Hunting for vulnerabilities in large software : the OpenOffice suite

Wei Ming Khoo
University of Cambridge

Saad Aloteibi
University of Cambridge
{wmk26,smsa3,rja14}@cam.ac.uk

Ross Anderson
University of Cambridge

Michael Meeks
Novell, Inc.
michael.meeks@novell.com

Abstract

How much effort does it cost to find zero-day vulnerabilities in widely-deployed software? As an exercise, we searched for vulnerabilities in OpenOffice, a productivity suite used by about a hundred million people. Within a 4-month period, we found a total of 15 vulnerabilities, including buffer overflow errors, out-of-bound array index errors and null pointer dereferences, using publicly available analysis and debugging tools. About half of the total effort was invested upfront in learning the software and tools; thereafter we found exploitable bugs at a steady rate. This is worrying; if two first-year research students working for 4 months can increase by about 10% the total number of vulnerabilities ever discovered in a large program that has been available for a decade, this suggests that no more than a few years' worth of security testing effort have been invested in total in this product – calling into question the 'many eyes' theory of open-source software security. It also suggests that, at equilibrium, the 'market price' for a zero-day exploit might be very reasonable. We discuss the challenges in analysing large software systems and suggest possible ways in which finding bugs might be made even cheaper.

1 Introduction

Large, complex free and open-source applications are being used by ever more people. As of 2009, the Firefox browser has an estimated 270 million users [10]. Since its inception in 2000, the popularity of the OpenOffice suite has grown steadily and it is currently second only to the Microsoft Office suite. According to a recent German study conducted in February 2010 [8] OpenOffice was installed on 21% of PCs in Germany. The market leader, Microsoft Office was found to be installed on 72% of PCs studied. The OpenOffice suite has been downloaded 98 million times as of 2007 [17].

Applications are also getting larger. The number of raw lines of code in the Firefox browser stood at about 7 million in version 3.0.11. The popular Apache web server version 2.2 had about 800,000 raw lines of code. OpenOffice 3.1.1 has approximately 8.2 million lines of C++ code, 2.4 million lines of Java code, 800,000 lines of C, 30,000 lines of BASIC, and 60,000 lines of Perl including testing code and comments. Compiling it on an Intel Dual-Core 2GHz machine with 4Gb RAM took about 2 hours.

Vulnerabilities are common in software, more so in large software suites such as OpenOffice. To date, the US National Vulnerability database (NVD) lists 42 vulnerabilities [15]; the security organization Secunia lists 94 vulnerabilities [24]. Of the supported file formats, parsing Microsoft Office 1997–2003 formats (9) and graphic objects (7) proved most error-prone, according to the NVD. Categorised according to error type, buffer overflow (15) and integer overflow errors (11) were most prevalent. There is a growing need to develop tools and techniques that can handle the scale and complexity of such software.

The contributions of this work are

- We discuss our experience in hunting for vulnerabilities in OpenOffice, explaining the methods and tools used, and the vulnerabilities discovered (Section 2). We made use of publicly available analysis and debugging tools, and devoted 4 months to testing. The work was substantially done as a training exercise by the first two authors, who at the start of the project were first-year graduate students rather than experienced security testers.
- We discuss the challenges in testing large software and propose avenues of future vulnerability testing research (Section 3).

Section 4 describes previous work done in large-scale vulnerability analysis.

Types of files checked	.c,.cpp,.cxx
Category 5 (e.g. chmod, readlink, strncat)	32
Category 4 (e.g. access, strcat, system)	2504
Category 3 (e.g. getenv, srand, getopt)	628
Category 2 (e.g. char, memcpy, strcat)	5609
Category 1 (e.g. read, strlen, getchar)	4260
Total	13033
Time taken	405.79 secs

Table 1: Summary of warnings by FlawFinder

Types of files checked	.cpp,.cxx
Category error (e.g. null pointer dereferences, memory leaks)	150
Category style (e.g. redundant conditions, uninitialised variables)	2416
Category possible style	3
Total	2569
Time taken	5.5 hours

Table 2: Summary of warnings by Cppcheck

2 Methods and tools

We used three methods in this work – static analysis tools, fuzzing, and multiple path exploration, which we will describe now.

