Final Year Project Report

Bugs in OpenBSD

Niall O'Higgins

A thesis submitted in part fulfilment of the degree of

BA/BSc (hons) in Computer Science

Supervisor: Dr. Joseph Kiniry

Moderator: Dr. Pavel Gladyshev

UCD School of Computer Science and Informatics
College of Engineering Mathematical and Physical Sciences
University College Dublin

March 23, 2006

Table of Contents

Abstract

This report presents a method for automatically identifying bugs in OpenBSD command-line applications. As OpenBSD is a UNIX-like operating system, this method also applies to general UNIX command-line applications. It is possible to view A UNIX command-line application as a function which transforms various inputs (standard input, some files, command line arguments) into some output (standard output/error, some files, exit code). "Interesting" boundary values are generated and fed to the program, quickly exploring the possible error paths and thus exposing bugs. Additionally, a tool implementing this bug detection is demonstrated and made to automatically populate a flexible Internet-accessible bug database. Furthermore, analysis of a number of bugs found during the course of the project case study is presented. Finally, guidelines are put forward for the API designer, API implementor and API user to avoid bugs in the future.

Acknowledgments

My supervisor, Dr. Joseph Kiniry, for his input and guidance.

Chapter 1: **Introduction**

1.1 Project Specification

OpenBSD [1] is widely regarded as the world's most secure operating system. It represents the combined effort of hundreds of dedicated individuals, most of whom work in their spare time for free, over nearly a decade.

OpenBSD developers try very hard to write code that is correct, a goal which has the convenient side effect of producing code which is also well-designed, maintainable, clean and secure.

The OpenBSD source code has been and continues to be relentlessly trawled for bugs. The cause of a bug could be, for example, due to a misunderstanding of an API, an incorrect assumption on the part of the programmer, or a simple typo. From time to time, a bug is discovered which really represents a whole new class of mistake. Once such a class is discovered, the entire source tree is scoured for further instances of this error — thus one bug fix may lead to many more, often solving problems in other unrelated places before they are even noticed.

Finding, analysing and fixing bugs has been central to the success of the OpenBSD project. However, no formal database is maintained of the bugs encountered during development. Furthermore, no official record is made of the tools and methodologies which help in the process of identifying and fixing them.

This project focuses on (a) identifying and fixing bugs in the OpenBSD source code (b) analysing and classifying these bugs, and making the resulting information made available in a searchable database (c) exploring the development of software engineering tools and processes to aid in this endeavour.

Mandatory

- Evolve a structured and systematic methodology to find bugs in the OpenBSD source tree.
- Design and implement a flexible Internet-accessible database for the storage of bug-data.
- Develop a concise process for analysing and categorising bugs.
- Find, fix, analyse and categorise a quantity of bugs in the OpenBSD source tree.

Discretionary

- Propose guidelines for both the API designer, API implementor and API user to avoid pitfalls in the future.
- Design software tools to aid in the discovery of bug instances.

Exceptional

- Implement software tools which automatically locate instances of a bug class in a source tree.

1.2 Overview

This project documents a journey through a real software engineering landscape. Early on, it was decided to focus on a specific area of the OpenBSD source tree and to use this as a case study for research into bugs and their nature. This case study would provide insight into and real-world examples of how bugs are introduced, how they are found and how they can best be avoided. The chosen component of OpenBSD was OpenCVS [2], with a strong focus on its RCS [3] implementation. The author has for the duration of the final project worked as an integral part of a small and dedicated team within the official OpenBSD project which has produced thousands of lines of correct high-quality source code. This work is freely available and is a part of the main OpenBSD source tree. Further discussion of the case study, its motivations and its results, can be found in later chapters.

In parallel to working on this case study, the author has strived to relate and use those experiences gained from its development to attain the goals of the project. This report attempts to show how he has extracted from his analysis plausible software engineering guidelines aimed at limiting future pitfalls. The ultimate product of this final project is a working software implementation of a method devised as part of the project for finding bugs in UNIX command-line applications. Furthermore the tool integrates with a bug database such that it reports its finding in a human-readable format. Additionally, this tool has found a significant number of real-world bugs in the case study software. These bugs have been categorised along the lines proposed in this report and a selection of them are presented for analysis.

Chapter 2: **Background Research**

The project can be broken down into three distinct areas; the nature of bugs in software and useful API guidelines to help prevent them, the development of a flexible database in which to store information pertaining to these bugs, and finally the creation of an automatic tool to locate instances of bugs. This chapter details existing research into these areas including tools and approaches.

2.1 Bugs and API Guidelines

According to Wikipedia [4], "A software bug is an error, flaw, mistake, failure, or fault in a computer program that prevents it from working as intended, or produces an incorrect result.". At the core of this project is the realisation that bugs are largely the result of programmer error. However, often the programmer is given APIs which are complicated and ambiguous, thus difficult to fully understand. So the programmer is in a position where it is easy for her to introduce potentially serious bugs in her program, unless she understands fully all the nuanced and complicated aspects of an API that may in fact seem simple on the surface.

The OpenBSD project has been very aggressive in identifying and documenting those C language APIs which are complicated, ambiguous and commonly lead to mistakes. In some cases, they developed their own replacement APIs which implement effectively the same functionality as those they replaced but with much less ambiguous and error-prone interfaces. These OpenBSD APIs have been so successful that they have been adopted by other projects, becoming de-facto standards for secure and correct programs. In order to meet the goals of this project to provide guidelines for the API designer, implementor and user, it was decided to examine this work done by the OpenBSD project. From this work on proposing entirely new APIs it is possible to extract details of what exactly it is that makes a given C API bad or dangerous. It is possible to gain insight into questions such as, what are the qualities of an API which lead to frequent misuse?

2.1.1 strlcpy() and strlcat()

Perhaps the most well-known C language API replacement developed by the OpenBSD team have been the *strlcpy()* and *strlcat()* functions. These functions are designed to replace the extremely widely used ANSI C string functions *strcpy()*, *strcat()* and *strncpy()*, *strncat()*. These functions are used to copy and concatenate strings. A comparison of these functions with each other and with their Java equivalent is provided for illustration below.

String copy comparison

String copy in Java

```
String myString = ``hello world'';
```

Assume "buffer" variable had been declared earlier as char buffer[10];

String copy in C using ANSI C *strcpy()*

```
strcpy(buffer, ``hello world'');
```

String copy in C using ANSI C *strncpy()*

```
strncpy(buffer, ``hello world'', sizeof(buffer));
// strncpy() does not guarantee NUL-termination of string, so it must
// be done manually.
buffer[sizeof(buffer) - 1] = '\0';
```

String copy in C using OpenBSD's *strlcpy()*

```
strlcpy(buffer, ``hello world'', sizeof(buffer));
```

String concatenation comparison

String concatenation in Java

```
String mySecondString = myString + ``foo bar'';
```

String concatenation in C using ANSI C *strcat()*

```
strcat(buffer, ``foo bar'');
```

String concatenation in C using ANSI C *strncat()*

```
strncat(buffer, ``foo bar'', sizeof(buffer));
// strncat() does not guarantee NUL-termination of string, so it must
// be done manually.
buffer[sizeof(buffer) - 1] = '\0';
```

String copy in C using OpenBSD's *strlcat()*

```
strlcat(buffer, ``hello world'', sizeof(buffer));
```

The OpenBSD replacement functions address two key issues with the existing ANSI C API. Firstly, *strcpy()* and *strcat()* do not take into consideration the length of the destination buffer. This makes it trivial to introduce dangerous buffer overflow bugs. Such bugs are not only likely to lead to program crash, but also can be used by attackers to make the system run arbitrary code. These two functions have in fact been responsible for some very major and widely publicised flaws, such as a remote root exploit in sendmail 8.7.5 [5]. The functions *strncpy()* and *strncat()* introduce a length parameter which prevents direct buffer overflow. However, these functions do not guarantee to NUL-terminate the resulting string. NUL-termination is used by the C language to determine the endpoint of a string. This endpoint is marked by the control character '\0'. To determine the length of a C string, one calls the function *strlen()* which returns the number of characters counted until the first '\0' is found. For this reason, a memory buffer containing a string without NUL-termination means it is very difficult to know how long the resulting string is. Furthermore, it is impossible to know whether or not these functions have, due to a lack of available space in the destination buffer, been forced to copy a truncated version of the source string.

