CS136 - Security Review of ispell

Report By

Name	UID
Kaela Polnitz	504751829
Hannah Nguyen	304787645
Asavari Limaye	605224431
Dennis Zang	704766877
Davis Gomes	104907446

Table of Contents

Summary

To evaluate the security of ispell, we identified and divided up review strategies, and reconvened to discuss the different issues we discovered, identifying ones we found in common and decided which approach to use to research the vulnerability further. Approaches we considered included: static automated code analysis, manual code review, live and fuzz testing, and researching known security issues and reported bugs.

We discovered and evaluated a wide range of issues, and while we do not believe that any of them are imminently fatal security flaws for the system, rendering it completely unusable and unable to recover from, they should be fixed as soon as possible, to prevent the possibility that

attackers will discover a way to exploit these vulnerabilities to reduce or prevent the usage of the system as intended. Security vulnerabilities we discovered include: segmentation faults and buffer overflows as a result of bad user input, possibility of symlink attacks, buffer overflows, null references, memory leaks, file access, buffer copying, and unverified string manipulation.

Original Plan For Security Evaluation

Group Meeting 1

Sunday, 17 November, 3:00 - 5:00 PM
During this meeting, we identified the different methods for evaluating the security of ispell. From this discussion, the potential strategies to focus on were:

1. Automated Static Code Analysis
2. Manual Code Review
3. Fuzz Testing
4. Online Search for Common & Reported Bugs
5. Live Testing of User Inputs
6.

These strategies were then divided among the team members:

Name	**Strategies**
Kaela Polnitz	Static Automated Code Analysis
Hannah Nguyen	Common Issues & Manual Code Review
Asavari Limaye	Live Testing, Fuzz Testing, User Inputs
Dennis Zang	Live Testing, Manual Code Review
Davis Gomes	Online Search for Common & Reported Bugs

We also brainstormed and compiled a list of common security issues which ispell may have which need to be looked for.

- Buffer overflows
- Special characters in input files
- File size
- Sensitive data stored in temporary files
- Flags/options
- Backups
- Privacy
- Access control
- Encryption, key storage
- Programs and tools that ispell calls

Group Meeting 2

Thursday, 21 November, 2:00 - 5:00 PM
In this meeting, each member shared their preliminary observations and findings from the strategy they had followed to evaluate the security of ispell. These preliminary findings were compiled together to decide the areas of focus and the next steps.

A spreadsheet was created to track the different security issues encountered, how to recreate this, code snippets, tools used, screen shots/reports and a brief analysis of the security implications of this issue. Each member added the security issues they had found by the preliminary evaluation, and continued adding to this spreadsheet after this meeting.

Group Meeting 3

Sunday, 24 November, 2:00 - 6:00 PM
Before this meeting, each member had completed their analysis and added the discovered issues to the spreadsheet. During this meeting, the format of the report was decided and each of the security issues was manually looked into through manual code review and further live testing. The results from the different approaches were compiled and a final summary and the results of the security evaluation of ispell were compiled.

These number of meetings, their agenda and the process of evaluation were decided during the first meeting and were mostly followed. Submission of the project was delayed by two days due to delays in completing the report. Although each member picked different strategies in the beginning, during the last meeting all the members reviewed the code manually and used different techniques for evaluation.

Tools and Approaches Used

Static Automated Code Analysis

For the automated tools review, we used a few different tools to draw conclusions about the integrity and security of iSpell.

The first tool we used was Flawfinder. We chose this tool because it integrated well with Python and it revealed a variety of different vulnerabilities within the iSpell program. To obtain results, we used the following command:

```
flawfinder ~/Downloads/ispell-3.4.00
```

After this command, Flawfinder generated a number of warnings and issues with iSpell, as seen from this summary:

```
ANALYSIS SUMMARY:

Hits = 387
Lines analyzed = 15148 in approximately 0.27 seconds (55335 lines/second)
Physical Source Lines of Code (SLOC) = 10014
Hits@level = [0] 212 [1]  65 [2] 108 [3]  22 [4] 186 [5]   6
Hits@level+ = [0+] 599 [1+] 387 [2+] 322 [3+] 214 [4+] 192 [5+]   6
Hits/KSLOC@level+ = [0+] 59.8163 [1+] 38.6459 [2+] 32.155 [3+] 21.3701 [4+] 19.1732 [5+] 0.599161
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

The next tool we decided to use was Splint. Setting up this tool was also very simple and it was used with the following command:

```
splint -preproc [c-file]
```

Given that it was a more manual tool (i.e could not be called on the entire directory), the output was lengthy, but detailed. In total, over 1000 vulnerabilities surfaced from the thorough search. Below is an example of what this output looked like:

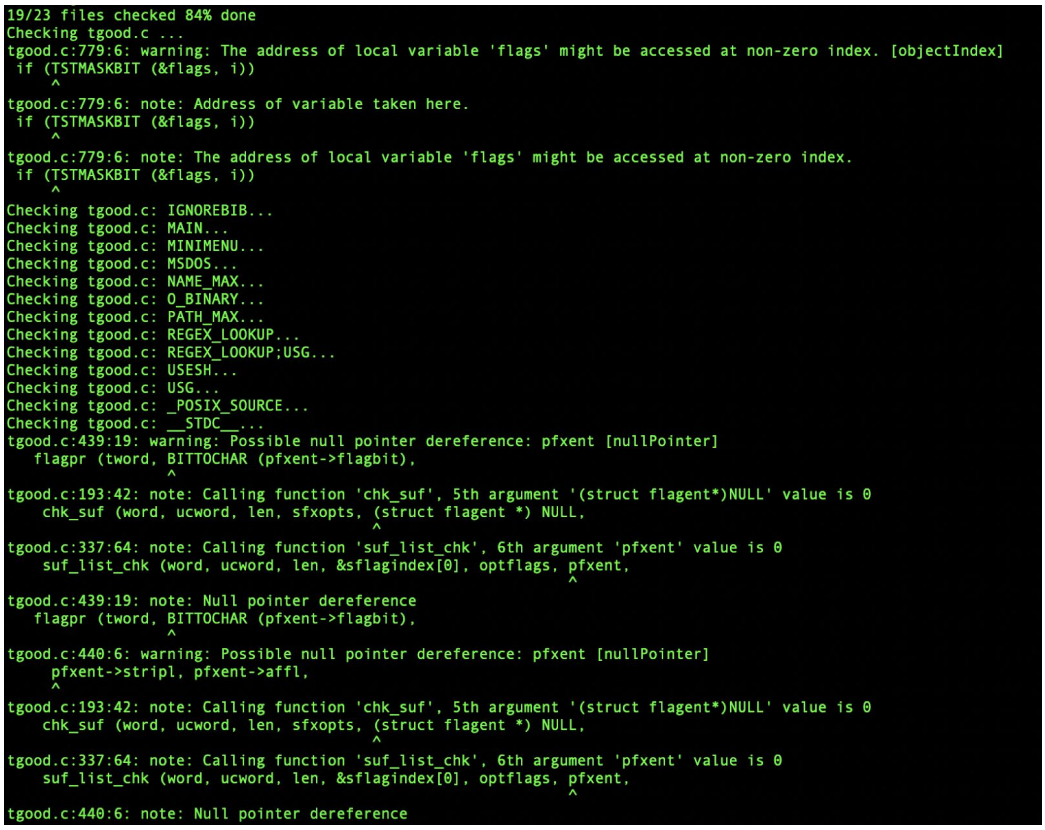
```
Splint 3.1.2 --- 15 Oct 2019

config.h:221: Include file <sys/types.h> matches the name of a POSIX library,
    but the POSIX library is not being used.  Consider using +posixlib or
    +posixstrictlib to select the POSIX library, or -warnposix to suppress this
    message.
  Header name matches a POSIX header, but the POSIX library is not selected.
  (Use -warnposixheaders to inhibit warning)
proto.h:306: Include file <unistd.h> matches the name of a POSIX library, but
    the POSIX library is not being used.  Consider using +posixlib or
    +posixstrictlib to select the POSIX library, or -warnposix to suppress this
    message.
xgets.c: (in function xgets)
xgets.c:101:51: Observer storage assigned to unqualified reference:
                  char * Include_File = "&Include_File&" = "&Include_File&"
  Observer storage is transferred to a non-observer reference. (Use
  -observertrans to inhibit warning)
   xgets.c:101:34: Storage becomes observer
xgets.c:114:6: Observer storage env_variable assigned to unqualified reference:
                 Include_File = env_variable
   xgets.c:113:22: Storage env_variable becomes observer
xgets.c:115:2: Assignment of size_t to int: Include_Len = strlen(Include_File)
  To allow arbitrary integral types to match any integral type, use
  +matchanyintegral.
xgets.c:123:2: Assignment of char to int: c = '\0'
  A character constant is used as an int. Use +charintliteral to allow
  character constants to be used as ints.  (This is safe since the actual type
  of a char constant is int.)
xgets.c:128:10: Operands of != have incompatible types (int, char): c != '\n'
xgets.c:137:14: Null storage returned as non-null: (NULL)
  Function returns a possibly null pointer, but is not declared using
  /*@null@*/ annotation of result.  If function may return NULL, add /*@null@*/
  annotation to the return value declaration. (Use -nullret to inhibit warning)
xgets.c:147:36: Function strncmp expects arg 3 to be size_t gets unsigned int:
                  (unsigned int)Include_Len
xgets.c:2:13: File static variable Rcs_Id declared but not used
  A variable is declared but never used. Use /*@unused@*/ in front of
  declaration to suppress message. (Use -varuse to inhibit warning)
```

For the last tool, we decided to use CPPCheck. This was a tool that we found via Google Search; it gave me an idea of common tools that were used for static C/C++ code analysis. With this tool, we were able to do a recursive static code check of the iSpell folder with the following command:

