Enbug: When Debuggers Go Bad

David Williams-King
Dept. of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB, Canada T2N 1N4
dcwillia@ucalgary.ca

John Aycock
Dept. of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB, Canada T2N 1N4
aycock@ucalgary.ca

Daniel Medeiros Nunes de Castro
Dept. of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB, Canada T2N 1N4
dmncastr@ucalgary.ca

ABSTRACT

We have developed a tool, enbug, that intentionally induces errors into software in a controlled fashion. The robustness of students' code can be challenged by presenting exotic failure scenarios for testing, without Herculean efforts on the part of teaching assistants or instructors. Enbug also has applications in computer security and secure software courses, by being able to inject specific flaws into existing software for students to locate and exploit. The implementation of enbug is an example of tool reuse, through the automated (ab)use of a debugger.

Categories and Subject Descriptors

K.3.1 [**Computers and Education**]: Computer Uses in Education—*Computer-assisted instruction*; K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms

Experimentation, Reliability

Keywords

education, testing, debuggers, aspect-oriented programming

1. INTRODUCTION

It is hard to think of reliable computer systems as a disadvantage. However, teaching budding computer scientists to produce robust code is a challenge in a modern computing environment. Dynamic memory allocation doesn't fail, file I/O never falters, network sockets always return the right number of bytes. Pick a favorite language, it doesn't matter – how many times has code like

```
p = (char *)malloc(123);
*p = 'X';
```

or

```
read(fd, buf, sizeof(buf));
puts(buf);
```

or

```
read(in_socket, buf, 100);
write(out_socket, buf, 100);
```

been turned in by students, unnoticed?[1]

All too often, unless students' code is being thoroughly audited, we assert that flaws like this may not be caught. One could argue that proper testing would reveal the flaws, but in fact there are many types of failure that are hard to produce under normal circumstances; race conditions, for instance, may lie dormant for years in well-tested code. Furthermore, the in-depth system knowledge needed to test certain cases may be lacking in teaching assistants and instructors alike.

The production of robust, well-written code is also not a priority in many cases. Assignments are crafted to provide hands-on reinforcement and learning of conceptual ideas from the classroom. The nature of the solution, elegant code or Rube Goldberg code, jury rigged or robust code, makes no difference so long as it works tolerably well for non-extreme test cases. Mediocre code is someone else's problem.

We temporarily turn now to a seemingly unrelated topic. The second author teaches a course in computer viruses and malicious software to senior undergraduate and graduate students [4]. One of the aspects of this course is that it is hands-on, and students are given assignments in both offense and defense in a secure laboratory environment. Part of the course content deals with exploitable software flaws like buffer overflows; in this sense the content overlaps with courses on secure software (e.g., [14, 16]).

One assignment in the course involves exploiting a software vulnerability to run some shellcode (code sent by an attacker to start a shell that they can issue commands to). Initially, students exploited short programs to which they had the source code. This is pedagogically sound in that the basic concepts are taught, but we wanted to extend this assignment to bigger, more realistic programs. While large programs do tend to naturally exhibit exploitable flaws, there is one problem: how will an instructor know which programs are exploitable, or where?

[1]For those unfamiliar with C: these examples dynamically allocate some memory (omitting the failure check) and write to it; read some NUL-terminated data (forgetting to check for failure and that a NUL was read); read and write 100 bytes (not checking for failure or that 100 bytes were read).

One approach is to simply abdicate that responsibility and see what students find on their own [10]; this is commendable in that the students' results give back to the community by finding bugs, but frustrating for students and insoluble in some instances. Another approach is for the instructor to add bugs into the source code themselves, but large programs can be nontrivial to build and rely on multitudes of libraries and other software packages. Ideally there would be some way to add bugs into software without recompilation.

The common theme in both these scenarios is that software failures don't occur at convenient times and in convenient places. We have developed a tool to address this problem: enbug. Enbug can be used to cause controlled failures and induce bugs in bug-free software, without substantial effort on the part of an instructor.

The remainder of this paper is structured as follows. Section 2 describes enbug from a usage standpoint; Section 3 describes enbug's implementation. Examples of enbug usage, along with our experiences using the tool so far, are in Section 4. Related work is in Section 5, and we conclude in Section 6.

2. ENBUG

Enbug can be run on an arbitrary program by specifying its process ID (if the program is already running) as

```
enbug -f spec.enbug -p processid
```

or by naming the program to run along with its arguments:

```
enbug -f spec.enbug program arg1 arg2 ... argn
```

The parameter "spec.enbug" is an enbug specification file describing where and how enbug should alter the program as it executes. Enbug, in essence, is implementing an aspect-oriented programming (AOP) system [9], and the language used in the specification file is based on AspectJ [1].