The string functions developed by the OpenBSD team, on the other hand, deal with these complications in a much more elegant fashion. To avert buffer overflow they have a length parameter. To deal with the NUL-termination issue, they guarantee to NUL-terminate the destination buffer regardless of whether or not the buffer is too small to accommodate the source string. Additionally, unlike the functions *strncpy()* and *strncat()*, they do not zero-fill the destination buffer and so in cases where the destination buffer is much larger than the source buffer there is a significant performance improvement. Finally, they return the number of bytes they have copied such that it is trivial to detect and handle truncation. Further discussion of the *strlcpy()* and *strlcat()* functions can be found in *de Raadt and Miller* [6].

2.1.2 strtonum()

A more recent C language API replacement developed by the OpenBSD team is the *strtonum()* function. This is designed to replace the ANSI C family of string-to-number conversion functions, namely *atof()*, *atoi()*, *atoi()*, *strtod()* and *strtol()*. These functions are used to convert string values to numerical values. A comparison of these functions with each other and with their Java equivalent is provided for illustration below.

String to number conversion comparison

String to integer conversion in Java

int i = java.lang.Integer.parseInt("1024");

String to integer conversion in C using ANSI C *atoi()*

```
int i = atoi(‘‘1024’’)
```

String to integer conversion in C using OpenBSD's *strtonum()*

```
char buffer[10] = ‘‘1024’’;
const char *errorstring;
int i;

i = strtonum(buffer, INT_MIN, INT_MAX, &errorstring);
if (errorstring != NULL)
        // error, message pointed to by errorstring
```

The OpenBSD replacement *strtonum()* is significantly better than the ANSI C function *atoi()*. This is because *atoi()* is unable to provide indication of a host of important conditions such as overflow, underflow or invalid input. *strtonum()* on the other hand, offers a simple mechanism by which the programmer can set minimum and maximum values for inputs. This can be extremely useful - for example it could be used to prevent inputs greater than 2^{31} overflowing a signed integer variable, or negative values affecting an unsigned integer variable. Furthermore it becomes trivial to test for invalid input, as the `errorstring` pointer is set to a message describing the error in cases of failure.

2.1.3 setresuid()

While this function was not actually developed by the OpenBSD team, its use is greatly encouraged as an alternative to the older, more ambiguous *setuid()*, *seteuid()*, *setruid()* and *setreuid()* API functions. *setresuid()* first appeared in HP-UX and is part of the POSIX specification. These functions all provide ways to set various user identifier (UID) numbers of the current process. UIDs lie at the core of the UNIX security model because privileges are granted on the basis of these identifiers.

The problem with the older *setuid()* family of functions is that they were designed for an era when there were only two types of UID under UNIX; real and effective. At that time, the situation was relatively simple. The *setuid()* and *setreuid()* functions set both effective and real UIDs. The functions *seteuid()* and *setruid()* could be used to set the effective or real UIDs separately. With the advent of POSIX UNIX systems, a third type of UID was introduced, the "saved" UID. Hence the old *setuid()* function was modified to change all three UIDs but worked for non-root users only if the new UID matched the real UID. The old *seteuid()* function was modified to change the effective UID if the new UID matched the saved or real UID. However, at the same time the functions *setruid()* and *setreuid()* were officially deprecated according to the POSIX specification. The *setreuid()* function continued to exist on many UNIX systems despite this fact while *setruid()* disappeared almost completely.

This situation, with the semantics of these important functions changing and some of them disappearing or being deprecated, led to considerable ambiguity. As such, programmers using the functions with the intention of switching a certain UID may in fact be switching the wrong UID without knowing it. Since UIDs are crucial in implementing the "principal of least privilege" under UNIX such mistakes can lead to programs running with increased privileges. This could in turn lead to security vulnerabilities in the programs.

The *setresuid()* function does away with much of the ambiguity of the older functions. This is because with a single call it clearly sets the real, effective and saved UIDs of the current process. Therefore, usage of this function in place of multiple usages of the older, more ambiguous functions leads to cleaner and less error-prone code.

2.2 Bug database

Bug tracking systems are central to practically every large-scale software development project. With bug tracking systems, bugs can be reported, managed and dealt with in an effective manner. A record is kept of old bugs which have been fixed, along with relevant information possibly including the source code patch. This kind of system is extremely valuable to developers and users alike. Through a bug tracking system, they can check to see if the bug has already been reported, if there is a fix pending or whether they can add more information.

Research into existing bug tracking systems was conducted to ascertain whether it is best to start entirely from scratch or modify an existing system. The three most prominent Open Source bug tracker suites were considered: Bug-Zilla [7], G-Forge [8]/Source-Forge and GNU GNATS [9]. The software was examined along these axes of analysis; search functionality, database extensibility, interface.

The database itself must be flexible enough to store additional data. Bugs need to be associated not just with, say, a particular subsystem but with a broader notion of class and category. Additionally, the interface to the system is of great importance. Use of the term "user interface" is specifically avoided here to make it clear that there are non-human (i.e. software) agents interfacing with the system. At the very least, the database should have a reasonably simple GUI but how to best provide that over the Internet is a non-trivial question.

The solution *du-jour* is, of course, to make it a web-based application, however this is not entirely satisfactory for the following reasons: the stateless nature of the Web makes implementing web-applications cumbersome at best, users are forced to interact with the system via a web browser, HTTP CGI makes it difficult for software agents to interface with the system cleanly without lots of extra work and overhead (SOAP, XML-RPC, etc). Ideally, the bug database would be accessible to the user as a web-application *in addition to* other possible interface methods.

One final consideration to the research in this area was to try to find something which is compatible with the philosophy and work flow of the OpenBSD project. It is important to produce something which will actually be of practical value to members of the OpenBSD community, and perhaps even the wider software development community. Indeed, this non-technical aspect is actually the most critical factor in choosing the bug tracker system. Without general acceptance and usage, the system will have little chance of gaining the critical mass required for it to become a very valuable resource.

2.3 Automatically finding bugs

The exceptional goal of the project is to produce a tool capable of automatically locating bugs in a source tree. This is a very diverse field and there are many possible approaches. In order to develop a method it was necessary to conduct research into existing work in this domain. Background research into the area of automatically locating bugs can be divided into two areas; tools and mechanisms for this which are already included with OpenBSD and tools and mechanisms which are not included with OpenBSD. The former are examined first followed by the latter.