```
cppcheck --force --enable=information .
```

With this tool, there was a lot of useful output as well. Below, there is a screenshot of what this looked like:



Manual Code Review

Manually reviewing the source code of ispell had to be done at a higher level given the time constraints of this security evaluation and the work breakdown our team decided on. Taking a look at the hierarchy of the codebase was useful to understanding how the code was organized, and the related functionalities of the files grouped together. This helped determine which files were the most important to focus on, and where security vulnerabilities were most likely to occur. Notes on the codebase hierarchy and individual files can be found in the appendix.

After looking into every single file in the codebase, skimming comments and major functions (which is what ispell is primarily written in), we identified the following files as the most important to focus on: buildhash.c, tree.c, xgets.c, hash.c, makedent.c, makedict.sh, good.c, and dump.c. Doing Internet research on common security vulnerabilities in C helped narrow down additional locations to look for issues: in particular, functions that accessed files, reading or copying buffers, and manipulating strings.

Once these priorities were identified, code review was a matter of searching and reading through the source code. This sometimes required a bit of jumping around function calls and in between files in order to understand the datapath flow of the code execution, but narrowing down ahead of time what security issues to be on the lookout for, as well as major entry-points into the system via the important files, were time-efficient strategies to do manual code review.

Live Testing and Fuzz Testing

Importance of Live Testing

Although analysing source code is a more comprehensive method of evaluating software, it is time consuming and can be difficult to carry out efficiently. Code bases are often very large, and it can be difficult to prioritize different modules for in-depth manual or automatic code review.

The most important errors in implementation or security are the ones which are visible to the user of the software. These are the most likely to be exploited as no knowledge of the source code is required to discover them, and can be exploited by a large number of people if the software is made available through a public interface.

Live testing of the software refers to finding errors in the code dynamically by observing it and interacting with it while it runs. Fuzz testing refers to using inputs which have been modified randomly to a program to check if the program handles invalid input correctly without creating any security issues. Some of the security vulnerabilities that can be discovered through fuzz testing are buffer overflows, SQL injection and Denial of Service.

The service being set up allows the user to upload files of their choice to the server, which are then sent to ispell. This means that the user controls the contents of the files being used as input to ispell, and this is a possible way by which a malicious user may try to attack the server. Thus, it is essential to perform live and fuzz testing on ispell using different input files to ensure that it securely handles the case where the files uploaded are invalid.

Plan for Live Testing

The plan for live testing ispell is outlined below:

1. Install ispell
2. Use ispell in different ways
3. List the different inputs to ispell

4. Find appropriate tools for live testing
5. Find appropriate tools for fuzz testing
6. Perform fuzz testing on each of the input methods
7. Perform live testing on different ways of using ispell

Installation of ispell

To perform live and fuzz testing on ispell, it needs to be installed.

System Details:
Operating System: Ubuntu 18.04.2 LTS / Ubuntu 18.04.3 LTS
Version of ispell used: ispell-3.4.00
Compiler: cc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0

The following steps were used to install ispell:

1. Installing Dependencies:
    a. sudo apt install bison flex
    b. sudo apt-get install libncurses5-dev
2. tar -xvf ispell-3.4.00.tar.gz
3. cd ispell-3.4.00
4. cp local.h.linux local.h
5. make all
6. make install

Since ispell depends on bison and ncurses, its security is dependent on these too. Security issues of these and other dependencies are not analysed and must be done as future work.

Inputs and Usage modes

Ispell can be used in two modes:

1. Interactive Mode
2. Non Interactive Mode

The specifications for the use of ispell by the web service does not specify how the web interface will allow the user to interact with ispell. It may use the ispell non interactively, and provide an additional web interface instead. However, due to limitations of the fuzzing tools, only the non-interactive mode was used for fuzz testing.

To use ispell in the non interactive mode, we simply need to pass the file through stdin (usually by pipelining).

The inputs to ispell are

1. The input file: This is the input text file which contains ASCII characters that is to be checked for spelling errors. This is specified on the command line.

2. The dictionary file: This is the main dictionary used by ispell. This must be converted to a hash file using the tool buildhash, and then can be specified on the command line with the -d option.
3. The personal dictionary: This contains additional words apart from the main dictionary. This does not need to be converted into a hash. This is specified with the -p option on the command line.

These are the three inputs which can be specified by the user. The specification mentions that the user provides the file to be uploaded. It does not mention if the user will be able to supply their own dictionaries and personal dictionaries through the web interface as well. Since this is needed to be able to upload files in different languages, or to add certain words to the default dictionary, it is likely that this feature will be provided by the web service, and hence it is taken into consideration as a possible input for fuzzing as well.

Live Testing Tools

The live testing tool used for evaluating the security is the address sanitization feature of the C compiler. This used to check if the access to memory are valid. For example, it can find cases where a part of memory has been freed or is unallocated but an access to it is made. Without address sanitization, some of these illegal memory accesses can go unnoticed. For example, accessing memory outside the limits of an array is allowed by C, but is a security vulnerability.

The address sanitization can be enabled using the compiler flags CFLAGS "-fsanitize=address" while compiling the source code for ispell. When the program is run, any invalid accesses are found. A buffer overflow in the heap was found using address sanitization. This was in the buildhash tool, in the function filltable.

Fuzz Testing Tools

Fuzzing tools are used to modify valid input randomly in different ways to a program to discover errors.

Zuff

The tool “zzuf” was used to perform fuzz testing on the input file, personal dictionary, and main dictionary, while using ispell in the non interactive mode. zzuff allows input files to be modified by changing bits in the file in a random way, depending on a seed value. Using the same seed value for the same file, will always give the same fuzzed file, so that the modification is repeatable.

These modified files can be used as input to ispell and any crashes of ispell are logged. A sample text file was modified, and random bytes were also used as input. Using a file with random bytes and a modified files results in a segmentation fault in ispell. Although the specification says that the file is processed and only ASCII characters are allowed in the file

when it is submitted to ispell, in case the user finds a way around this, then it is very likely that ispell will crash with a segmentation fault. The web server needs to handle this case carefully.

Fuzzing Personal Dictionary

The personal dictionary was fuzzed and used as the input to ispell. It was observed that an invalid personal dictionary file was handled by ispell and this did not result in any segmentation faults or crashes.

Fuzzing Main Dictionary

Fuzzing the constructed hash of the dictionary and using that as the input to ispell always resulted in ispell detecting that the hash was malformed and exiting gracefully without any crashes. Using fuzzed text dictionary files as the input to the tool buildhash was also handled and did not result in crashes.

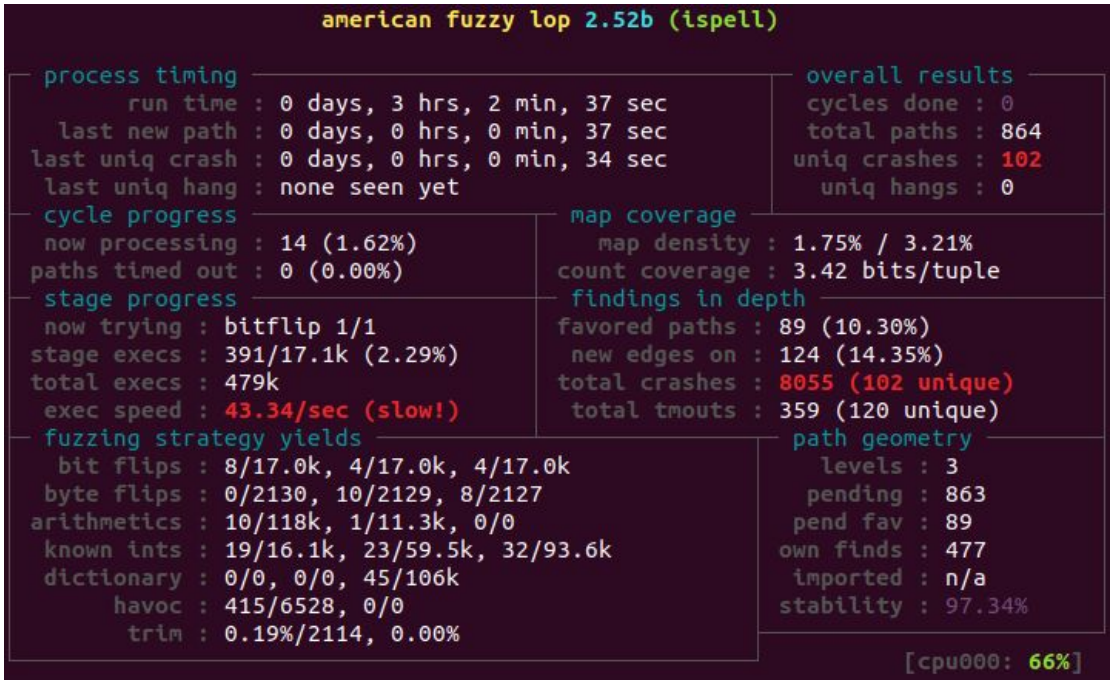
This shows that the web server must carefully ensure that the input file only has ASCII values, and handle the situation where ispell crashes with a segmentation fault. Invalid dictionary and personal dictionaries is not likely to be a problem and has been handled correctly by ispell.

American Fuzzy Lop

Another tool we used to perform live testing on ispell is American Fuzzy Lop (AFL). This uses genetic algorithms to create fuzzed inputs which are likely to find bugs and logs them. A sample input file is provided as an example and its position as input in the command is specified.

There is one thing to note when one runs AFL: the code must be compiled with the package's given compilers for runtime instrumentation. To do this, an easy way to modify the program would be to modify the value of "CC" in "config.X" before running "make all" and "make install". In addition, an input file needs to be placed in a mandatory input directory (in our case, "afl_in"), along with a mandatory output file for the hueristics. The tests we performed are as follows:

```
afl-fuzz -i afl_in -o afl_out_a_option /usr/local/bin/ispell -a
afl-fuzz -i afl_in -o afl_out_A_option /usr/local/bin/ispell -A
afl-fuzz -i afl_in -o afl_out_l_option /usr/local/bin/ispell -l
afl-fuzz -i afl_in -o afl_out_default /usr/local/bin/ispell
afl-fuzz -i afl_in -o afl_out_p_option /usr/local/bin/ispell -p @@ UTF8.txt
afl-fuzz -i afl_in -o afl_out_p_option /usr/local/bin/ispell -p ./mydict.hash
```



(Note that “mydict.hash” is generated through using the “buildhash” helper function, and “UTF8.txt” is a file of randomly generated UTF-8 characters [which are a better representation of a byte stream and what a user can input compared to ASCII].)

AFL was used to perform fuzz testing on the ispell application with the -a, -A, -p, and -l options to quickly detect any crashes due to malformed input. The program was run for about 2-3 hours for each case, and anywhere between 10 and 100 crashes were found depending on how long AFL is run (for -a , -A, and -p each, with none for -l), but all of those crashes correspond to a single segmentation fault error (likely the same one found using zzuf).

Using gcc for compilation (with the “-g” option) and gdb to debug with the inputs that caused the crashes, we find a single error in line 1711 for correct.c (“hadnl = filteredbuf[bufsize - 1] == '\n';” in function “askmode()”). Apparently, there is a conflict between the use of “xgets()” and “fgets()” in lines 1689 and 1695, which writes bytes into “filteredbuf” (regardless of what byte it is), and “strlen()” in line 1710, which reads up to the first null character. In some cases, if a file contains a null at the beginning of a line, then if the line were to be placed in “filteredbuf”, “strlen()” would return a length of 0, and accessing the last character using the above line will fail. Note that although the program will not crash if a null character is not at the beginning of a line, “strlen()” will truncate the line if this were the case.

One thing to note is that for the -d and -p options, one cannot input a corrupt dictionary file to use for spell-checking, as the program would catch on and exit the program with no errors. If one were to run with the -a and -A options, in theory it is possible to modify and corrupt the personal dictionary file, but to prove that with AFL would take a long time.

From a security standpoint, this wouldn't be too severe as this error isn't exploitable (attempts to read 1 byte out the span of the array), and aside from clean programming logic, the error wouldn't jeopardize security in any way.

Live and Fuzz Testing: Conclusions

The segmentation fault caused by invalid input file for checking the spelling is caused by a read in an invalid memory location. Since the segmentation fault is caused by a read and not a write, it is unlikely that this is a buffer overflow vulnerability which can be exploited by a user.

From live testing and fuzz testing, we conclude that:

- Ispell cannot handle null characters within files properly, so we must make sure that no null characters are present within a file.
- There is a heap buffer overflow error in the code of one of the additional tools buildhash, which may cause a security issue.
- Ispell handles invalid dictionary and personal dictionaries and exits gracefully with the right error.

Known Security Issues and Reported Bugs

We wanted to get a good picture of the known bugs and security vulnerabilities contributed to ispell across the internet. Doing a few google searches lead me down a few paths of discovery. One of the first websites we discovered to list reported bugs in ispell was the official debian report page for ispell. Unfortunately in this case, the only bug that I found applied to the current version 3.4.00 of ispell would not impact the integrity of ispell and thus would not be a security issue: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=926372

Debian Bug report logs - #926372
ispell: -a mode +nroff sets tex instead

We found another website under a University of Utah page. This website outlined multiple known bugs in ispell. However, having been written in 2008, we were not sure how up to date these bugs were and if they had been solved since then. Nevertheless we gave them a look. Through my various methods of looking for reported bugs, we had come to the conclusion that a lot of the bug reports online had to do with ispell and its relationship to emacs. we did not think that continuing the path of looking for bug reports would lead to anything noteworthy in the current sense.
We found a site that kept track of the security breaches in ispell. This site found one reported symlink attack that led to a privilege escalation. However this was also patched back in 2001. Although it was patched, it did give some ideas about attacking the security analysis.
Concluding from the internet research we found that much of the online resources were outdated compared to the current version of ispell. The research however, gave good inspiration on how to attack the problem at its base.

Results

User Input

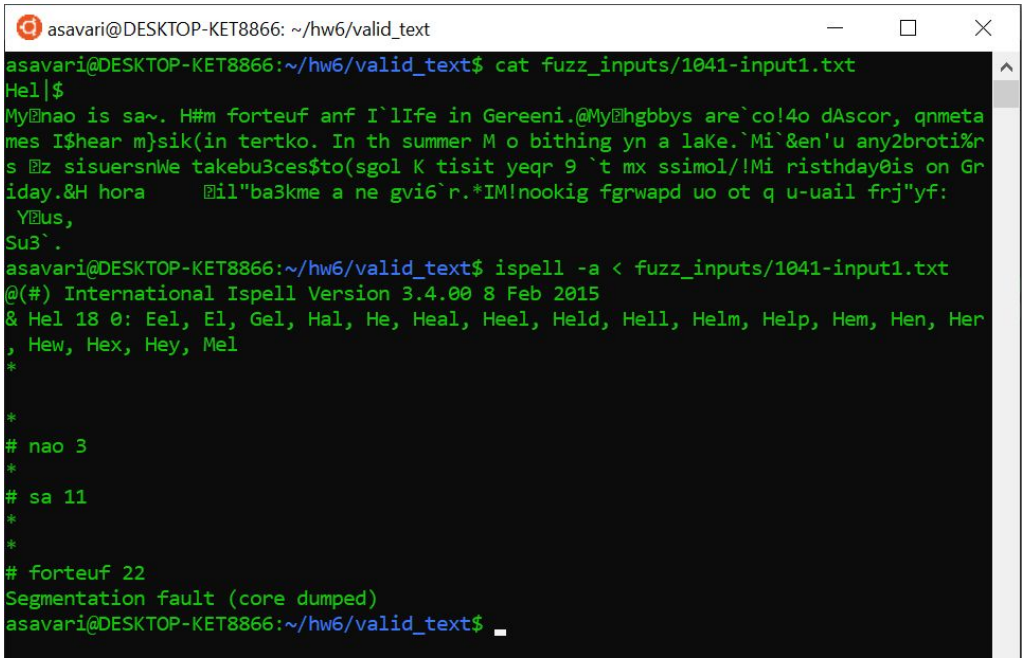
Segmentation Faults

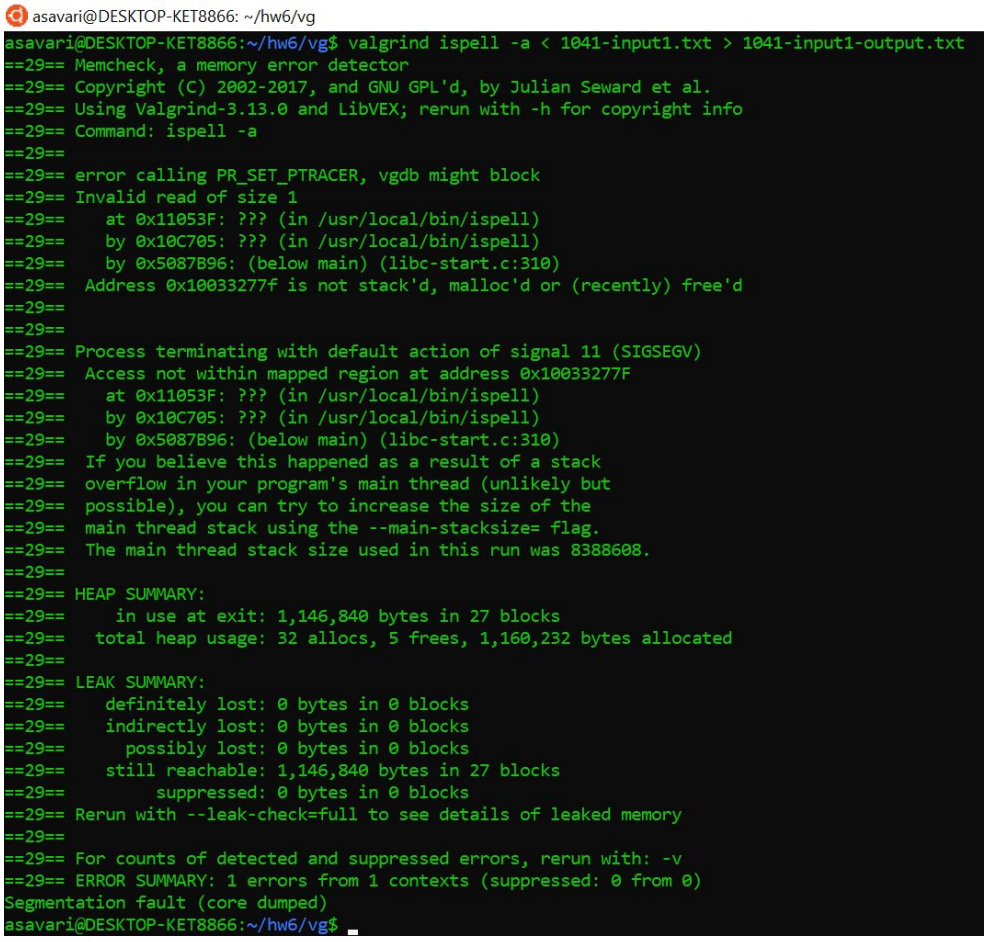
ispell takes in a file provided by the user on the command line and checks for spelling errors in this file. Ispell specifies that the file to be provided as the input must only contain ASCII characters. Since this input to the program is controlled by the user, it is one of the main ways in which a malicious user may try to exploit any vulnerability. To ensure that ispell safely handles the case where the input file does not only have ASCII characters, we can provide different files to ispell and observe its behaviour. These tests were carried out as a part of live testing and fuzz testing. A valid input file was randomly modified using fuzz testing and the behaviour of ispell was noted. It was seen that on some inputs, when ispell is used in the non interactive mode with the -a flag, it results in a segmentation fault and crashes.