2.1 Static analysis tools

FlawFinder [28] and CppCheck [11] work on a purely syntactic level, detecting programming constructs that are known to cause security problems. FlawFinder identifies commonly misused declarations, such as static arrays, and system calls, such as strcpy, while the algorithm behind Cppcheck is not documented. Summaries of the flaws found by FlawFinder and Cppcheck are given in Tables 1 and 2 respectively.

We made use of three strategies to deal with such a large number of possible bugs. First, we sorted the results by module. Second, we eliminated what we deemed to be non-essential files based on their directory or file names, and concentrated on the few directories that contained the most errors. Examples of uninteresting directories were "examples", "testing", "qa", "Sample*". Our final strategy was to concentrate only on possible security vulnerabilities such as buffer overflow errors and null pointer dereferences.

2.1.1 Results of using static analysis tools

The warnings flagged by FlawFinder and Cppcheck fell into three categories–true positives, false positives and inconclusive warnings.

2.1.2 True positives

An example of a true positive was a buffer overflow vulnerability found in `NPP_StreamAsFile`. The variable `filename` is a fixed-length 1024-byte buffer. There is no check for the length of `stream->url` before `strcpy` is called, so a 1025-char url will overflow the buffer at line 724.

```
714: // save fname to another file with
        the original file name
715: void
716: NPP_StreamAsFile(NPP instance,
                      NPStream *stream,
                      const char* fname)
717: {
718:    debug_fprintf(NSP_LOG_APPEND,
                      "Into Stream\n");
719:    char* url = (char*)stream->url;
720:    char filename[1024] = {0};
721:    char* pfilename = NULL;
722:    if (NULL !=
            (pfilename = strrchr(url, '/'))
        )
723:    {
724:        strcpy(filename, pfilename+1);
725:    } else {
726:        return;
727:    }
```

A second example of a true positive was the following null pointer dereference in `ImplDeleteCharTabData`. There is no check before the pointer `pToolsData` is dereferenced at line 336, and this will cause a segmentation fault if `BOOTSTRAP` is defined. Though not exploitable on its own, this error could be used as part of a local privilege escalation attack.

```
328: void ImplDeleteCharTabData()
329: {
330: #ifndef BOOTSTRAP
331:     TOOLSINDATA* pToolsData =
            ImplGetToolsInData();
332: #else
333:     TOOLSINDATA* pToolsData = 0x0;
334: #endif
335:     Impl1ByteUnicodeTabData*
            pTempUniTab;
336:     Impl1ByteUnicodeTabData*
            pUniTab =
            pToolsData->mpFirstUniTabData;
```

2.1.3 False positives

False positives were generated because

- checks that were done by the programmer were not taken into consideration, and
- control-flow logic that ensured that the error condition would not occur was not considered.

The first example of a false positive is given below. The `strcpy` was safe as the size of buffer `mimetype` was sufficient to store the contents of `type`.

```
NPMIMEType
dupMimeType(NPMIMEType type)
{
    NPMIMEType mimetype =
     (NPMIMEType) NPN_MemAlloc(
                    strlen(type)+1);
    mimetype[strlen(type)] = 0;
    if (mimetype)
        strcpy(mimetype, type);
         // possible buffer overflow
```

Another example of a false positive is as follows. The if-statement at line 445 guarantees that `pFrameWork` is only non-null when `pDoc` is non-null and therefore `pDoc` cannot be null at line 481.

```
428: ::sd::FrameView* pFrameView = NULL;
429: ::sd::DrawDocShell* pDocSh =
        PTR_CAST(::sd::DrawDocShell,
                 SfxObjectShell::Current()
                );
430: SdDrawDocument* pDoc = NULL;

439: if (pDocSh)
440: {
441:     pDoc = pDocSh->GetDoc();

445:     if( pDoc && eDocType ==
             pDoc->GetDocumentType() )
446:         pFrameView =
             pDocSh->GetFrameView();
451: }

479: UINT16 nDefTab = 0;
480: if( pFrameView)
481:  nDefTab = pDoc->GetDefaultTabulator();
      // possible null pointer deref
```

2.1.4 Inconclusive warnings

Warnings that were inconclusive were ones where an unsafe function was used, but no evidence of its misuse was found in that same function. This meant that either the variable was not used at all, or was used elsewhere in the program. An example is provided below.