2.3.1 GNU Compiler Collection

There are a number of existing tools and mechanisms included with OpenBSD which can automatically detect bugs. First and foremost, the *GNU Compiler Collection* (GCC) itself has some detection features to warn against the most blatant of mistakes.

Consider this trivial program, containing an obvious stack overflow[1]:

```
#include <string.h>
#include <stdio.h>

int
main(int argc, char **argv)
{
        char buffer[2];
```

[1]To paraphrase Wikipedia, a stack overflow is an anomalous condition where a program somehow writes data beyond the allocated end of a memory buffer stored on the program stack.

```
        strcpy(buffer, "far far too long for the buffer");
        printf(buffer);

        return (0);
}
```

When GCC compiles this program, it will output *"warning: strcpy() is almost always misused, please use strlcpy()"*. This informs the developer that the library he is using is unsafe [2], and prompts him to consider the safe alternative *strlcpy()*. GCC on OpenBSD will issue warnings like this very aggressively with the intention of scaring developers away from any unsafe library functions.

GCC has a very useful command line flag "-Wall" which enables all warnings. This is very good for catching errors and so the OpenBSD project enables it across practically all of its source tree. Consider the following trivial program:

```
#include <stdlib.h>

int
main(int argc, char **argv)
{
        printf(``hello world!\n'');
        exit(0);
}
```

When compiled on OpenBSD without the -Wall flag, no warnings will be printed and the program will function normally. However, if it is compiled with the -Wall flag, the following warning will be printed (the source code is in a file called test.c):

```
test.c: In function 'main':
test.c:6: warning: implicit declaration of function 'printf'
```

This tells the developer that the function *printf()* has no prototype defined anywhere. In fact, according to the ANSI C standard, the *printf()* function is defined in the header *stdio.h* which should be included. In effect, this warning tells the developer, correctly, that the source code is not including a required header file. Although the program may function properly under OpenBSD, on other systems that is not guaranteed. Thus by seeing that GCC warning and fixing the root cause the portability and correctness of the program has been increased.

2.3.2 IBM ProPolice

Furthermore, when compiled on OpenBSD, this program will abort execution, dump core and attempt to report the problem in the system logs with a critical-priority message "stack overflow in function XXX". This is due to the inclusion of IBM's "ProPolice"[10] stack protection extension in OpenBSD's version of GCC, which is enabled by default [11].

[2]Typically, unsafe functions are those which are unable to guarantee they will not write beyond the allocated end of a memory buffer.

2.3.3 mmap() based malloc()

Another useful mechanism included with OpenBSD is the random heap memory allocator. This is a re-written version of the *malloc()* C library function. The *malloc()* function is used to allocate dynamic memory (from the heap) to a program. It is extremely widely used in practically all non-trivial programs written in the C language. Traditionally, *malloc()* has been predictable in the addresses of the memory chunks it allocates. This predictable behaviour can be abused by an attacker who is exploiting a vulnerability in a program. OpenBSD's new *malloc()* implementation is based on the *mmap()* system call. This new implementation actually addresses two important issues. Firstly, it randomises the address space of the program - it will now be different on each execution. This greatly impedes the attacker who is relying upon a predicable address space. Secondly, it introduces the notion of "guard pages". These are pieces of unallocated memory which follow each page-size memory allocation. This guard page mechanism is the most useful aspect of this new *malloc()* implementation for detecting bugs in programs. If the program attempts to access memory located in one of these guard pages, its execution will immediately terminate with a segmentation fault. Thus, this feature is extremely useful for detecting situations where a program attempts to read beyond the end of a memory buffer. If one has a program and it crashes with a segmentation fault only under OpenBSD, it is almost certainly due to an erroneous memory access. When the program is terminated, it will leave a core dump [3]. By examining this core dump, one can quickly find the cause of the bug. Typical bugs which are detected by this mechanism are: use of a memory chunk after it has already been freed, attempting to free a memory chunk twice and off-by-one errors. Further discussion of using this feature to find bugs and develop better software can be found in *O'Higgins and Stühler* [12].

2.3.4 Lint

One final tool worth mentioning is *Lint*[13]. This is a static analysis tool, albeit a basic one. In practice it detects common programmer oversights in C code such as unused variables, missing "return" statements, incorrect casts etc. Lint has been significantly improved by members of the OpenBSD team and has been integrated fully into the build system. As the case study, OpenCVS, is an integrated part of the OpenBSD source tree, it is simple to analyse it with Lint. For illustration, here is some sample output of the command `make lint` in the **src/usr.bin/rcs** directory:

```
/tmp/src/usr.bin/cvs/rcsnum.c:342: rcsnum_dec defined, but never used
ident_line returns value which is always ignored
rcs_desc_set returns value which is always ignored
ci.c:124: cvs_date_parse used, but not defined
vsnprintf returns value which is always ignored
strlcpy returns value which is sometimes ignored
cvs_buf_write_stmp returns value which is sometimes ignored
```

As the OpenCVS source code is of high quality, no serious issues have been found. However, looking at the last line of output, this does show that there are instances where the return value of the function *cvs_buf_write_stmp()* function is being ignored. Looking through the source code for uses of this function reveals there are indeed places where the return value is not being checked. Usually, it is very bad practice not to check for error conditions after a

[3]A core dump is a record of the contents of working memory at a specific time which can be used to debug a program that has crashed.

function call. However, examining the *cvs_buf_write_stmp()* function itself reveals that it has been re-factored so that it can never return on failure - hence it is not dangerous to ignore its return value. Yet, its prototype has not been modified to reflect this change. So in this case, Lint has pointed out a function whose API is ambiguous, which should be updated to increase code clarity.

2.3.5 Static Checking and Automatic Program Testing

Static checking is an important area of research which operates on the source or byte code level. Essentially, functions are converted to mathematical theorems and then verified. Should a function behave in an illegal way (for example, by crashing the program or returning a strange value) then this is considered to be a bug.

One method involves writing formal specifications defining the legal inputs to a function and how the function is supposed to behave. *ESC/Java2* is an implementation of this method. Using *ESC/Java2*, the program is annotated with the Java Modeling Language (JML) and then *ESC/Java2* converts this into verification conditions which are submitted to a theorum prover [14].

A related field is automatic program testing. One method developed by researchers at Stanford attempts to transform an existing program's source code into a form of test upon itself. Instead of analysing pre-written formal specifications, this method attempts to automatically determine the constraints bounding the inputs to a function. Once these constraints are known, "interesting" values can be quickly generated to test corner cases and trigger bugs far faster than by providing random input values [15].

2.4 Development Tools

The ultimate goal of this project is to produce a tool which is capable of locating bugs in a source tree automatically. To implement the method developed (discussed later) a language with strong UNIX integration was required. At first, the Korn Shell was considered and in fact an implementation using it was attempted. The advantages of this language were that it was simple, readily available and very familiar to the author. However, it quickly became evident that it lacked the necessary features. While the Korn Shell is perfect from the perspective of UNIX integration, it is sorely lacking in the areas of text and list processing. Furthermore, it was extremely limited in its capacity to provide complex data structures which were deemed necessary for the implementation of the tool. The Korn Shell has poor and cumbersome support for arrays and no support whatsoever for hash tables. This, more than anything else, led to the decision to choose another implementation language.