Using the tool valgrind, we can find out more about what caused this segmentation fault. The segmentation fault is due to an invalid read, where an attempt is made to read an invalid address. If ispell is used in interactive mode, then there is no segmentation fault observed for all the inputs which were tried.

This is a security issue as a user can carry out a denial of service attack by providing files with random bytes. The specification does not indicate whether ispell is used in the interactive or non interactive mode. Although the web server is supposed to ensure that the files submitted by the user do not contain any ASCII values, we cannot assume that this is done correctly without access to the source code. If a user manages to submit a file with non-ASCII characters, the webserver must handle the segmentation fault correctly to prevent denial of service attacks.

In conclusion, ispell in non interactive mode with the -a flag does not handle non-ASCII input gracefully. The web application must use ispell only in the interactive mode. Any crashes due to segmentation faults must be handled safely.


asavari@DESKTOP-KET8866: ~/hw6/valid_text
asavari@DESKTOP-KET8866:~/hw6/valid_text$ cat fuzz_inputs/1041-input1.txt
Hel|$
My□nao is sa~. H#m forteuf anf I`lIfe in Gereeni.@My□hgbbys are`co!4o dAscor, qnmeta
mes I$hear m}sik(in tertko. In th summer M o bithing yn a laKe.`Mi`&en'u any2broti%r
s □z sisuersnWe takebu3ces$to(sgol K tisit yeqr 9 `t mx ssimol/!Mi risthday0is on Gr
iday.&H hora □il"ba3kme a ne gvi6`r.*IM!nookig fgrwapd uo ot q u-uail frj"yf:
Y□us,
Su3`.
asavari@DESKTOP-KET8866:~/hw6/valid_text$ ispell -a < fuzz_inputs/1041-input1.txt
@(#) International Ispell Version 3.4.00 8 Feb 2015
& Hel 18 0: Eel, El, Gel, Hal, He, Heal, Heel, Held, Hell, Helm, Help, Hem, Hen, Her
, Hew, Hex, Hey, Mel
*
*
nao 3
*
sa 11
*
*
forteuf 22
Segmentation fault (core dumped)
asavari@DESKTOP-KET8866:~/hw6/valid_text$


asavari@DESKTOP-KET8866: ~/hw6/vg
asavari@DESKTOP-KET8866:~/hw6/vg$ valgrind ispell -a < 1041-input1.txt > 1041-input1-output.txt
==29== Memcheck, a memory error detector
==29== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==29== Command: ispell -a
==29==
==29== error calling PR_SET_PTRACER, vgdb might block
==29== Invalid read of size 1
==29==   at 0x11053F: ??? (in /usr/local/bin/ispell)
==29==   by 0x10C705: ??? (in /usr/local/bin/ispell)
==29==   by 0x5087B96: (below main) (libc-start.c:310)
==29== Address 0x10033277f is not stack'd, malloc'd or (recently) free'd
==29==
==29==
==29== Process terminating with default action of signal 11 (SIGSEGV)
==29== Access not within mapped region at address 0x10033277F
==29==   at 0x11053F: ??? (in /usr/local/bin/ispell)
==29==   by 0x10C705: ??? (in /usr/local/bin/ispell)
==29==   by 0x5087B96: (below main) (libc-start.c:310)
==29== If you believe this happened as a result of a stack
==29== overflow in your program's main thread (unlikely but
==29== possible), you can try to increase the size of the
==29== main thread stack using the --main-stacksize= flag.
==29== The main thread stack size used in this run was 8388608.
==29==
==29== HEAP SUMMARY:
==29==   in use at exit: 1,146,840 bytes in 27 blocks
==29==   total heap usage: 32 allocs, 5 frees, 1,160,232 bytes allocated
==29==
==29== LEAK SUMMARY:
==29==   definitely lost: 0 bytes in 0 blocks
==29==   indirectly lost: 0 bytes in 0 blocks
==29==   possibly lost: 0 bytes in 0 blocks
==29==   still reachable: 1,146,840 bytes in 27 blocks
==29==   suppressed: 0 bytes in 0 blocks
==29== Rerun with --leak-check=full to see details of leaked memory
==29==
==29== For counts of detected and suppressed errors, rerun with: -v
==29== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
asavari@DESKTOP-KET8866:~/hw6/vg$

Buffer Overflow

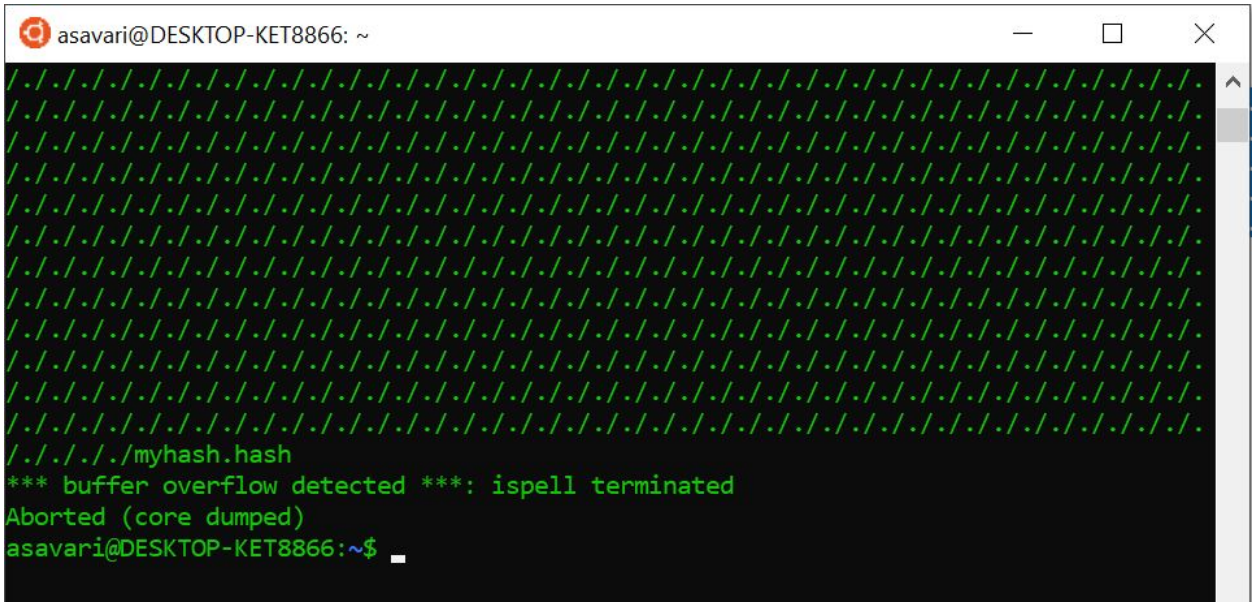
Path to Dictionary

Another major vulnerability based on user input to ispell was a buffer overflow error using the path to the dictionary to be used. The main dictionary to be used by ispell can be provided with the command line option -d followed by the path to the hash of the user's dictionary while launching ispell. The path to the file is provided by the user in the command line. It was observed during manual code review and live testing, that when this path is too long, ispell copies it into a buffer of a fixed size without checking its size. This results in a buffer overflow of a global array. This is confirmed using the tool valgrind.

This is a major security vulnerability as buffer overflow vulnerabilities are very well known and a lot of techniques to exploit them have been developed. This can be exploited by the user to launch a program of their choice or potentially take over the system is ispell is run as root. The spec doesn't specify if the user is allowed to specify their own dictionary, but since support of multiple languages is desired, it is likely that this option is provided to the user.

Recreation:
Command Line:
ispell -d ./../../../../../../../../../...../../../../../../../../../../myhash.hash
Repeat "./" so that the total length of the command is around 5000 characters.

Result:



asavari@DESKTOP-KET8866: ~