The system call `getenv` retrieves an untrusted environment variable and stores it in a string with the appropriate size. There is no check for its length because it is unclear what the maximum length should be as it is returned from the routine to be used elsewhere. Thus this warning was inconclusive from analysing this function alone.

```
   // check whether we have an environment
   // variable which helps us
   const char* pProfileByEnv =
     getenv( ProductRootEnvironmentVariable[
             productIndex ] );
   if ( pProfileByEnv )
   {
      sProductPath = ::rtl::OUString(
         pProfileByEnv,
         rtl_str_getLength( pProfileByEnv ),
         osl_getThreadTextEncoding() );
   // assume that this is fine,
   // no further checks
   }
   else
   ...
   s_productDirectories[ productIndex ] =
         sProductPath;
}

return s_productDirectories[ productIndex ];
```

Although many static analysis tools, including the ones used here, suffer from high false positive rates due to the limited reasoning of the program semantics, six buffer overflow vulnerabilities and five null pointer dereference errors were found. This indicates that these tools are still useful and can be used to provide an initial set of points from which more thorough analysis could be conducted. Static analysis tools found real vulnerabilities that were easy to locate and diagnose when they were local to a single function.

2.2 Fuzzing

Fuzzing as a method of vulnerability discovery and software testing is widely used in a large range of applications. The simplest form of fuzzing involves sending a stream of random input to software, either as files, network packets or arguments. Another common technique is mutating existing input, e.g. from a test suite, by randomly flipping bits. Since fuzzing often generates invalid input, it is good at testing error-handling routines. However, one main limitation of fuzzing is that exhaustive testing of an application's input space quickly becomes infeasible.

Specification-based fuzzers make use of prior knowledge of the input data structure to provide more extensive coverage by limiting the randomness in the data to format-specific components. An example of such a fuzzing framework is Peach [6]. The main disadvantage of this technique is that the specification, which must be defined prior to fuzzing, is sometimes propriety or complex to define.

We used mutated input fuzzing which is lightweight; it is able to find severe bugs; there are no false positives – every crash is a bug; and the test input is available. The source code of our simple fuzzer is given below.

```
int main (int argc, char **argv)
{
   FILE *in, *out;
   int num_mods;
   int mod_rate = 8; // 8%

   in  = fopen (argv[argc-2], "r");
   out = fopen (argv[argc-1], "w");

   srand( time(NULL) );

   while (!feof (in)) {
#define BUF_SIZE 1024*1024
      char buffer[BUF_SIZE];
      size_t len;

      len = fread (buffer, 1, BUF_SIZE, in);
      num_mods = (len * mod_rate)/100;

      for (int i = 0; i < num_mods; i++)
      {
        int offset = rand()%len;
        buffer[offset] = rand()%256;
      }
      fwrite (buffer, 1, len, out);
   }
   fclose (out);
   fclose (in);
   return 0;
}
```

We focused primarily on the Microsoft Office formats (.doc, .xls and .ppt) based on the high number of previously disclosed vulnerabilities. The seed files were randomly downloaded from Google and Yahoo. We used 2 ways to obtain more feature-rich files, namely selecting files based on size, and using search terms to find files that contained specific features, e.g. "pivot table filetype:xls" or "visual basic macro filetype:doc". In total, we used 10 seed files, ranging from a simple "Hello World" 9kb file to a large 6 MB PowerPoint file containing images of different formats.

Our fuzzer took a valid input file, f, and randomly modified 8% of bytes to produce f'. We found the corruption ratio of 8% to be ideal–any lower and there were very few crashes; higher ratios were too destruct and files were more likely to crash very early in the execution. If opening f' caused a crash, the offending byte was located by incrementally transforming f' back to the original f until the byte or bytes that caused the crash were isolated. The algorithm was as follows.