The next choice was Perl, which is similar to the Korn Shell in its excellent integration with UNIX. Indeed, it is possible to use constructions very similar to those available in Korn. It is trivial to execute a standard UNIX program and capture the program output and return code. Perl also has a standard function library which is very similar to the UNIX C standard library. Many of the function calls have the same names, behaviours and parameters as the C equivalents. For example, the UNIX library function *stat()* is similarly called *stat()* in Perl. In addition to this, Perl has very good support for complex data structures like hashes and arrays, and indeed it is trivial to combine these to produce for example arrays of hash tables or even hash tables of hash tables. Such complex data structures are used extensively in the implemented tool. Furthermore, Perl also offers excellent text processing tools with in-built support for regular expressions. Also, the more advanced variable system, with notions of scope and a concept of references, was far superior to that which was available in the Korn Shell. Lastly, Perl is also included in the OpenBSD base system and so is as readily available as the Korn Shell. This also means that the possibility of including the implemented tool in the OpenBSD base system at some point in the future, so that the project can benefit from it, exists.

Chapter 3: **Determining How to Find and Store Bugs**

In order to be able to find bugs automatically, it is first necessary to have a solid understanding of bugs. On one level, bugs are a simple concept. Everybody who has used a computer for any length of time probably has encountered a bug in one form or another and therefore has an intuitive grasp of them. On another level, they are highly complicated things, sometimes incredibly difficult to find, fix or from time to time even reproduce. For the developer, bugs typically represent a flaw in the logical structure of a program. Sometimes, bugs can be due to flaws beyond the control of the developer. For example, they could be caused by a third party system library for which that developer is not responsible. They could also be caused by a flaw not in software but in hardware, as was the case a number of years ago with the famous Intel Pentium floating point division (FDIV) bug.

However, this project is concerned with bugs specifically within the OpenBSD project. The relevant question, then, is how bugs are found within the context of OpenBSD. Furthermore, what kind of bugs are found, and how are they typically solved? A very important detail is that OpenBSD is an operating system. It is almost entirely written in the C language, with relatively tiny amounts of machine code, Korn Shell and Perl. Furthermore, work on OpenBSD is classed as systems programming.

According to Wikipedia [16], systems programming

[...] is the activity of programming system software. The primary distinctive characteristic of systems programming when compared to application programming is that application programming is to produce software which provides services to the user (e.g. word processor), whereas systems programming is to produce software which provides services to the computer hardware (e.g. disk defragmenter) requires a greater degree of hardware awareness.

Indeed, the environment within which the systems programmer works, is one which is closer to the hardware. Fewer abstractions are available, especially in the realms of memory management and inter process communication (IPC). Hence, the tools and methods used to find and fix bugs in OpenBSD are quite different from those employed by a modern applications programmer. The next question is how to acquire the requisite knowledge so one can make judgments about the nature of bugs under OpenBSD.

3.1 The OpenCVS Case Study

3.1.1 Case Study Motivation

It was felt that the most effective and productive method to acquire a deep understanding of the work on finding and fixing bugs undertaken within the OpenBSD project was to actually become a participant in it. Real-world, practical experience would yield the best results and additionally make a positive contribution to the larger Open Source community. It was therefore decided to undertake a case study on developing software for OpenBSD.

3.1.2 What is OpenCVS

The next decision taken was in what way to contribute to the project. OpenBSD is comprised of a huge number of diverse components including network daemons, compilers, document formatters, device drivers and filesystems. While it is indeed possible to work on many disparate components of the system - some developers do precisely that - it would be more productive to have a specific focus. This specific focus materialised in the form of the OpenCVS project. OpenCVS is a subproject of OpenBSD tasked with designing and implementing a replacement for the GNU Concurrent Versioning System (CVS). CVS is widely used for the coordination of software development, especially in the Open Source community. However, the GNU implementation is known to have a large number of bugs, some of which had even led to significant security vulnerabilities [17]. A serious problem is that the GNU CVS source code is over fifteen years old and is quite poor from a maintainability perspective. In fact, the GNU CVS code base has been mostly abandoned for some time. For these reasons, the OpenCVS project was started. Its primary goals are to implement from scratch a more secure, maintainable, stable and fully compatible replacement for GNU CVS.

3.1.3 Status of OpenCVS

By mid 2005, OpenCVS had been under development for approximately twelve months, chiefly by Jean-Francois Brousseau. However due to personal commitments he was forced to cease working on the project, leaving it largely unimplemented. Two other OpenBSD developers, Xavier Santolaria and Joris Vink, continued to work on OpenCVS but such a large project required more than two people. In July 2005, the OpenBSD project leader Theo de Raadt suggested to the author of this report that he join with Santolaria and Vink in developing OpenCVS. He did so, and quickly noticed that a very rough beginning had been made to use the existing OpenCVS code to re-implement GNU RCS. GNU RCS was first developed in the early 1980s by Walter Tichy at Perdue University. It operates on individual files, whereas CVS operates on directories of files. The RCS file format is also used by CVS which is why an RCS implementation lies at the heart of OpenCVS. To re-implement GNU RCS is a significantly smaller task than to re-implement GNU CVS. Hence it was decided that work should focus there, especially since this would ensure that the design of the RCS API - which is also central to OpenCVS - was correct.

3.1.4 Why OpenCVS

The timing of these developments coincided almost perfectly with the specification of this final year Computer Science project. The RCS re-implementation project, henceforth referred to as OpenRCS, provided the ideal environment within which the nature of bugs in OpenBSD could be researched. Furthermore, as it was being written from scratch, there was much scope to assess the true potential of an automatic tool developed for finding bugs.

3.2 The Bug Database

Once bugs are found, one needs somewhere to store information about them. Initially, it was the intention that the bug database should be developed completely from scratch. The preliminary idea was to implement a web-based front end to a database such as PostgreSQL which would store the bug information. However, Dr. Kiniry suggested that a better approach would be to take some existing bug tracking system and modify it to suit the needs of the project. Furthermore, he offered the view that in fact a web-based front end should be avoided and instead a more interesting and versatile system be investigated. He suggested GNATS as an example of such a system.

3.2.1 GNATS

Research revealed that the OpenBSD project was already using GNATS as its bug tracking system. This fact constituted a strong argument for the adoption of GNATS. It would surely be far more beneficial to the OpenBSD project if any tool developed would be able to integrate with the bug tracking software they were using. Additionally, it was noticed that the version of GNATS in use by the OpenBSD project was quite old - the 3.x series. Development of that version has ceased and moved onto the 4.x series. 4.1.0 is the most recent GNATS version, and it was decided that this version should be ported to OpenBSD and utilised for the purposes of this final year project.

3.2.2 OpenBSD Ports Collection Framework

The existing GNATS 3.x series port for OpenBSD had been implemented through the OpenBSD Ports Collection framework, henceforth simply referred to as the "ports framework". Software ported using this framework will henceforth be referred to simply as "ports". The ports framework provides a uniform structure within which to port software. It provides many benefits. First and foremost, it generates a single archive file known as a binary package which can be installed on other machines of the same architecture. This is extremely useful as it greatly reduces time consuming compilation. For example, one could use a fast machine to build the binary package and then install that package on other, slower machines. Additionally, the ports framework handles in-place updates of software packages. For example, assume that there is an existing port of application X which is at version 1.1, and a new version 1.2 was released. If one updates the port to support application X version 1.2, the resulting binary package can be used to upgrade any machines which have already installed version 1.1. Finally, the ports framework allows a port author to specify various forms of meta-data to be associated and included with the binary package. This is very useful to inform the end-user of, for example, configuration steps necessary after package installation or a detailed description of the features the software package provides.