```
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
/./././././././././././././././././././././././././././././././././././././././././././././././././
==28806==
==28806== error calling PR_SET_PTRACER, vgdb might block
**28806** *** strcpy_chk: buffer overflow detected ***: program terminated
==28806==    at 0x4C326EC: ??? (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==28806==    by 0x4C370CE: __strcpy_chk (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==28806==    by 0x10C5B5: ??? (in /usr/local/bin/ispell)
==28806==    by 0x5087B96: (below main) (libc-start.c:310)
==28806==
==28806== HEAP SUMMARY:
==28806==     in use at exit: 0 bytes in 0 blocks
==28806==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==28806==
==28806== All heap blocks were freed -- no leaks are possible
==28806==
==28806== For counts of detected and suppressed errors, rerun with: -v
==28806== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
asavari@DESKTOP-KET8866:~$
```

ispell.c

```
                    argc--;
                    if (argc == 0)
                        usage ();
                    cpd = *argv;
                    if (*cpd == '\0')
                        cpd = NULL;
                    }
                LibDict = NULL;
                break;
            case 'd':
                p = (*argv) + 2;
                if (*p == '\0')
                    {
                    argv++;
                    argc--;
                    if (argc == 0)
                        usage ();
                    p = *argv;
                    }
                if (last_slash (p) != NULL)
                    (void) strcpy (hashname, p);
                else
```

ispell.h

```
616 EXTERN struct dent *
617                 hashtbl;        /* Main hash table, for dictionary */
618 EXTERN unsigned int
619                 hashsize;       /* Size of main hash table */
620
621 EXTERN char     hashname[MAXPATHLEN]; /* Name of hash table file */
622
623 EXTERN int      aflag;          /* NZ if -a or -A option specified */
624 EXTERN int      cflag;          /* NZ if -c (crunch) option */
625 EXTERN int      lflag;          /* NZ if -l (list) option */
626 EXTERN int      incfileflag;    /* whether xgets() acts exactly like gets() */
627 EXTERN int      nodictflag;     /* NZ if dictionary not needed */
628
629 EXTERN int      uerasechar;     /* User's erase character, from stty */
630 EXTERN int      ukillchar;      /* User's kill character */
631
632 EXTERN unsigned int laststringch; /* Number of last string character */
633 EXTERN int      defstringgroup; /* Default string character group type */
634
```

Temp Folder Environment Variable

While performing manual code review of the file ispell.c, it was found that the contents of the environment variable for the path to the temp folder are copied into a buffer of fixed size without checking to see if its length exceeds that of the global buffer.

This is another buffer overflow error similar to the previous one. The user can set the path to the temporary folder to be used in the environment variables $TMPDIR, $TEMP or $TMP. This string gets copied into the global array “tempfile” using sprintf.

ispell.c

```
asavari@DESKTOP-KET8866: ~/hw6/ispell-3.4.00
1080     if (last_slash (TEMPNAME) != NULL)
1081         (void) strcpy (tempfile, TEMPNAME);
1082     else
1083         {
1084         char *tmp = getenv ("TMPDIR");
1085         int   lastchar;
1086
1087         if (tmp == NULL)
1088             tmp = getenv ("TEMP");
1089         if (tmp == NULL)
1090             tmp = getenv ("TMP");
1091         if (tmp == NULL)
1092 #ifdef P_tmpdir
1093             tmp = P_tmpdir;
1094 #else
1095             tmp = "/tmp";
1096 #endif
1097         lastchar = tmp[strlen (tmp) - 1];
1098         (void) sprintf (tempfile, "%s%s%s", tmp,
1099                         IS_SLASH (lastchar) ? "" : "/",
1100                         TEMPNAME);
1101         }
1102 #ifdef NO_MKSTEMP
```

ispell.h

```
729 INIT (int terse, 0);                    /* NZ for "terse" mode */
730 INIT (int correct_verbose_mode, 0);     /* NZ for "verbose" -a mode */
731
732 INIT (char tempfile[MAXPATHLEN], "");   /* Name of file we're spelling into */
733
734 INIT (int minword, MINWORD);            /* Longest always-legal word */
735 INIT (int sortit, 1);                   /* Sort suggestions alphabetically */
```

For both these buffer overflow errors, sprintf and strcpy must be replaced with snprintf and strncpy, where the maximum number of characters to copy can be set to the maximum size of the buffer.

Symlinks

While conducting research, I came across a reported security flaw in ispell that had been exploited in 2001. this bug was the result of a symlink attack. The code insecurely used the function mktemp to create a temporary storage file. However, an attacker could create a file and symbolically link it to a file that they did not have permission. If the file the attacker created had the same name as the temporary file created by the program, it would allow the attacker to insert malicious information into a file that they did not have write access to. This is the exact exploitation discovered in the program. This is an extremely fatal flaw as it would allow privilege escalation and eventually root access to any system that ispell was located on. Obviously being a high priority security flaw, this bug was mended quickly after the bug was noticed and exploited. A linux function mkstemp creates a unique temporary file so that the name of the file will not be known beforehand.

Curious of some current uses of mktemp in the code, I did a grep on all of the files in the program. I came across one occurence of mktemp in ispell.c on line 1103.