An enbug specification file contains zero or more aspects, which act as namespaces. The shortest nonempty, noninteresting specification is one defining an aspect that has no further instructions in it:

```
aspect foo {
    // do nothing
}
```

Inside an aspect, two things can be defined: pointcuts and advice. Intuitively, pointcuts specify where enbug should pause the program's execution, and advice tells enbug what to do when a pointcut is reached.

The general form of a pointcut is

```
pointcut name: expression;
```

where the name identifies the pointcut so that it may be referenced by other pointcuts. It is also used to select which advice to execute. The expression may be one of three types of built-in pointcut, the name of a user-defined pointcut, or the logical combination (using Boolean and, or, and not, plus grouping with parentheses) of any of these. The three built-in pointcuts are:

`execution(`*pattern*`)`
: The execution of all functions matching the pattern, which may consist of characters and the Kleene star, e.g., `foo`, `bar*`, `*baz*`.

`atline(`*filename*`,`*line*`)`
: Execution that reaches the given filename and line number.

`caller(`*pattern*`,`*framenumber*`)`
: Adds context, by selecting those locations where the function specified by the pattern occurs at a given frame number – frame number 1 is the caller, frame 2 is the caller's caller, and so on.

Combining built-in pointcuts together, for example, an aspect "log" could tell enbug to watch a program's execution for the function "main" (via the "start" pointcut) or the execution of the "openfile" function:

```
aspect foo {
    pointcut start(): execution(* main(..));
    pointcut log(): start() ||
                    execution(* openfile(..));
}
```

A pointcut has been reached. Now what? The answer is specified using advice. The general form of advice is

```
advice location: pointcut {
    debugger code
}
```

The location indicates when the debugger code will run relative to the named pointcut. The locations "before" and "after" run before and after the pointcut, as is obvious; "around" is meant to be used when the advice replaces a function call completely.[2]

In a foreshadowing of enbug's implementation, the debugger code is a sequence of commands for the GNU GDB debugger [6],[3] with two enhancements. The ith parameter to a function can be referenced as `@param(`i`)`, and `@proceed()` can be used in "around" advice to replace a function call.

3. ENBUG IMPLEMENTATION

Enbug is implemented in C++, with occasional detours into Lex and Yacc.[4] It runs on both Linux and FreeBSD. The key to enbug's functionality, though, is its use of the GNU GDB debugger.

Besides the normal command-line user interface, GDB has alternative "command interpreters" that may be used to control its operation. Their format allows easy machine parsing, and permit GDB to be used as the engine behind a graphical debugger, for example. Enbug uses the GDB/MI interface [7]. For illustration, Figure 1 is the GDB/MI interaction (edited for brevity) that sets a breakpoint at line 3 of a program, then runs it; input to GDB is highlighted, and indented lines are wrapped for readability.

Enbug uses GDB breakpoints to implement pointcuts, with the exception of the "caller" built-in pointcut. For caller, it would be infeasible to set a breakpoint at every location which had a particular calling routine, for example (in fact, the general problem is statically undecidable, because it requires knowing which functions will call each other prior to run time). Instead, caller relies on the "atline" or

[2]Although it is currently an alias for "before."

[3]The latest GDB allows Python scripting, so this is not a major limitation.

[4]Strictly speaking, flex and bisonc++.

```
~"GNU gdb Red Hat Linux (6.5-37.el5_2.2rh)\n"
~"Copyright (C) 2006 Free Software Foundation, Inc.\n"
(gdb)
-break-insert 3
^done,bkpt=number="1",type="breakpoint",disp="keep",enabled="y",
    addr="0x0804836c",func="main",file="x.c",fullname="/home/user/foo.c",line="3",times="0"
(gdb)
-exec-run
^running
(gdb)
*stopped,reason="breakpoint-hit",bkptno="1",thread-id="0",frame=addr="0x0804836c",
    func="main",args=[],file="x.c",fullname="/home/user/foo.c",line="3"
(gdb)
```

Figure 1: Example GDB/MI interaction

"execution" built-ins to set breakpoints, and as such must be specified in conjunction with one of them.

The location of advice reflects the implementation slightly too. Advice specified as "before" and "around" executes on the first line of a function (when a breakpoint is set with an execution pointcut), but "after" advice executes on the next line of the calling function.

One final complication is that GDB does not stop twice if two breakpoints are set at the same place. Upon reaching a breakpoint, enbug must walk through the pointcuts to see which ones were reached, because two different pointcuts may have resulted in the same breakpoint being set.