While the ports framework offers many useful benefits, it can make it more difficult to port an application. This is because the ports framework expects applications to have a standards-compliant build system. UNIX application build systems usually respect various environment variables which specify parameters such as compiler flags, the base application install directory, etc. If a UNIX application build system does not respect these environment variables, usually it is necessary to modify it in order that it will build correctly under the ports framework. Fortunately, the ports framework is extremely mature, with over three thousand existing ports, and so it has well-developed support for patching.

It was decided that a port of GNATS 4.x series to OpenBSD should be done within the ports framework. Such a port is likely to be useful to many people and if there is sufficient interest it may be included in the official OpenBSD Ports Collection. This would make the GNATS 4.x port available on all future releases of OpenBSD and a binary package would be available for all architectures via FTP. Although using the ports framework requires much more work than a simple ad-hoc port, the benefits are numerous.

3.2.3 Integration with an Automatic Tool

A most interesting possibility was that of integrating any automatic bug finding tool developed with the bug database. With this feature, the results of the tool would be easily visible in the central bug database. As the primary goal of the bug database is for it to be Internet-accessible, the tool reporting its results to the database ensures that they too will be available on the Internet. One can imagine a scenario where a dedicated computer regularly runs the automatic tool after every commit to the source code - a process which is often relatively time-consuming - and sends the results to the bug database. Thus developers are not burdened with having to waste time while their machines run the tool and furthermore run no risk of forgetting to run it after a commit. As GNATS has a very well supported SMTP interface, it was found that this could be leveraged to support remote bug submission. Details of this implementation shall be presented in a later chapter.

Chapter 4: **Design of the Automatic Tool**

From the very beginning of this project, the focus was on trying to find a method for automatically finding bugs in software. During the course of the OpenCVS case study, there came a necessity for a regression testing framework for that project. The problem encountered was that as new features were implemented, frequently they would negatively impact existing features. An automatic method for testing OpenRCS was required. This tool would be run by developers to verify that no feature regressions had taken place. It would test as much functionality as possible, in order to have the greatest possible coverage.

4.1 Manual Regression Tests

The OpenBSD project already had in place a series of regression tests. The first step was to examine these. It was found that the existing regression tests were all written by hand. Effectively they consisted of a series of scripted program executions with manual tests for various error conditions, such as incorrect exit code or bad file permissions on output files.

The author of this report initially set about writing a comprehensive set of manual regression tests for OpenRCS. While effective to a certain extent, they were extremely time consuming to create. A very limited set of test cases took many hours. These tests consisted of a large number of repetitive Korn Shell statements, such as the following:

```
echo "this is a test file" > test
echo "a test file" | $BSD_CI -l test
echo "another revision" >> test
echo "Testing 'ci test' with non-interactive log message"
$BSD_CI -m'a second revision' test
ls test 2>1 1> /dev/null && failure || echo "'test' working file is gone [OK]"
echo "Testing 'co -l test'"
$BSD_CO -l test
ls test 2>1 1> /dev/null && echo "test file exists [OK]" || failure
echo "new stuff" >> test
echo "Testing 'ci -r1.30 test' with non-interactive log message"
$BSD_CI -r1.30 -m'bumped rev' test
ls test 2>1 1> /dev/null && exit 1 || echo "'test' working file is gone [OK]"
```

The above sample implements only three tests. A further problem with such manual regression tests is that they are effectively complicated programs in their own right. As such, they are error-prone and require debugging, just as any other program written by a human. Hence it was decided that while manual regression tests are indeed useful and greatly aid in development, they are not optimal for automatically discovering a significant quantity of bugs. It is simply unfeasible to manually write tests which cover a large portion of the possible executions of a program with as many options as OpenRCS. OpenRCS consists of two main programs, *ci* and *co*, each of which has approximately sixteen command-line switches, many of which take optional arguments and all of which may be combined almost arbitrarily.

4.2 A Different Approach to Automatic Testing

4.2.1 Existing automatic approaches

One successful method of testing is, as detailed in the background research section, to specify the valid inputs to a function, feed it "interesting" inputs and then verify the output of the function. Unfortunately, the C language has a number of qualities which make this sort of analysis at the source code level very difficult. While approaches do exist, for example through the use of Berkeley CIL [18] as demonstrated by researchers at Stanford, they were considered beyond the scope of a final year undergraduate Computer Science project.

4.2.2 A higher level view

However, if one takes a higher level view of a UNIX command-line program, one realises that it can be treated as a sort of function in its own right. UNIX is based fundamentally on the concept of multiple byte streams. Programs read a stream of bytes from some source - e.g. the disk, standard input, a network socket - and write a stream of bytes to some output - e.g. the disk, standard output, a network socket. Thus a given UNIX command-line program can be viewed as a sort of "black box" which takes input, transforms it, and outputs it. If one specifies what sort of inputs it accepts, and what its outputs should be, one can automatically feed it values and verify the outputs. While complete test coverage would still be infeasible, given the fact that certain inputs are effectively infinite (e.g. files can theoretically be of an infinite size) this approach offers orders of magnitude better test coverage when compared with manual regression tests. Furthermore, the human element is no longer required for the creation of each individual test, greatly reducing the possibility of introducing bugs.

4.2.3 Automatically determining inputs

While this approach appeared to offer a practical method of automatically testing a large part of OpenRCS, two major questions remained. First of all, how to best specify the inputs to a program. The initial approach was to design an input specification language which would be used for this express purpose. This would undoubtedly be a feasible solution, but during the course of development a simple but crucial observation was made. All high quality UNIX command-line applications have a 'usage' function which can be invoked to inform the user what command line options are. Typically this is printed when the '-help' argument is passed. For example the *ls* command, when invoked with '-help', prints the following:

```
usage: ls [-1AaCcdFfghikLlmnopqRrSsTtuWx] [file ...]
```

The crucial observation is that this "usage" output constitutes a pre-written input specification for the program. The OpenBSD project works hard to ensure that such "usage" messages are correct and synchronised with the actual functionality of the program. Thus is it very unlikely that any “usage” message will be an incorrect specification. Furthermore, such messages are quite easy for a machine to understand. They follow a relatively simple and consistent grammar. In the case of *ls*, the valid inputs are series of command line options, none of which taken any arguments, followed optionally by one or more file names.

For illustration, here is the much more elaborate “usage” message from the OpenRCS program *ci*:

```
usage: ci [-jMNqV] [-d[date]] [-f[rev]] [-I[rev]] [-i[rev]]
          [-j[rev]] [-k[rev]] [-l[rev]] [-M[rev]] [-mmsg]
          [-Nsymbol] [-nsymbol] [-r[rev]] [-sstate] [-tfile|str]
          [-u[rev]] [-wusername] [-xsuffixes] [-ztz] file ...
```

Obviously, this is more involved - but it is still quite possible for a machine to parse this information to build an input specification automatically. The following rules were observed: all options which accept no argument at all are in the first set of square brackets. The remaining options all take either an optional or a mandatory argument. Arguments are optional if they are enclosed in square brackets. Otherwise, arguments are mandatory. The type of argument expected by a particular option is extrapolated from the string description that argument. For example, the string “rev” implies revision number value. The string “date” implies a date value. These strings are common across all the programs comprising OpenRCS.