```
1102 #ifdef NO_MKSTEMP
1103     if (mktemp (tempfile) == NULL  ||  tempfile[0] == '\0'
1104 #ifdef O_EXCL
1105       ||  (outfd = open (tempfile, O_WRONLY | O_CREAT | O_EXCL, 0600)) < 0
1106       ||  (outfile = fdopen (outfd, "w")) == NULL)
1107 #else /* O_EXCL */
1108       ||  (outfile = fopen (tempfile, "w")) == NULL)
1109 #endif /* O_EXCL */
1110 #else /* NO_MKSTEMP */
1111     if ((outfd = mkstemp (tempfile)) < 0
1112       ||  (outfile = fdopen (outfd, "w")) == NULL)
1113 #endif /* NO_MKSTEMP */
1114         {
1115         (void) fprintf (stderr, CANT_CREATE,
1116           (tempfile == NULL  ||  tempfile[0] == '\0')
1117             ? "temporary file" : tempfile,
1118           MAYBE_CR (stderr));
1119         (void) sleep ((unsigned) 2);
1120         return;
1121         }
```

This only uses mktemp when mkstemp is not available to create a file. If mkstemp is not supported on a system, this could cause security issues. However this is not the case for the majority of systems and thus, this fix addresses the original security concern.

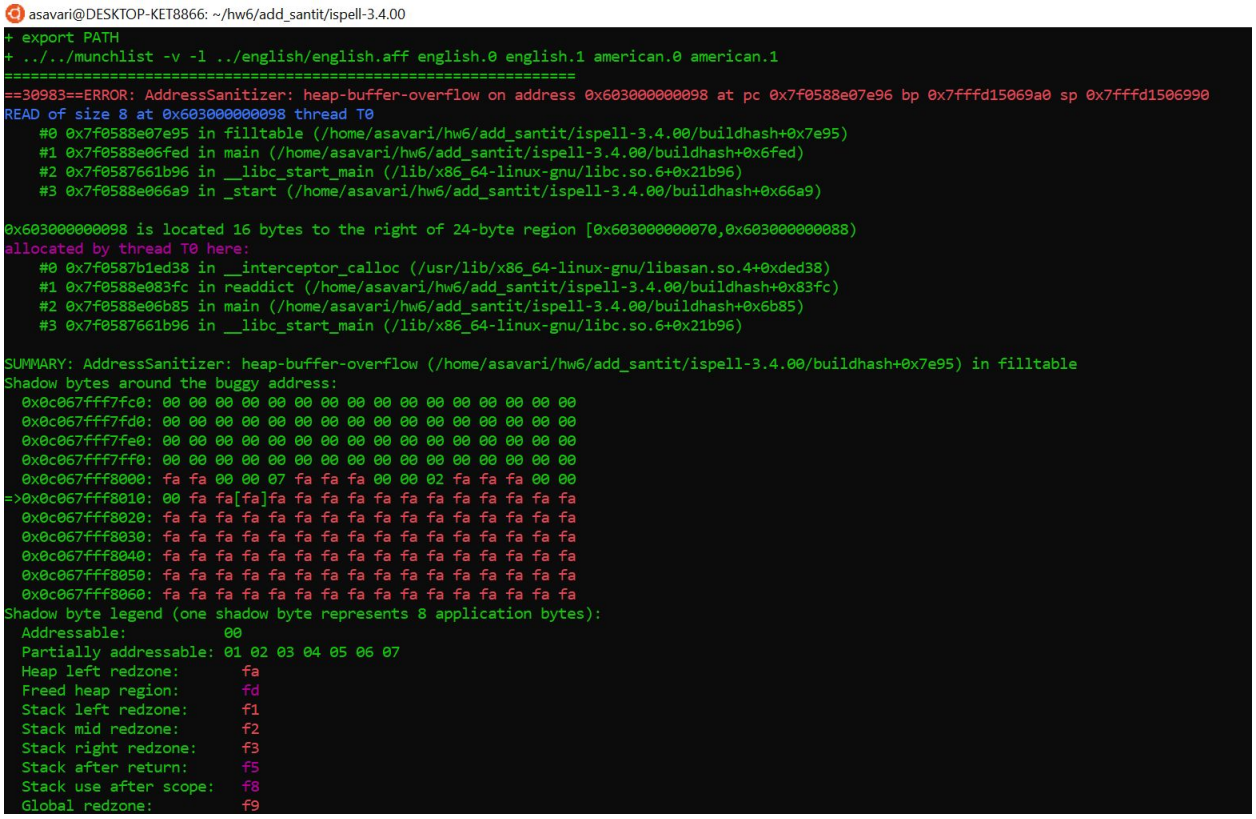
Buffer Overflow

There was a comment and code in correct.c on lines 349-386 that stood out to me as a possible security flaw. the code outlined that for any word read in, it would create a buffer for the correction of the word. The idea behind the buffer is that they would only read half of the buffer size and thus have half the buffer left to insert more characters if it is required for a correction. while this is robust for the vast majority of words or corrections, it is not the safest way to handle a buffer for corrections. If a word were to fill the buffer and its correction happened to be twice the length of that word, this could cause a segfault and crash the program. The attacker could utilize this across the server and create a DoS attack on ispell. Granted, this attack would take a massive amount of time and resources to pull off. the combination of letters and incorrect spelling would require a lot of testing and time. that being said, it does seem possible that it could be pulled off. the code snippet addressed is as follows:

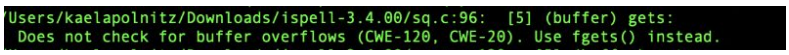
```
349         /*
350          * Only read in enough characters to fill half this buffer so that any
351          * corrections we make are not likely to cause an overflow.
352          */
353         if (fgets ((char *) filteredbuf, sizeof filteredbuf / 2, infile)
354           == NULL)
355             {
356             if (sourcefile != NULL)
357                 {
358                 while (fgets ((char *) contextbufs[0], sizeof contextbufs[0],
359                     sourcefile)
360                   != NULL)
361                     (void) fputs ((char *) contextbufs[0], outfile);
362                 }
363             break;
364             }
365         /*
366          * If we didn't read to end-of-line, we may have ended the
367          * buffer in the middle of a word.  So keep reading until we
368          * see some sort of character that can't possibly be part of a
369          * word. (or until the buffer is full, which fortunately isn't
370          * all that likely).
371          */
372         bufsize = strlen ((char *) filteredbuf);
373         if (bufsize == sizeof filteredbuf / 2 - 1)
374             {
375             ch = (unsigned char) filteredbuf[bufsize - 1];
376             while (bufsize < sizeof filteredbuf - 1
377               &&  (iswordch ((ichar_t) ch)  ||  isboundarych ((ichar_t) ch)
378               ||  isstringstart (ch)))
379                 {
380                 ch = getc (infile);
381                 if (ch == EOF)
382                     break;
383                 filteredbuf[bufsize++] = (char) ch;
384                 filteredbuf[bufsize] = '\0';
385                 }
386             }
```

A simple fix to the problem could be to dynamically allocate memory for the buffer so you do not have to deal with strict limitations to the size of your words. This would mend the buffer overflow problem and, avoiding, memory leaks, would mend any inkling of a security issue in this section.

Ispell was compiled with address sanitization enabled to look for invalid memory accesses while the code is running. This is a live testing tool. A possible buffer overflow vulnerability was discovered in the file buildhash.c in the function “filltable”. This is probably the same vulnerability which was discovered using the automated static code review. This is less dangerous than a buffer overflow vulnerability, but can still be exploited to overwrite other data in the heap.



Another issue dealing with buffer overflow can be found at one particular case in the sq.c program, where we were prompted with the following warning from a static code automated analysis tool:



After further investigation, it was found that this was a pretty serious issue. This was because the original implementation (gets()) did not limit the characters that were given by standard

input. This would allow an attacker to manipulate the information provided and the subsequent encodings that were done in this program.
To minimize vulnerability, it would have been a better choice to use a function like fgets() that would set boundaries on what would be provided as input. An effort to implement this could look like the following:

sq.c, line 96

```
while ~~(gets (word) != NULL) J~~(fgets(word, 257, stdin) != NULL)
        trunc (word, prev);
```

Incorrect Behavior with the Null Character

While not much of a security concern as it likely not exploitable in any useful way, we found that if a file were to contain a null character, if the null character were to make its way to any buffer in the ispell program from reading from stdin, any instance of “strlen()” run on the buffer would return a length that corresponds to truncated data.

In a particular case from our testing, we found that if a line were to begin with a null character, “strlen()” would return with a length of 0, and accessing the last character using the position “length - 1” would result in a segmentation fault. This is the case we found through using a combination of zzuf, AFL, and gdb. As mentioned before, due to the conflicting nature of the xgets/fgets functions (which works with raw bytes) and “strlen()” (which works strictly with c-string representations), the “askmode()” function in “correct.c” would fail on lines beginning with a null character. The problematic code is as follows (lines 1687-1711):

```
if (contextoffset == 0)
{
        if (xgets ((char *) filteredbuf, (sizeof filteredbuf) / 2, stdin) == NULL)
                break;
}
else
{
        if (fgets ((char *) filteredbuf, (sizeof filteredbuf) / 2, stdin) == NULL)
                        Break;
}
 /*
  * Make a copy of the line in contextbufs[0] so copyout works.
  */
(void) strcpy ((char *) contextbufs[0], (char *) filteredbuf);
/*
 * If we didn't read to end-of-line, we may have ended the
 * buffer in the middle of a word.  So keep reading until we
 * see some sort of character that can't possibly be part of a
 * word. (or until the buffer is full, which fortunately isn't
 * all that likely).
 */
bufsize = strlen ((char *) filteredbuf);
hadnl = filteredbuf[bufsize - 1] == '\n';
```

If we were to recompile ispell by modifying “config.X” with “CC” as gcc and “CFLAGS” as “-g” (no optimization with “-O” as it would interfere with gdb), we can test ispell using gdb. Below is a

sample of what we would get if we were to take an input that would give a segfault from AFL as input in gdb.