1. Compare f' with f and extract the set of modified byte locations. Let the set of location and corresponding byte value pairs be $S = \{(l_0, b_0), (l_1, b_1), \ldots, (l_{n-1}, b_{n-1})\}$. Let the transformation function be T, where $f' = T(f, S)$. In other words, to obtain f' from f, we transform f based on the location–byte value pairs in S.

2. If S contains 1 or less elements or if number of iterations reaches the limit, stop.

3. Randomly divide S into 2 half sets, S_0 and S_1, each containing $n/2$ elements.

4. Generate 2 new files, $f'_0 = T(f, S_0)$ and $f'_1 = T(f, S_1)$.

5. Open f'_0 and f'_1 in OpenOffice.

6. If f'_0 causes a crash, assign $S = S_0$ and go to step 2.

7. Else if f'_1 causes a crash, assign $S = S_1$ and go to step 2.

8. Else neither file causes a crash or both cause a crash. Keep the current S and go to step 2.

The Gnu debugger, GDB, was then used to analyse the crash.

2.2.1 Results of fuzzing

There were no false positives as every crash was a bug, but not every bug was a vulnerability. In total, we found

- one out-of-bounds array index vulnerability,
- one null-pointer exception, and
- one unchecked pointer dereference vulnerability.

2.2.2 Debugging

The procedure we used to debug crashes was to repeat an execution in order to reverse-trace the execution up the call stack. This involved

1. stopping execution at an initial instruction, either the crash point or a breakpoint,

2. determining the caller function,

3. setting a breakpoint in the caller, and

4. re-running the program with the new breakpoint.

This is a trial-and-error process as both caller and the called functions may be executed multiple times and identifying the correct iteration of the breakpoint is often hard. [1]

The main factor that made debugging OpenOffice difficult was the use of complex data structures.

2.2.3 Complex data structures

As an illustration, we will use an out-of-bounds array index vulnerability in `SdrPowerPointImport::ApplyTextObj`.

The segmentation exception occured in the function `ImpEditEngine::SetStyleSheet` due to an invalid instruction pointer address. The error was caused by the index `mnDepth` having too large a value for the array `pStyleSheetAry`. The vulnerability can be summarised by the function call tree given below. The call tree, which was obtained through debugging, shows where

1. the `mnDepth` variable is read,
2. the `pStyleSheetAry` array is defined,
3. the out-of-bound index error occurs, and
4. the location of the crash.

```
SdrEscherImport::ProcessObj
|-PPTTextObj::PPTTextObj
| |-PPTStyleTextPropReader::
|   PPTStyleTextPropReader
|   |-PPTStyleTextPropReader::Init
|     |-PPTStyleTextPropReader::
|       ReadParaProps-------------------(1)
|
|-SdrPowerPointImport::ReadTextObj
  |-ImplSdPPTImport::ApplyTextObj-------(2)
    |-SdrPowerPointImport::ApplyTextObj-(3)
    |-ImpEditEngine::SetStyleSheet------(4)
```

To diagnose this error, we needed 2 pieces of information – the bound on `mnDepth` and the bound on `pStyleSheetAry`. The 2 structures involved in this error were `PPTParaPropSet` and `SfxStyleSheet`. `PPTParaPropSet` contained a pointer to the `ImplPPTParaPropSet` structure, which in turn had a member `mnDepth`, a 16-bit unsigned integer or 2 bytes in the input file. The fuzzed file gave us the location of one of the bytes, and trial-and-error gave us the other.

```
struct PPTParaPropSet
{
...
    ImplPPTParaPropSet* pParaSet;
...
};

struct ImplPPTParaPropSet
{
    sal_uInt16  mnDepth;
...
};
```