4.2.4 Automatically validating outputs

The outputs of a command-line program in UNIX are typically exit code, output to the standard output and standard error streams, and some files. In the case of OpenRCS, the specific outputs are exit code, standard output, standard error, the RCS file and the working copy. Initially, the the approach was to design an output specification languge which would be used to determine the correctness of a given output. However, a second observation greatly simplified the situation. As the primary goal of OpenRCS is to be full compatible with the GNU RCS implementation, if one feeds the same inputs to GNU RCS that had been fed to OpenRCS, the outputs should be identical. Therefore, if the outputs are different, there is a bug.

Chapter 5: **Implementation of the Automatic Tool and Bug Database**

As was mentioned in the Background Research section, Perl was chosen as the language in which to implement the automatic tool. This chapter will describe its implementation. Additionally, this chapter describes the work required to port GNATS to OpenBSD.

5.1 Data Structures

As Niklaus Wirth stated in his book *Algorithms + Data structures = Programs*, one of the first things one should do when writing a program is to choose one's data structures. Fortunately Perl makes complex compound data structures (hash tables of arrays, hash tables of hash tables, etc..,) relatively easy to use. At the core of the automatic tool lies the requirement to store a machine-usable description of the options and arguments accepted by the program it is analysing. This can be represented by a hash table where each command-line option comprises the key used to access a property list for that option. The property list would contain information about the arguments required by that option, whether they were mandatory, etc. Two options were considered to implement the property lists; arrays and hash tables. Thus, the proposed core data structures were a hash table of arrays and a hash table of hash tables.

5.1.1 Hash table of arrays

The array would contain information about the arguments required by that option, whether they were mandatory, etc. However, storing information about the arguments in an array is not optimal. It requires that the position of various properties be hard coded at a specific index, or that an additional search be performed. To illustrate, consider the array corresponding to the "-d[date]" option of the *ci* program. The properties of this option are that it takes an argument called "date", and that this is an optional argument. The Perl code below shows how this would be expressed and processed using an array:

```
# Load the properties of -d[date] into the array
my @properties = [ ‘‘optional’’, ‘‘date’’ ];
# Operate conditionally based on the properties of the array
if ($properties[0] eq ‘‘optional’’) {
  if ($properties[1] eq ‘‘date’’) {
    # handle optional date ...
  }
} elsif ($properties[0] eq ‘‘mandatory’’) {
  if ($properties[1] eq ‘‘date’’) {
    # handle mandatory date ...
  }
}
```

Hence, the use of the array data structure, complicates the program logic considerably. Should the developer use an incorrect array index at any point, the program will fail.

5.1.2 Hash table of hash tables

The second proposed approach was to use a hash table to store these properties. Using this approach, the developer need not consider array indexes at all, but simply uses the name of a property to find its value. The Perl code below shows the hash table version of the previous example:

```
# Load the properties of -d[date] into the hash table
my %properties = { ``date'' => ``optional'' };
# Operate conditionally based on the properties of the hash table
if ($properties{``date''} eq ``optional'') {
    # handle optional date ...
} elsif ($properties{``date''} eq ``mandatory'')  {
    # handle mandatory date ...
}
```

This code is less complicated than the array-based version with no additional costs, so it was chosen.

5.2 Algorithms

There are two fundamental algorithms implementing the automatic tool; the usage message parser and the test loop. Additionally, there are other algorithms for result verification and bug reporting.

5.2.1 Usage parser

As was discussed previously, the input specification is determined by parsing the usage message of the program. This is implemented by two functions; *parse_usage()* and *parse_arg()*. The *parse_usage()* function splits the usage message into individual strings per argument. For example, for *ci*, it would split the usage message into “[-jMNqV]”, “[-d[date]]”, “[-f[rev]]” etc. *parse_usage()* then calls *parse_arg()* with each such argument string as the parameter. This function has two parsing modes. One is a special case for the very first argument string e.g. “[-jMNqV]” in the case of *ci*. The first argument string is always comprised of those options which take no argument, optional or otherwise. Thus, these are inserted into the hash table with just an empty list as their argument property list. The second parsing mode is more complicated. It implements a small finite state machine to parse the argument string and determine the type of the argument and whether it is optional or mandatory. There are only two states; one state where the parser is looking for the option e.g. ’d’ in the case of “[-d[date]]”. The other state is where the parser has found the option and is attempting to determine whether the argument following it is optional or mandatory. This is deduced by checking the character directly following the option character, e.g. '[' in the case of "[-d[date]]" and 's' in the case of "[-xsuffixes]". If the character directly following the option character is '[', then the argument is optional. Otherwise, the argument is mandatory. To find the argument type, the parser simply copies the string from its starting character - that is, either directly after the option in the case of mandatory arguments, or directly after the following '[' character in the case of optional arguments - to the ending ']' in the string.

5.2.2 Test loop

The actual testing algorithm consists of a nested loop which iterates through the keys in the input specification hash table. Pseudo code illustrates this below:

```
for each key in hash table
do
    let option1 = current key1
    for each key in hash table
    do
        let option2 = current key2
        { get correct arguments etc }
        run test with option1 and option2
    od
od
```

In the case of *ci*, this leads to a series of executions similar to the following rough illustration (leaving out arguments etc):

```
ci -d -d
ci -d -f
ci -d -I
{ ... }
ci -f -d
ci -f -f
ci -f -I
{ ... }
ci -u -d
ci -u -f
ci -u -I
{ ... until all combinations are exhausted }
```

The test loop algorithm has been implemented in the automatic tool in the function *run_tests()*. It has been made independent of the actual tests being performed. That is, the loop function is not aware of the details of the tests it is running. This has been achieved by making the *run_tests()* function accept a reference to a function as one of its parameter. This means one writes the test logic in a separate function which is then passed as an argument to the *run_tests()* function. In the current implementation, there are two test functions corresponding to the two applications being tested; *test_ci()* and *test_co()*.

Applying the test loop algorithm to these different functions is simple, as illustrated by the following code snippet:

```
# Run test loop for co
run_tests(\&test_co);
```

This method adds greatly the modularity of the program and significantly reduces code duplication.

5.2.3 Result verification

Two functions are responsible for verifying test results, *get_res()* and *check_res()*. The function *get_res()* actually executes the program and gathers the outputs. The function *check_res()* is then passed the resulting outputs from executions of GNU RCS and OpenRCS and checks them for differences.

get_res() returns its results in the form of a hash table. There are five keys in this hash table, "command", "exit_code", "wrk_stat", "rcs_stat" and "rcs_contents". The "command" key accesses the the full command-line executed during the test. The "exit_code" key accesses the resultant exit code of the execution. The "wrk_stat" key accesses the results of the Perl *stat()* function when called with the working file as an argument. The *stat()* function is roughly analogous to the UNIX *stat()* system call - it returns the various properties of a file. Properties include file permissions, file size in bytes, etc. The "rcs_stat" key accesses the results of the Perl *stat()* function when called with the RCS file as an argument. Finally, the "rcs_contents" key accesses the full contents of the RCS file.

check_res() compares two such hash tables for differences. It first compares the "wrk_stat" values, followed by the "exit_code". The contents of the RCS files are compared using the *diff()* function. This function provides a wrapper around the standard UNIX *diff* program, which is very commonly used for disparate tasks from generating source code patches to performing checking whether two directories have identical contents. The *diff* program has a number of different output formats. The format selected for the purpose of the automatic tool was unified diff. Unified diffs are easily understood by humans, and are typically used for patches to source code.