4. EXAMPLES AND EXPERIENCE

We have included a number of examples in this section, to illustrate how enbug can be used for causing bugs that are conducive to testing and teaching. For space reasons, we only show the enbug specifications.

The first example causes the malloc dynamic memory allocator to fail (returning 0, or NULL) after the Nth call to it. Malloc was chosen arbitrarily, and this could be done for any library routine. We use the debugger variable "$n," initialized in advice executed just before main, to keep track of the number of calls. The "@proceed(0)" command performs the return with the failure value of 0.

```
aspect mallocfail {
  pointcut main(): execution(* main(..));
  pointcut malloc(): execution(* malloc(..));

  advice before(): main() {
    set $n = 0
    set $N = 2
  }
  advice before(): malloc() {
    if $n > $N
      @proceed(0)
    else
      set $n = $n + 1
    end
  }
}
```

A related example, more specific to malloc, is to have it fail after N bytes have been allocated. We again use debugger variables, adding malloc's parameter (the requested allocation size) to track the amount of memory.

```
aspect mallocfail {
  pointcut main(): execution(* main(..));
  pointcut malloc(): execution(* malloc(..));

  advice before(): main() {
    set $n = 0
    set $N = 256
  }
  advice before(): malloc() {
    if $n > $N
      @proceed(0)
    else
      set $n = $n + @param(1)
    end
  }
}
```

This next example causes fread to return fewer objects than requested. Reading or writing less than requested is a legitimate occurrence in network code, albeit one not often well-handled, making it a prime test case.

```
aspect returnless {
  pointcut fread(): execution(* fread(..));

  advice before(): fread() {
    set @param(3) = @param(3) - 1
  }
}
```

A similar enbug specification could increase the size of a buffer to inject a potentially-exploitable flaw; this would simulate the programmer giving the wrong buffer size in their code. For example, here the length parameter for a call to snprintf is increased by 100 bytes:

```
aspect incbuffer {
  pointcut snprintf(): execution(* *snprintf(..));

  advice before(): snprintf() {
    set @param(2) = @param(2) + 100
  }
}
```

To weaken hardened code, more secure functions can be replaced with less secure ones. This example replaces strlcpy with strncpy, and also shows how to return a value from a previous function call that was performed using the GDB "call" command. (Note that the return value semantics do not match between the two functions, however; this is shown for illustration only.) As the function is being replaced completely, we use "around" advice here.

```
aspect makeinsecure {
  pointcut strlcpy(): execution(* strlcpy(..));

  advice around(): strlcpy() {
    call strncpy( @param(1) , @param(2) , @param(3) )
    @proceed($$0)
  }
}
```

Finally, this last example adds (pseudo-)random noise into input being read. This would be useful to test networking code that should be robust to transmission errors. A breakpoint before main allows initialization: we seed the pseudo-random number generator by calling srandom with the current time, using C library functions.

The pointcut for fgets is specified slightly differently, as "*fgets" as opposed to "fgets." The reason is that fgets is implemented on Linux using a function called "_IO_fgets," and without the extra star the pointcut would not match. In the "incbuffer" aspect example above, "snprintf" was given as "*snprintf" for similar reasons.

Each time fgets, a.k.a. _IO_fgets, is entered, we flip a coin by calling the random function for a single random bit, then we skip to the end of fgets' execution using the GDB finish command. This is not specified as "after" advice, because we need access to fgets' parameters, and that is not possible with "after" because of where the GDB breakpoint is placed.

At the end of fgets' execution, if the random bit was 1, we pick a random position in the string that was read in, and replace that character (effectively indexing into the buffer as an array) with an "X."

```
aspect addnoise {
  pointcut main(): execution(* main(..));
  pointcut fgets(): execution(* *fgets(..));

  advice before(): main() {
    set $t = time(0)
    call srandom($t)
  }
  advice before(): fgets() {
    set $rndbit = random() & 1
    finish
    if $rndbit == 1
      set $len = strlen( @param(1) )
      set $pos = random() % $len
      set @param(1) [$pos] = 'X'
    end
  }
}
```

Our experience with an initial deployment of enbug admittedly did not go perfectly. Requiring debugging information to be in an enbugged program is to be expected, because this is needed for a debugger. However, enbug's functionality was limited without debugging versions of shared libraries as well; this required a separate package installation on Linux.

(More generally, a limitation of our approach is that we are restricted to languages that can be debugged using GDB. We would be unable to easily affect changes in an interpreted Python script, for example.)