Next, we needed to locate the bound on `pStyleSheetAry`. The 4 relevant function snippets are listed below.

```
ReadParaProps(SvStream& rIn, ...){
   // mnDepth is read
1: rIn >> aParaPropSet.pParaSet->mnDepth;
   ...
}

ImplSdPPTImport::ApplyTextObj(...){
   // Array is defined
2: SfxStyleSheet* pStyleSheetAry[ 9 ];
3: ppStyleSheetAry = &pStyleSheetAry[ 0 ];
4: SdrPowerPointImport::ApplyTextObj(...,
                         ppStyleSheetAry );
}

SdrPowerPointImport::ApplyTextObj(...,
   SfxStyleSheet** ppStyleSheetAry){
   // Out-of-bound index
5: SfxStyleSheet *pS =
     ppStyleSheetAry[aParaPropSet.pParaSet
                    ->mnDepth];
6: if( pS )
7:   rOutliner.SetStyleSheet( ..., pS );
   ...
}

SetStyleSheet( ..., SfxStyleSheet *pStyle ){
8: if( pStyle )
     // Crash
9:   aNewStyleName = pStyle->GetName();
   ...
}
```

From the definition of the array in line 2, the bound of the index was 8 and the error occured when the variable `mnDepth`, which had an upper bound of `0xffff`, was greater than this value. The error caused the program to access out-of-bounds memory at line 9.

This vulnerability was difficult to diagnose because of the complex data structure of PPTParaPropSet, which resulted in the need to analyse 4 functions, which were located in different modules.

File name	size	n
hello_world.doc	9kb	13925
gif.doc	10kb	16311
text.ppt	50kb	7363
images.doc	700kb	18365
balmer.ppt	6Mb	25527

Table 3: Number of tainted conditional jumps, n, for different file sizes and types

2.3 Multiple path exploration

The third method that we used to find vulnerabilities was exploring paths using the taint analysis mechanism of Flayer [5], a fuzzing tool.

Taint analysis [14] is a well-known technique to analyse data flow in a program, particularly data originating from a particular source, such as an input file. Flayer logs conditional branches in the program that are dependent on tainted data. Let a taint trace based on an input file f contain n conditional branches $b_i, i \in \{0, \ldots, n\}$, and b_i can be 1 or 0 corresponding to whether it is taken or not. Fine-grained taint analysis can reveal which bytes in f affect the value of b_i to produce f' which causes the alternate branch, b'_i, to be taken. Table 3 lists the n for different input files.

In practice, the number of b_i was not small and the following heuristic was used to explore areas that were more likely to contain errors.

- Code complexity. Explore b_i's located in more complex parts of the program. Complexity can be determined by metrics such as number of lines of code or number of linearly independent paths [12].

- Distance to other known errors. If an error was previously found in a location l, choose b_i's near to l. Nearness of a b_i to l can be determined by their relative position in a taint trace, or by their relative distance in source code.

The algorithm we used was as follows. Firstly, we obtained a taint trace of an input file f to obtain the conditional branches $b_i, i \in \{0, \ldots, n\}$. We then determined b_i's that were located in our target function using a stack trace. Let's call the first branch in the target function $b_k, 0 \leq k \leq n$. Fine-grained taint analysis was used to determine the bytes that affected the value of the b_k, and the alternate branch was calculated manually to determine f'. If f' caused a crash, GDB was used to diagnose the error. If not, the next branch b'_{k+1} was selected to generate f'' and so on.

2.3.1 Results of multiple path exploration

We explored different paths in `ImpPeekGraphicFormat`, a function containing 425 lines of code. `ImpPeekGraphicFormat` is the function that parses the first few bytes of an image to decide the image format and subsequently the image parser to use. We chose this function as parsing graphics has been known to cause a number of previously disclosed vulnerabilities.