5.2.4 Bug reporting

When differences are found by the automatic tool, they are reported two ways. The default is to print to standard output. This makes it trivial to save the results of analysis to a file, to search it or to compare with a previous analysis. The second bug reporting mechanism is to submit it to the GNATS bug database described previously. This is achieved using the SMTP protocol. GNATS' primary bug submission interface is e-mail. As such, the automatic tool need simply email the report to a specific address and it will be processed by GNATS. This is extremely useful as it makes it trivial for the automatic tool to submit its results to a remote database.

In its bug reports, the tool attempts to include as much information as possible. It includes the full command line which triggered bug, the specific output which differed along with the value it should have been. In the case of RCS file differences, a full unified diff illustrating file differences is generated.

5.2.5 Input value generation

Generation of "interesting" boundary values is performed by various functions. The vast majority of command-line input values to the OpenRCS programs consist of revision numbers. Internally to OpenRCS, these are stored in unsigned 16-bit integer variables. Hence, values which are "interesting" in this context include: 1, 0, -1, maximum signed integer size i.e. 2^{31} and minimum signed integer size i.e. -2^{31}. The next most common command-line input values consist of time stamps i.e. a date in YYYY-MM-DD form followed by a time in HH:MM:SS form. "Interesting" values in this context include a completely zero timestamp, i.e. "0000-00-00 00:00:00", a maximal timestamp, i.e. "9999-99-99 99:99:99", an invalid pre-1970-01-01 timestamp (UNIX time is counted as seconds elapsed since 00:00:00 UTC 1970-01-01) e.g. "1900-19-19 19:19:19" and a final valid pre-1970-01-01 timestamp e.g. "0400-11-27 01:03:01".

These values are stored in simple arrays. Two functions, *generate_rev()* and *generate_date()* provide access. Each will return the values in sequential order, looping when they are depleted. This is to ensure that failures are repeatable. The test loop will execute each set of options and arguments enough times so that every "interesting" value is tested.

It was found that generating random input files did not increase the number of bugs found. Furthermore, RCS file are extremely difficult to generate dynamically. For this reason, such functionality has been left out of the automatic tool at this time. Sample "typical" RCS and working files are used instead.

5.3 GNATS 4 Port

The GNATS 4.1.0 port was done using the OpenBSD Ports Collection framework. This consisted mostly of setting various values in a Makefile template, for example specifying where the program source code could be retrieved from and the system libraries the software relies on. Often, filling in the values in the template is all that is needed to create a working port. However, in the case of GNATS 4.1.0, this was not so.

5.3.1 Creation of directories

Issues were discovered with the order in which GNATS 4.1.0's build system was creating directories. Effectively, when built using the ports framework, it would not create the directories into which it was installing itself. The solution to this problem was to manually create these directories explicitly using a "pre-fake" target in the port Makefile. The "pre-fake" target is executed before the port is installed into a fake environment from which the binary package is built.

5.3.2 Incorrect install arguments

UNIX applications built from source code typically install themselves using the *install* program. This program copies the files and sets permissions and ownership of them. GNATS 4.1.0 does not use the system *install* program. Instead, it uses its own script which is written in Bourne Shell, named *install-sh*. In the case of GNATS 4.1.0, it was found that the files were being installed with completely the wrong arguments. This rendered the port unusable. The root cause was very difficult to discover. In the end, it was found to be a regular expression in the *install-sh* script:

```
sed -e 's,[^/]*$,,;s,/$,,;s,^$,.,'`
```

Needed to be changed to:

```
sed -e 's,/$,,g' -e 's,[^/]*$,,;s,/$,,;s,^$,.,'`
```

The additional expression ensures that the correct arguments will be used in the install script. This change is made automatically by the ports framework when a file containing the patch is placed in the port's `patches` directory.

5.3.3 Sendmail set up

GNATS 4.1.0 accepts bug reports via SMTP. In order to facilitate this functionality, a mail server must be configured to pass the relevant information to GNATS. On OpenBSD, the system comes pre-configured with the Sendmail mail server. The only Sendmail configuration required is to specify the GNATS bug submission processor as the handler for mail going to a specific address. This is done in the Sendmail *aliases* file. On OpenBSD, this file is located in */etc/mail/aliases.* The change consists of a single line:

```
bugs:  "| /usr/local/libexec/gnats/queue-pr -q"
```

This instructs Sendmail that all mail sent to user “bugs” should be piped to the GNATS *queuepr* program for processing. Once this change has been made, the *newaliases* program should be run to inform Sendmail to refresh its aliases database.

Chapter 6: **Results and Analysis**

In this chapter, a sample of bugs found by the automatic tool in the OpenCVS source code is presented. Additionally, analysis is performed on the root causes of these bugs from which API guidelines are proposed.

6.1 Revision Number Zero

6.1.1 Details

The very first bug found by the automatic tool was related to the handling of revision number zero. The program *ci* when passed a revision number of 0 as an argument to any of the options would fail like so:

```
<niallo@obsidian:rcs>$ echo "a test input file" > test
<niallo@obsidian:rcs>$ ./ci -l0 test
test,v  <--  test
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> this is a test file
>> .
ci: test,v: No revision on branch has.
```

There were a considerable number of changes required in the code for it to fully support revision number 0 correctly, and the correct fix was relatively involved [19]. However, two flaws were fundamental to this bug. Firstly, the *rcsnum_aton()* function in rcsnum.c was unable to handle any strings containing revision numbers in non-dotted form. For example, the revision number 1 is in non-dotted form, the revision number 1.0 is in dotted form. Secondly, the number would be decremented by the *rcsnum_rev_add()* function in rcs.c in order to find the previous revision. However, it is not legal to have a previous revision where the second number in the dotted form is zero e.g. 1.0. Instead, the second number of the previous revision must be (at minimum) 1 e.g. 1.1. Thus, the *rcsnum_dec()* function contained an off-by-one error. The conditional check:

```
if (num->rn_id[num->rn_len - 1] <= 0)
```

Needed to be changed to:

```
if (num->rn_id[num->rn_len - 1] <= 1)
```

6.1.2 API guidelines

This bug highlights some important issues in API design, namely that functions should guarantee that they will always return valid data. Both *rcsnum_aton()* and *rcsnum_dec()* in this case did not correctly guarantee that they would return legal RCS revision numbers. Additionally, this bug emphasises the way in which seemingly simple conditionals should always be thought about very carefully by the implementor to avoid bugs. Off-by-one errors appear on the surface to be minor issues but in reality they very often cause major problems in software.

Additionally, this bug points out the limitations of human testers. Over a period of several months, not once had a developer tried to specify 0 as a revision number. This illustrates that it often does not not occur to humans to test software with counter-intuitive boundary cases. As such, an automatic tool which is specifically designed to test such boundary cases has the capacity to very quickly reveal bugs.