We wanted to deploy enbug in a security course where it would inject bugs into network servers. The servers in question were started from inetd, which waits for incoming network connections, then starts the server running. The newly-started server inherits the open network connection on the standard input and output during process creation. Unfortunately, because enbug is communicating with gdb using the standard input and output, enbug did not play nicely with inetd servers. We conjecture that this problem may be worked around.

The other issue had do to with forking processes, a common operation in some server architectures. The desired behavior is for enbug to continue monitoring *both* parent and child processes for pointcuts, but GDB does not support this. On some platforms, GDB will allow both processes to be under its control, but only one may be actively executing [6], which is not appropriate for network servers.

Finally, as would be expected, running any program under a debugger has a negative impact on performance. Like the story of a mirror distracting people from noticing the slow elevator, part of dealing with performance issues is psychological: students would be exploiting "remote" servers, and so therefore some latency would be expected anyway. We hope to report on our experience using enbug with students and its benefits in future work.

5. RELATED WORK

Hunt and Thomas use the term "enbugging" but just as a lead-in to discussing defensive programming techniques. In fact, they go so far as to say 'We rarely put bugs in directly' [8, page 10], the opposite of what our tool does.

Testing of student code has been done for decades, of course, and various techniques are published (e.g., [3, 5, 12]). We are not aware of any prior use of deliberate bug insertion, nor does there seem to be any work on inserting exploitable flaws into programs as we can do with enbug, for educational or other purposes. Locasto [11] relates his experience having students attack their own programs, and a tool like enbug may be used to support that process.

PROSE, an early AOP system, was implemented using the Java VM's debugger interface, JVMDI [13]. JVMDI is not a debugger *per se*, but is an interface that debuggers and similar programs can use [15]. The Axon AOP system from 2003 uses the JVMDI too [2]. No other AOP systems appear to employ software tool reuse by using an existing debugger like enbug does.

6. CONCLUSION

Bugs do not always occur where and when they are helpful, in terms of testing student code and teaching students about security exploits. Our contribution to address this problem is enbug, a tool that can inject exotic bugs into existing programs simply by modifying an specification based on aspect-oriented programming. Enbug's intended target audience is teaching assistants and instructors, and because students do not directly see or use enbug themselves, it may be used at any level of instruction. In terms of implementation, enbug is an example of software tool reuse, resulting in a novel method of implementing aspects. The range of examples enbug is able to handle demonstrates its usefulness, and we speculate that it has other uses beyond adding bugs into code.

7. ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Council of Canada via ISSNet, the Internetworked Systems Security Network. Rob Walker assisted with some AOP aspects.

8. REFERENCES

[1] AspectJ Team. The AspectJ language. `http://www.eclipse.org/aspectj/doc/released/progguide/language.html`, last accessed 12 January 2010.

[2] S. Aussmann and M. Haupt. Axon – dynamic AOP through runtime inspection and monitoring. In *Proceedings of the Workshop on Advancing the State-of-the-Art in Runtime Inspection*, 2003.

[3] J. Aycock. The ART of compiler construction projects. *ACM SIGPLAN Notices*, 38(12):28–32, 2003.

[4] J. Aycock and K. Barker. Viruses 101. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 152–156, 2005.

[5] J. Denzinger and J. Kidney. Evaluating different genetic operators in the testing for unwanted emergent behavior using evolutionary learning of behavior. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 23–29, 2006.

[6] Free Software Foundation. Debugging with GDB, 2010. 9th edition.

[7] Free Software Foundation. The GDB/MI interface. In GDB [6], chapter 27. 9th edition.

[8] A. Hunt and D. Thomas. The art of enbugging. *IEEE Software*, 20(1):10–11, 2003.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[10] J. Leyden. Students find 44 Unix flaws as homework. The Register, 16 December 2004.

[11] M. E. Locasto. Helping students 0wn their own code. *IEEE Security & Privacy*, pages 53–56, May/June 2009.

[12] C. MacNish. Evolutionary programming techniques for testing students' code. In *Proceedings of the Australasian Conference on Computing Education*, pages 170–173, 2000.

[13] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 141–147, 2002.

[14] M. L. Stamat and J. W. Humphries. Training $\neq$ education: Putting secure software engineering back in the classroom. In *Proceedings of the 14th Western Canadian Conference on Computing Education*, pages 116–123, 2009.

[15] Sun Microsystems. Java(tm) virtual machine debug interface reference. `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html`, last accessed 12 January 2010.

[16] J. Walden and C. E. Frank. Secure software engineering teaching modules. In *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development*, pages 19–23, 2006.