We found a memory out-of-bounds error in WMF-Reader::ReadRecordParams using this method. The taint sources are `pnPoints[0]` and `pnPoints[1]`; the taint sink is a call to `malloc` (line 10). The sum of `pnPoints[0]` and `pnPoints[1]` in line 9 causes the size of the buffer `pPtAry` to be unexpectedly small. This causes `memcpy` to crash because of an out-of-bounds read in line 11. In addition, it is possible for the variable `nPoints` to overflow in the second iteration of the loop (line 9) if `pnPoints[0] + pnPoints[1]` is more than $2^{16} - 1$.

```
1:  USHORT  i, nPoly, nPoints;
2:  USHORT* pnPoints;
3:  Point*  pPtAry;
4:  *pWMF >> nPoly;
5:  pnPoints = new USHORT[ nPoly ];
6:  nPoints = 0;

7:  for( i = 0; i < nPoly; i++ ){
8:      *pWMF >> pnPoints[i];
9:      nPoints = nPoints + pnPoints[i];
    }
10: pPtAry  = (Point*) new char[nPoints *
                          sizeof(Point)];
    ...
11: memcpy( mpPointAry, pPtAry,
      (ULONG)nInitSize * sizeof(Point) );
```

The main advantage of manual path exploration was the flexibility to choose the branches to explore. The challenge was manually generating and solving path constraints.

However, analysing deeper semantics requires an automated constraint generator and analyser such as KLEE [3]. Nevertheless, we were able to successfully find errors this way.

2.4 Other tools

Besides the 3 main methods, we also tried MemCheck [25], a Valgrind tool for detecting memory-related errors, such as use of uninitialised memory and memory leakage. MemCheck detected 2014 errors. We did not investigate these errors as we were focusing on security vulnerabilities. We note, however, that Valgrind has been used to build other vulnerability detection tools [14].

We also considered BOON [27], a buffer overflow detector for C, MOPS [23], a model checker for C, and CQUAL++/Oink [4] a static analysis tool for C/C++. The first 2 tools were primarily for C and thus could only analyse 8% of OpenOffice code. We were unsuccessful in using the last tool. [2]

3 Discussion

In this section, we argue that the community needs better automated analysis tools for large software. We summarise three features of OpenOffice that make it complex to analyse and propose some possible strategies to handle such complexity.

First, data structures in OpenOffice are large and complex. The number of classes and data structures used in parsing a Word document alone exceeds 500. This includes structures to store text, drawings, images, styles, tables, fonts and so on. Moreover, parsing of these structures occurs in different modules. For example, drawing and tables are handled by the "svx" module; images are handled by "svtools"; the input file stream is handled by the "sot" module; the main parser for Word documents is the "sw" module.

Second, the number of dynamically loaded libraries can be large. Callgrind, a call graph profiler, easily runs out of memory on longer executions [21]. Static tools need to be able to handle the large number of dependencies. As an example, the main Word document parser file, ww8par.cxx, alone has 103 include directives, not including secondary and tertiary dependencies.

Third, OpenOffice has a number of use scenarios besides the commonly known word processing (Writer), slideshow creation (Impress) and spreadsheet creation functionalities (Calc). Other functionality includes:

- OpenOffice supports over 30 other file formats from HTML to PDF files to MathML formula formats [19];
- It can be used to open and view documents as a plugin to the Firefox browser;
- Runtime behaviour of OpenOffice can be altered via about 30 different environment variables, including definitions for locations of temporary files and automatic update URLs [18];
- It allows the use of extensions to further customise user interface via both the StarBasic scripting language and native code [20];
- OpenOffice can be started in server mode and run as a service via the "headless" option.

3.1 Possible strategies

Automatic test input generation is essential if we are to systematically test large parts of the software. Therefore, a tool has to be general enough to flexibly handle all these testing scenarios.

One possibility is that such a tool could be built on top of dynamic binary instrumentation frameworks. Frameworks such as Valgrind [13], Qemu [2], Bochs [26] and Pin [9] have become increasingly popular in recent years. Such tools are advantageous in that the source code is not needed, no re-compilation of the program is needed and analysis is based on what actually gets executed. However, since these are software frameworks, the analysis overhead is naturally high. Moreover, the question of what information to capture and what to discard is important.

Dynamic taint analysis is a method that defines propagation rules to extract information flows arising from a designated taint source. This is useful for isolating flows from files or network packets. A common strategy is to taint only explicit, or direct, flows and discard control flow dependencies and pointer aliasing. There are shortcomings to this propagation rule as some errors are only evident when considering indirect flows. The difficulty, however, is that tainting control flow dependencies and pointers inadvertently leads to "taint explosion". Thus far there is no feasible solution to deal with this problem.