6.2 Revision Number Overflow

6.2.1 Details

The next bug found was similar to the previous bug. The program *ci* when passed a very large revision number as an argument to any of the options would fail like so:

```
<niallo@obsidian:rcs>$ ./ci -l9999999999 test
ci: RCSNUM overflow
Segmentation fault (core dumped)
```

This is clearly an integer overflow error. However, what makes it interesting is the fact that the program appears to correctly detect the overflow condition correctly, yet the program crashes nonetheless. The problem again lies in the *rcsnum_aton()* function. While the overflow is correctly detected, the rcsnum.c API attempts to pass the error handling to the calling function instead of simply aborting execution. The root cause is related to the character pointer that *rcsnum_aton()* function takes as a parameter, the char **ep parameter. On successful completion, *rcsnum_aton()*, will leave this pointer pointing to a valid memory address. However, when returning from this overflow condition, *rcsnum_aton()* will leave it pointing to an invalid memory address. Thus when the calling function attempts to dereference this pointer, the program crashes. Essentially, the function does not correctly restore values to a consistent state.

The fix [21] was simply to abort execution cleanly and warn the user by calling *fatal()*, instead of trying to return:

```
fatal("RCSNUM overflow!");
```

Instead of

```
goto rcsnum_aton_failed;
```

6.2.2 API guidelines

This bug again emphasises many of the same issues in API design and implementor as the previous one. API implementors should ensure that the functions they write leave data in a consistent state, even in the case of error. Additionally, in this case the decision to return at all in the case of error was arguably a poor one. When dealing with such a critical error such as integer overflow, the best practice is to abort execution entirely. Overly complicated error handling of integer overflows has in the past led to major security flaws in applications [20].

6.3 Exit Code Bug

6.3.1 Description

A common incompatibility between OpenRCS and GNU RCS was in the exit codes produced. Frequently, OpenRCS would exit without setting a correct exit code. This was an important issue because exit codes of UNIX command-line applications are often used by shell scripts to test for failure or success. The program *ci* when passed an invalid argument would fail like so:

```
<niallo@obsidian:rcs>$ ./ci -l nonexistant file
nonexistent,v  <--  nonexistent
ci: cvs_buf_load: open: 'nonexistent': No such file or directory
<niallo@obsidian:rcs>$ echo "exit code is: $?"
exit code is: 255
```

The value 255 was being specified by the *fatal()* function. *fatal()* was an API function borrowed from the OpenSSH code base. It immediately aborts execution of the application after printing a message to standard error. The fix [22] was to change:

```
exit(1);
```

Instead of

```
exit(255);
```

6.3.2 API guidelines

When implementing an API with borrowed code, it is critical to consider any differences between the environments. Furthermore, developers of UNIX applications should pay close attention to the exit codes returned by their programs. While exit codes are invisible to the casual user, they are of vital importance within a scripted, task automation context.

Chapter 7: **Conclusions and Future Work**

7.1 Conclusions

A method for automatic testing of UNIX command-line applications has been developed and implemented. It has been used successfully to find a number of bugs very quickly in the OpenCVS case study applications. This form of automatic testing has in particular demonstrated that it is effective in testing cases which humans would typically overlook. API guidelines based on analysis of bugs found with this tool have been proposed.

With the help of this tool, quite a number of bugs - large and small - have been fixed in the OpenCVS code base. The GNATS bug tracking system has been ported to OpenBSD in a way which makes it readily available to any member of the OpenBSD community. Both the work done during the course of the OpenCVS case study and the GNATS 4 port represent significant contributions to both the OpenBSD and wider Open Source communities.

However, it is important to acknowledge the shortcomings of the project. No specification language has been developed for program outputs. As such, adapting the current tool to work with other UNIX command-line applications would be a difficult task if there was no existing reference implementation. In effect, GNU RCS offered a convenient short cut to verifying the OpenRCS outputs.

Furthermore, the inputs generated are very limited and restrict the tool to quite a small test coverage. Lacking the ability to generate RCS files dynamically means that the RCS parser is hardly tested at all by this tool.

While the integration with the GNATS bug database works, reporting bugs over SMTP is quite slow. Additionally, due to limitations in GNATS itself, the SMTP bug submission method makes it impossible to specify many additional properties of the bug being reported. Specifically, class and category cannot be specified using the SMTP bug submission method.

7.2 Future Work

Future work would of course include addressing the shortcomings mentioned above. However, the general approach of automatically determining inputs, generating “interesting” boundary values and then checking the outputs could have very useful applications in the domain of Web-based applications. Forms on Web sites implemented in HTML should also be possible to parse in order to determine input specifications. Deploying a tool on the local network segment or even on the machine hosting the Web-based application would enabled it to submit large quantities of generated inputs. Output pages could be verified using an XML parser in combination with regular expressions. Perl would be well suited to implementing such a tool.

Bibliography

[1] The OpenBSD Project website.
http://www.OpenBSD.org/

[2] The OpenCVS Project website.
http://www.OpenCVS.org/

[3] The GNU RCS website.
http://www.gnu.org/software/rcs/rcs.html

[4] Wikipedia entry on software bugs.
http://en.wikipedia.org/wiki/Software_bug

[5] Sendmail remote root vulnerability.
http://seclists.org/bugtraq/1996/Sep/0021.html

[6] T.C. Miller, T. de Raadt, ”strlcpy and strlcat—Consistent, Safe, String Copy and Concatenation,” FREENIX’99, USENIX Assoc., Berkeley, CA.

[7] The Bugzilla Bug Tracker.
http://www.bugzilla.org

[8] The GForge collaborative development environment.
http://www.gforge.org/

[9] The GNATS Bug Tracker.
http://www.gnu.org/software/gnats/

[10] IBM ProPolice.
http://www.trl.ibm.com/projects/security/ssp/

[11] OpenBSD gcc-local(1) manual page.
http://www.openbsd.org/cgi-bin/man.cgi?query=gcc-local

[12] Niall O’Higgins, Uwe Stühler *Embedded OpenBSD*, Section 1.7, Proceedings, EuroBSDCon 2005.

[13] Lint, a C program verifier manual page.
http://www.openbsd.org/cgi-bin/man.cgi?query=lint

[14] Kiniry, J., Chalin, P., and Hurlin, C. ”Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification.” VSTTE 2005.

[15] Cristian Cadar and Dawson Engler ”Execution Generated Test Cases: How to Make Systems Code Crash Itself.” SPIN 2005.

[16] Wikipedia entry on systems programming.
http://en.wikipedia.org/wiki/Systems_programming

[17] Heap-based buffer overflow in CVS.
http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0396

[18] CIL (C Intermediate Language).
http://manju.cs.berkeley.edu/cil/

[19] Revision number zero bug fix commit messages.
http://article.gmane.org/gmane.os.openbsd.cvs/47911
http://article.gmane.org/gmane.os.openbsd.cvs/47910

[20] OpenSSH integer overflow vulnerability.
http://www.openssh.com/txt/preauth.adv

[21] Revision number overflow bug fix commit message.
http://article.gmane.org/gmane.os.openbsd.cvs/46516

[22] Exit code bug fix commit message.
http://article.gmane.org/gmane.os.openbsd.cvs/46552