 1 /00A
02007 ? 1 %
02008 ? 1 !
02009 I 16 PS-AdobeFont-1.0
02019 ? 1 :
0201A B 17 /080/080/080/080/080/080/080/080/080
/080/080/080/080/080/080/080/080
0202B B 17 /080/080/080/080/080/080/080/080/080
/080/080/080/080/080/080/080/080
```

Test Log – The results of this test case
What follows is the corresponding section of the resulting test log.

```
[Suite Name]         Testing Acrobat 5.0
[Machine]            Machine
[Start Time]         10/20/2001 08:37:13
[End Suite Header]

[Detail 0]           Testing atest-2.pdf
[Detail 0] Fill from 0201A 0BFF characters of /80
[Detail 0] Failed to Detect File Corruption.

[Suite Result]       INCOMPLETE
[Elapsed Time]       23.494
[End Suite]
```

These results are extracted from the test log for the test case corresponding to the example deformation file shown above. The log shows the target of the test, the machine on which it was tested (aliased here for security reasons), the time of the test, the properties of the test case file, the response to the test case, and the elapsed time of the test case (in seconds). Note that "INCOMPLETE" is an indication that the application failed catastrophically with an Application Error.

Application Error – Pop up window

Following is the contents of the Application Error pop-up window as a result of the execution of the example test case:

```
---------------------------
DDE Server Window: AcroRd32.exe - Application
Error
---------------------------
The instruction at "0x0805aaaf" referenced memory
at "0x8080808c". The memory could not be
"written".

Click on OK to terminate the program
Click on CANCEL to debug the program
---------------------------
OK   Cancel
---------------------------
```

The debug information corresponding to this failure indicates that AAR failed during an attempt to store 0x80808080 into location 0x8080808c (0x80808080 + 0xc).

```
eax=0152f098 ebx=80808080 ecx=011a2fc4
edx=0152f6a0 esi=0152daec edi=80808080
eip=0805aaaf esp=0012ea0c ebp=0012ea18
iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023
fs=003b  gs=0000             efl=00000282

0805aaaf 895f0c     mov     [edi+0xc],ebx
```

Note that the program failed attempting to store the contents of the EBX register, (ebx=80808080), into the memory location specified by the contents of the EDI register (edi=80808080) plus 0xC, or location 0x8080808c. Further note that the contents of the EBX and EDI registers are obtained from the data contained in the corruption string. Careful manipulation of the contents of this string might allow specified data to be stored in a specified location.

Test results

The case study included 141,306 uniquely deformed files. There were four (4) recorded categories of failure, Application Failure, infinite loop, failure to respond, and steganographic. Of the 141,306 test cases, there were 426 failures of the severe types. Other than the uniqueness of the test cases, failure-to-respond failures were not classified to determine uniqueness of the failure.

Application Failure

In addition to the catastrophic failure described in the example above, there were ten (10) other similar instances of "Application Failure" with unique symptoms (identified by the unique location of the failing instruction). The "Application Error" pop-up window indicates an instruction reference to an invalid memory location and defines a catastrophic application failure ("crash"). Each of these failures is at least a denial of service and is generally considered a security vulnerability.

Infinite Loop

In other failure instances AAR continued to run but never ran to completion. Windows 2000® Task Manager indicated that AAR was still responding but it was no longer possible to communicate with the application. This type of failure is a denial of service.

Failure to Respond

Some instances of failure occurred when AAR stalled and would not respond to external stimulus. Windows 2000® Task Manager indicated that AAR was not responding. This type of failure is a denial of service.

Steganographic

A common failure resulting from the Lexical tests was that AAR did not detect the file deformation. After elimination of those cases where detection would not be expected, there still remained a set of deformations that could have been detected but were not. These cases fell into areas that apparently were not parsed or parsing was deferred. Each of the following cases presents a steganographical opportunity:

- After header and before first "object"
- Comments
- "Document Property" objects (Parsing is deferred).
- After the End of File indicator ("%%EOF")

Future Work

Generalization of Buffer Overrun Exploitation

There are two schools of thoughts about buffer overruns. The conservative view is that buffer overruns are a security risk. Another view is that, in general, a buffer overrun poses little more risk than that of denial of service. The creation of computer viruses is a highly specialized art form. Is it possible to create a general procedure for developing seriously exploitive attacks from buffer overruns in general?

Malicious Protocol Testing

The case study provided in this note concerns only a single direction of data flow. It seems likely that in bidirectional data transfers (protocols) failures might be exposed at any point in the data exchange. Can the technique described here be applied to applications, such as web page servers, that utilize complex protocols?

Encoded or Encrypted Data streams

Though portions of the case study described in this note included data streams with encoded data, the code actually tested was, in most cases, the decoding software rather than the affect of data that was malformed prior to encoding. Is is possible to extend this technique to create corrupt data streams prior to encoding (or encryption) such that the application functionality that processes the decoded (or decrypted) data might uncover buffer overruns?

References

[1] Costello, Sam (2002): McAfee: New virus is first to infect image files, ComputerWorld (Online publication) June 13, 2002.

http://www.computerworld.com/securitytopics/security/story/0,10801,71968,00.html

(Also see: Costello, Sam (2002): Users question JPEG virus; McAfee stands firm, ComputerWorld, June 24, 2002)

http://computerworld.com/securitytopics/security/story/0,10801,72220,00.html

[2] Whittaker, James A. and Alan A. Jorgensen (1999): Why Software Fails. ACM Software Engineering Notes, July 1999.

[3] Jorgensen, Alan A. (1999): *Software Design Based on Operational Modes*, Ph.D. dissertation, Florida Institute of Technology, 1999.

[4] Software Engineering Institute (2002): CERT Coordination Center, http://www.cert.org Carnegie Mellon University, 2002.

[5] Gutjahr, Walter J. (1999) Partition Testing Versus Random Testing: the Influence of Uncertainty. In IEEE Transactions on Software Engineering, vol. 25, 1999, pp. 661 - 674.

[6] Duran, J. W. and S.C. Ntafos (1984): An evaluation of random testing, IEEE Trans. on Software Engineering, SE-10(7) pp. 438-444, July 1984.

[7] Voas, Jeffrey M. and Gary McGraw (1998): *Software Fault Injection, Inoculating Programs Against Errors*, Wiley Computer Publishing, 1998.

[8] Kaner, Cem (2000) Architectures of Test Automation: Alternatives to GUI Regression Testing, Proceedings, International Conference on Software Test, Analysis and Review, STARWEST 2000, (http://www.kaner.com/pdfs/slides/star2000.pdf)

[9] Hamlet, D. (1994) Random testing, In *Encyclopedia of Software Engineering*, J. Marciniak, Editor, Wiley, New York, pp. 970-978, 1994. (http://citeseer.nj.nec.com/hamlet94random.html).

[10] Adobe Systems Incorporated (2002): Adobe Acrobat – Download, http://www.adobe.com/products/acrobat/readstep2.html

[11] Adobe Systems Incorporated (2002): Adobe PDF, http://www.adobe.com/products/acrobat/adobepdf.html, 2002. Accessed July 16, 2002.

[12] FastIO Systems (2002): http://www.fastio.com/ Accessed July, 2002, Last updated: April 8, 2002.