Capture-replay systems, such as Chronicle Recorder [16], can be used to record the entire execution trace in a compact form and thereafter several offline analyses can be performed concurrently on the data. This method is well-suited to extend taint analysis to a more general tag analysis where richer metadata, e.g. time accessed, security tags, can be tagged to input data.

Due to the large amount of data generated by execution of large programs, visualisation tools can aid analysts in processing and reasoning about this information. For example, visual representations of dynamic data flow through a program can provide analysts with an interface to correlate data or detect anomalous flows. Ball and Eick [1] describe four visual representations that scale to large software systems. An example of how visualisation can aid in reverse engineering of malware is by Quist and Liebrock [22].

4 Related Work

There is relatively little published research on the security testing of large programs or large-scale software. As far as we know, there are two previous work in this area.

The MOPS model checker [23] was used to check 60 million lines of C code in the Red Hat 9 distribution. MOPS could find race conditions, format string errors and strncpy errors among others, and it achieved scalability through checking only relevant operations and through integration with build processes.

Chen and Wagner [4] found 1,533 possible format string vulnerabilities after analysing 66% of the Debian Linux distribution using the Oink-CQual++ static analysis tool. To address scalability, virtual machines were used to intercept the .i intermediate files for analysis, and optimization techniques were used to reduce RAM usage of the algorithms.

5 Conclusions and future work

We have shown that OpenOffice is not error-free and have demonstrated several severe vulnerabilities that we found over a 4-month period through the use of publicly available tools. This work was undertaken as a training exercise by the first two authors; it was motivated in part by the lack of published research on the amount of effort (or cost) involved in finding real zero-day vulnerabilities in widely-deployed large-scale software systems. After an initial period of some weeks spent on learning the target software and the analysis tools, vulnerabilities were found at a steady pace.

It has been argued that free and open-source software such as OpenOffice may be more secure than proprietary systems such as Office since 'to many eyes, all bugs are shallow'. This may well be true, but there is also a matter of incentives: how many people who read the OpenOffice code are looking for vulnerabilities, and how many are looking for a chance to add new features? Our work suggests that the total security testing effort invested in OpenOffice to date may not be large. Certainly the marginal cost of finding a new vulnerability, once the code and the tools have been mastered, is at present less than a month of effort.

Our first conclusion is thus a security-economics one: we need to find ways to get people to invest more time and effort into finding and fixing vulnerabilities in free and open-source software that millions of people use on a daily basis.

A second lesson learned from this exercise is that security testing of such large software systems is made more difficult by its complexity, which manifests itself in complexity of data structures, of code and of use cases. Our second conclusion is thus a technical one: a valuable research project would be the development of automated tools that are simple to use and extensible.

Our third point is a methodology issue for discussion by the free and open-source software community. Is there some way we can modify our methods of working, culture and discipline so that we design large complex software systems for testability from the start?

Our fourth point is practical. We have studied only a small part of the OpenOffice code; we are sure that many vulnerabilities remain to be found and fixed. We encourage other researchers to continue this challenge. First, it is instructive to find real vulnerabilities in real systems, and we'd recommend the experience to novice (or even experienced) security researchers. Second, it's desirable that vulnerabilities in widely-used products such as OpenOffice be found and fixed. The main problem we encountered was that it took us several weeks to get started on security testing; understanding the target code (and even getting it to compile) took a fair bit of effort. We have therefore made our tools available, with a guide, so as to jumpstart future effort in testing the OpenOffice suite. These materials are available at

`http://www.cl.cam.ac.uk/~wmk26/openoffice`

6 Acknowledgements

The authors would like to thank OpenOffice developers Fridrich Strba and Thorsten Behrens for helping to validate the errors, and to Joseph Bonneau and other anonymous reviewers for their valuable comments.

References

[1] BALL, T. A., AND EICK, S. G. Software visualization in the large. In *IEEE Computer 29(4):33-43* (1996).

[2] BELLARD, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference* (2005).

[3] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 10th Symposium on the Operating System Design and Implementation (OSDI 2008)* (2008).

[4] CHEN, K., AND WAGNER, D. Large-scale analysis of format string vulnerabilities in debian linux. In *Proceedings of the ACM SIGPLAN workshop on programming languages and analysis security (PLAS'07)* (2007).

[5] DREWRY, W., AND ORMANDY, T. Flayer: Exposing application internals. In *Proceedings of the 1st USENIX Workshop on Offensive Technologies (WOOT'07)* (2007).

[6] EDDINGTON, M. Peach. `http://peachfuzz.sourceforge.net`.

[7] FREE SOFTWARE FOUNDATION. Gdb and reverse debugging. `http://www.gnu.org/software/gdb/news/reversible.html`.

[8] HUMMER, T. Webanalyse: Openoffice auf über 21% der computer. `http://www.webmasterpro.de/portal/news/2010/01/25/verbreitung-von-office-programmen-openoffice-ueber-21.html`, 2010.

[9] LUK, C., COHN, R., PATIL, R., KLAUSER, H., LOWNEY, A., WALLACE, G., REDDI, S. V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005).

[10] MANN, J. Firefox user base swells to 270 million. http://www.techspot.com/news/34599-firefox-user-base-swells-to-270-million.html, 2009.

[11] MARJAMKI, D. Cppcheck - a tool for static c/c++ code analysis. http://cppcheck.wiki.sourceforge.net/.

[12] MCCABE, T. J. A complexity measure. In *IEEE Transactions on Software Engineering Vol. 2, No. 4, p. 308* (1976).

[13] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Programming Language Design and Implementation (PLDI'07)* (2007).

[14] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS'05)* (2005).

[15] NIST. National vulnerability database. http://nvd.nist.gov.

[16] O'CALLAHAN, R. Chronicle-recorder. http://code.google.com/p/chronicle-recorder, 2007.

[17] OPENOFFICE STATS PROJECT. Openoffice stats project. http://stats.openoffice.org/, 2007.

[18] OPENOFFICE.ORG. Environment variables. http://wiki.services.openoffice.org/wiki/Environment_Variables.

[19] OPENOFFICE.ORG. File formats. http://wiki.services.openoffice.org/wiki/Documentation/OOo3_User_Guides/Getting_Started/File_formats.

[20] OPENOFFICE.ORG. Openoffice extensions. http://extensions.services.openoffice.org/.

[21] OPENOFFICE.ORG. Callgrind. http://wiki.services.openoffice.org/wiki/Callgrind, 2009.

[22] QUIST, D. A., AND LIEBROCK, L. M. Visualizing compiled executables for malware analysis. In *Proceedings of the 6th International Symposium on Visualization for Cyber Security (VizSec 2009)* (2009).

[23] SCHWARTZ, B., CHEN, H., WAGNER, D., MORRISON, G., AND WEST, J. Mops: An infrastruture for examining security properties of software. In *Proceedings of the ACM conference on computer and communications security (CCS'02)* (2002).

[24] SECUNIA. http://secunia.com.

[25] SEWARD, J., AND NETHERCOTE, N. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference* (2005).

[26] THE BOCHS PROJECT. Bochs: The open source ia-32 emulation project. http://bochs.sourceforge.net.

[27] WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Network and Distributed System Security Symposium (NDSS'00)* (2000).

[28] WHEELER, D. A. Flawfinder. http://www.dwheeler.com/flawfinder/.

[29] WILKERSON, D. S., AND CHEN, K. Build interceptor. http://build-interceptor.tigris.org.

Notes

[1]Reverse debugging, which is a capability recently added to the GDB debugger [7], addresses this issue by providing program snapshots to restore previous execution states. However, at this time of writing multi-threaded programs are not supported.

[2]The prerequisite for using CQUAL++/Oink was the generation of the gcc intermediate .i and .ii files. We tried using GCC's "save temps" option but were not successful. We did not try using Build-Interceptor [29].