```
(gdb) r -a < afl_out_a_option/crashes/id:000000,sig:11,src:000009,op:flip2,pos:28
Starting program: /home/dzang/Testing/ispell-3.4.00/ispell -a <
afl_out_a_option/crashes/id:000000,sig:11,src:000009,op:flip2,pos:28
@(#) International Ispell Version 3.4.00 8 Feb 2015
*
Program received signal SIGSEGV, Segmentation fault.
0x000055555555da43 in askmode () at correct.c:1711
1711        hadnl = filteredbuf[bufsize - 1] == '\n';
(gdb) bt
#0  0x000055555555da43 in askmode () at correct.c:1711
#1  0x0000555555558f74 in main (argc=0, argv=0x7fffffffdfb8) at ispell.c:978
(gdb) info locals
bufsize = 0
ch = 0
cp1 = 0x636f6c2f7273752f <error: Cannot access memory at address 0x636f6c2f7273752f>
cp2 = 0x5555555564b0 <_start> "1\355I\211\321^H\211\342H\203\344\360PTL\215\005*&\001"
itok = 0x121c7499f3851a00
hadnl = 1
(gdb) p 'ispell.c'::filteredbuf
$1 = "\000\031\032\033\034\n", '\000' <repeats 8185 times>
```

Of course, the program wouldn't crash here if the first character weren't a null character, but having a null character in the middle of a line would end up with some data being truncated and ignored.

Potential Spellcheck / File Access Exploitation with Wrong Ispell Options

Although not completely related to reviewing or testing the actual code, the command-line options of ispell that modify the behavior of ispell should also be taken into account. We have reviewed each of the options for ispell, and have found that, if ispell were to be used to a web server application, there are two options we need to be particularly careful about: '-a' and '-A'.

The '-a' option indicates that ispell will parse lines beginning with particular characters ('*', '&', '@', '+', '-', '~', '#', '!', '%', '`', or '^') to modify the current dictionary, like inserting/removing words and changing parsing mode. If ispell were to be left continuously running and access the files to check as a continuous stream of bytes (as in one user's changes can affect another user).

The '-A' option is exactly like the '-a' option, but allows for an extra capability: accessing files besides the current one. By providing a line beginning with "&Include_File&", the rest of the line will be taken as a file name, and ispell will open the new file and run itself on that file. Depending on the program's and files' permissions, this may pose as an issue if set up incorrectly (an attacker may find ways to utilize this to modify the files of other users).

Incompatible Types

We also looked at warnings dealing with the use of incompatible types. More specifically, we looked at warnings in xgets.c similar to the following:

```
xgets.c:123:2: Assignment of char to int: c = '\0'
  A character constant is used as an int. Use +charintliteral to allow
  character constants to be used as ints.  (This is safe since the actual type
  of a char constant is int.)
xgets.c:128:10: Operands of != have incompatible types (int, char): c != '\n'
```

With this, we searched through the file to understand these instances. After taking a look, the mixed use of types looked intentional. I believe this was a stylistic choice, because the use of characters such as ‘\n’ and ‘\0’ are used correctly in their integer form. If one wanted to fix this issue, they could simply change their usage from an integer form to a character form (as they are used interchangeably in this situation), but in general, it seems to be a nonissue.

Null Reference

Looking for somewhere to exploit some null references, I searched for a tool that would help automate a process for searching. I came across a program called cppcheck which scans for potential null pointer references. I was able to get some output from the tool:

tgood.c:439:19: warning: Possible null pointer dereference: pfxent [nullPointer]
flagpr (tword, BITTOCHAR (pfxent->flagbit),
^
tgood.c:193:42: note: Calling function 'chk_suf', 5th argument '(struct flagent*)NULL' value is 0
chk_suf (word, ucword, len, sfxopts, (struct flagent *) NULL,
^
tgood.c:337:64: note: Calling function 'suf_list_chk', 6th argument 'pfxent' value is 0
suf_list_chk (word, ucword, len, &sflagindex[0], optflags, pfxent,
^
tgood.c:439:19: note: Null pointer dereference
flagpr (tword, BITTOCHAR (pfxent->flagbit),
^
tgood.c:440:6: warning: Possible null pointer dereference: pfxent [nullPointer]
pfxent->stripl, pfxent->affl,
^
tgood.c:193:42: note: Calling function 'chk_suf', 5th argument '(struct flagent*)NULL' value is 0
chk_suf (word, ucword, len, sfxopts, (struct flagent *) NULL,
^
tgood.c:337:64: note: Calling function 'suf_list_chk', 6th argument 'pfxent' value is 0
suf_list_chk (word, ucword, len, &sflagindex[0], optflags, pfxent,
^
tgood.c:440:6: note: Null pointer dereference
pfxent->stripl, pfxent->affl,

One such reference is pictured below:

```
if (cflag)
    {
    if (optflags & FF_CROSSPRODUCT)
        flagpr (tword, BITTOCHAR (pfxent->flagbit),
          pfxent->stripl, pfxent->affl,
          BITTOCHAR (flent->flagbit), flent->affl);
    else
        flagpr (tword, -1, 0, 0,
          BITTOCHAR (flent->flagbit), flent->affl);
    }
```

To translate this output, I searched through the code for what this could be referring to. Pfxent is a prefix flag that is only set if a prefix is used for its input. This could easily cause a null pointer reference if a prefix is not initialized before it is referenced. A simple if statement to check if the variable is null would easily fix the problem. I am not sure how easily this could be exploited, but is fatal to the program's success and would cause a segfault. This should be a simple fix in order to ensure the programs reliability and prevent a denial of service attack if this flaw could be exploited with any regularity.

Memory Leaks

Using a tool called cppcheck, I was able to find a memory leak in the program. This problem showed up in fields.c in line 107. Some dynamically allocated memory was declared but not properly released upon an error. more specifically, a variable "linebuf" was dynamically allocated and reallocated without being properly freed upon an error. If the attacker was able to time the code right and cause an error right at the time of the potential memory leak, they could cause a denial of service attack on the system. The error output I received was as follows:

fields.c:107:6: error: Common realloc mistake: 'linebuf' nulled but not freed upon failure [memleakOnRealloc] linebuf = (char *) realloc (linebuf, linemax);

```
 98      while (fgets (&linebuf[linesize], linemax - linesize, file)
 99        != NULL)
100          {
101          linesize += strlen (&linebuf[linesize]);
102          if (linebuf[linesize - 1] == '\n')
103              break;
104          else
105              {
106              linemax += field_line_inc;
107              linebuf = (char *) realloc (linebuf, linemax);
108              if (linebuf == NULL)
109                  return NULL;
110              }
111          }
112      if (linesize == 0)
113          {
114          free (linebuf);
115          return NULL;
116          }
117      return fieldmake (linebuf, 1, delims, flags, maxf);
118      }
```

As shown in the code snippet, if the linebuf is filled and then not freed given an error, an attacker could leverage this for a denial of service attack. A simple solution for this is to attempt to free all memory upon an error exit as a clean up so that no memory is left once the program encounters an error. This is not a greatly vital issue and it would be hard to attack the program in the way described consistently. However this could be fatal to the service of ispell.

Using automated tools, we found another memory leak. Particularly, in defmt.c:

```
defmt.c:1403:14: Last reference wlist to owned storage end not released before
                   return
  A memory leak has been detected. Only-qualified storage is not released
  before the last reference to it is lost. (Use -mustfreeonly to inhibit
  warning)
```

Once again, we navigated to the source code to gain context for this vulnerability. After searching, it was found that this would be a pretty serious issue. Not only is bad practice, but it also leaves the program open for attacks. For example, an attacker could easily use this vulnerability to trigger a denial of service attack for the user; they could cause the program to crash by exploiting the loss of memory, preventing users from being able to validate filenames. With low memory, you could have a variety of undefined behavior for the user as well.

In general, this problem is one that should be fixed. One potential solution would be to make sure to free all parts of the `keywordbuf` variable. The creator of the source code attempted this (line 1394), but it doesn't seem that it is being done correctly or globally.

There was also an issue of allocated storage. During the search, we were prompted with the following warnings in the buildhash.c file:

```
buildhash.c:639:2: Storage d.word is released in one path, but live in another.
   buildhash.c:624:3: Storage d.word released
buildhash.c:639:2: Storage d.next is released in one path, but live in another.
   buildhash.c:572:6: Storage d.next released
```

This looked worrisome, so we decided to take a look at the source code. After taking a look, I found a comment (line 587) that explained that the storage would always be alive during the condition that these warnings were found in. From our observations, it seemed that that was the reason why the storage always remained active. With this, we found this particular warning to be harmless in the greater security and privacy of the program itself.

File Opening and Reading

Using manual code review to check for function calls to open, read, or write files, we executed grep on the codebase for variants of f/gets() and f/open(). As mentioned earlier, replacing the usage of gets() with fgets() is a safer, more secure way of avoiding buffer overflows.

There are 7 usages of fdopen(), fopen(), and freopen(), and all calls to these functions check for errors using the return value of the function. If the function returns NULL, the program will exit or return with error code 1. No fix is needed, since unauthorized attempts to access these files will fail with an error that is checked.

Buffer Reading and Copying

To look for security issues in buffer reading and copying, we executed grep to search for usages of memcpy() and memmove(). We discovered in file config.X that BCOPY() is a macro for either memcpy() or bcopy(), and BZERO() is a macro for memset() or bzero(), depending on the version system. These two variants have different syntax: memcpy(dest, src, length) and memset(dest, length) have parameters reversed from bcopy(src, dest, len) and bzero(dest, len). However, the macros BCOPY() and BZERO() are defined to be BCOPY(src, dest, length) and BZERO(dest, length).

The signatures for these functions are as follows:

```
void* memcpy(void* s1, void* s2, size_t len); //returns s1, the dest
void* memset(void* str, int c, size_t n); //returns str, the dest
void bcopy(const void* s1, void* s2, size_t n); // s1 is source
void bzero(void* s, size_t n);
```

Doing some research, we discovered that bcopy() has been deprecated in favor of memmove(), and bzero() has been deprecated in favor memset(). These dual defined macro is probably for backwards compatibility.

However, we noticed the macro defines BCOPY() using memcpy(), but bcopy() was deprecated in favor of memmove() instead of memcpy().

Some research revealed that memcpy() has undefined behavior when the s1 and s2 buffers have overlapping regions. On the other hand, memmove() will handle this scenario and perform correct buffer copying instead. The tradeoff is that memmove() is slightly slower. The signature for memmove() is below

```
void* memmove(void* to, const void* from, size_t numBytes);
```

Replacing the usage of memcpy() with memmove() is a safer one, and the expense of slightly slower buffer movement does not seem to be a huge hindrance to the system, given the relative lack of frequency that BCOPY() seems to be called. If BCOPY() is ever called on memory that overlaps, this could crash the program, access invalid memory, like another process' memory, or something else dangerous. We have no idea what could happen, since the behavior is undefined with the usage of memcpy().

We took a closer look at the calls to BCOPY(), and they are addressed below.

exp_table.c, line 125, in function add_expansion_copy()

```
BCOPY ((char *) &flags[0], &e->flags[e->size * MASKSIZE], MASKSIZE * sizeof
flags[0]);
source: &flags[0]
dest: &e->flags[e->size*MASKSIZE]
length: MASKSIZE*sizeof flags[0]

//initialization:
e->flags = malloc (e->max_size * sizeof (*e->flags) * MASKSIZE)
```

This usage seems to be safe, as long as e->size is less than or equal to e->maxsize, and e->flags is always initialized via exp_table_init(), which will restrict it to a max size. Grepping through source code, it looks like every creation of any struct exp_table will call exp_table_init(), so no additional security patches are needed for this call to BCOPY().

correct.c, line 1550

```
(void) BCOPY ((char *) (p + prestrip),
                (char *) (newword + preadd),
                (len - prestrip - sufstrip) * sizeof (ichar_t));
    source: p+prestrip
    dest: newword+preadd
    length: len-prestrip-sufstrip * sizeof(ichar_t)
```

We need to verify that newword+preadd+(len-prestrip-sufstrip)*sizeof(ichar_t) is valid memory location that has been allocated for access, and looking below;

```
//initialization:
ichar_t newword[INPUTWORDLEN + 4 * MAXAFFIXLEN + 4]
```

icharcpy(newword, word);
preadd: number of chars added to front of root
len: number of characters in dent->word

This usage seems to be safe, by the logic that the length of newword is the same as the length of word, preadd is the same length prestrip, and the length of sufstrip will be greater than or equal to 0, so the memory location at dest+length should be safe for the program to access.

parse.y, line 714, 724, 1735, 1736
(void) bcopy ((char *) $2, hashheader.nrchars, sizeof (hashheader.nrchars));
(void) bcopy ((char *) $2, hashheader.texchars,sizeof (hashheader.texchars));
(void) bcopy (NRSPECIAL, hashheader.nrchars, sizeof hashheader.nrchars);
(void) bcopy (TEXSPECIAL, hashheader.texchars, sizeof hashheader.texchars);

We took a closer look at calls to BZERO() as well.

ispell.c, line 1419
(void) BZERO ((char *) mask, sizeof (mask));

makedent.c, line 183
(void) BZERO ((char *) d->mask, sizeof (d->mask));

parse.y, line 818, 1191, 1211, 1491
(void) bzero ($$.set, SET_SIZE + MAXSTRINGCHARS);
(void) bzero (ent->conds, SET_SIZE + MAXSTRINGCHARS);
(void) bzero (ent->conds, SET_SIZE + MAXSTRINGCHARS);
(void) bzero (yylval.charset.set, SET_SIZE + MAXSTRINGCHARS);
// set = (unsigned char *) malloc (SET_SIZE + MAXSTRINGCHARS);

tgood.c, line 625
BZERO (applied, sizeof (applied));

The calls to BZERO() in the four files above seem to be safe, as the exact size of the buffer being zero-ed out is always specified as the argument to the length parameter of the function.

String Manipulation

Checking for functions that manipulate strings, strcpy() and strcat() were the most common and also the most likely to be vulnerable to security issues, because they have no return value, and do not require an argument that would specify the length of the source string to copy or the size of the destination string's buffer to concatenate to.

Executing grep on “strcpy” yielded 44 usages throughout the codebase. This analysis addresses the two scenarios: safe and unsafe usage of strcpy().

Some usages of strcpy(dest, src) are safe-- even without an explicit check or assertion before the call to strcpy(), tracing the logic to where the source and destination buffers are defined, the length of the destination string was larger than or equal to the length of the source string.

For example, buildhash.c line 593 is safe:

```
(void) strcpy ((char *) ucbuf, (char *) d.word);

//declaration of dest always larger than d.word, which is INPUTWORDLEN
unsigned char ucbuf[INPUTWORDLEN + MAXAFFIXLEN + 2 * MASKBITS];
```

correct.c line 337-338 also seems to be safe:

```
(void) strcpy ((char *) contextbufs[bufno], (char *) contextbufs[bufno - 1]);
```

However, some usages are not secure, because the source buffer is a string that is user-inputted and the length isn’t verified.
For example, buildhash.c line 181 is NOT secure:

```
strcpy (Sfile, Dfile);
    // Dfile = argv[1]
    // char Sfile[MAXPATHLEN];
    // could be buffer overflow since Dfile is inputted
    // and MAXPATHLEN doesn’t consider absolute path lengths
```

To fix these insecure usages of strcpy(), as in buildhash.c, the destination string’s length should be checked, and especially if the source string to copy is user-inputted, the length should be determined and verified to be shorter or equal to the length of the destination string, so there isn’t overwriting of the buffer beyond the allocated region.

Executing grep on strcat yielded 10 occurrences in the codebase. The function’s signature strcat(dest, src) appends a copy of the source string to the destination string. The usages of strcat() produced the same concerns as of strcpy(). However, instead of needing the destination buffer to be at least the same length as the source buffer, calls to strcat() require that the destination buffer be the length of what’s currently in the destination string, plus the length of the source string to concatenate to the end. strcat() is called in the following files:

buildhash.c, line 185

```
strcat (Sfile, STATSUFFIX);
// char  Sfile[MAXPATHLEN];
```

The destination string may or may not be large enough to append STATSUFFIX, since it is defined to be the size of MAXPATHLEN, and buildhash.c does not verify the length of Sfile and STATSUFFIX. This check should happen, in case there is a buffer overflow and other data is overwritten or accessed.

defmt.c, lines 1331-1340

```
if (wlist[0] != '\0')
   strcat (wlist, "","");
strcat (wlist, envtags);

if (deftags != NULL) {
if (wlist[0] != '\0')
   strcat (wlist, "","");
strcat (wlist, deftags);
}
//    char *       wlist;            /* Modifiable copy of raw list */
// unsigned int wsize;            /* Size of wlist */
// envtags = (envvar == NULL) ? NULL : getenv (envvar);
// deftags is a function input
```

There are repeated usages of strcat() in defmt.c, and the concern is that wlist is not long enough to concatenate envtags and deftags, both of which are some form of input that don't have their lengths verified.

ispell.c, lines 304, 846, 1282-1284

```
strcat (hashname, HASHSUFFIX);
if (strlen (pathtail) > MAXNAMLEN - sizeof BAKEXT + 1)
   pathtail[MAXNAMLEN - sizeof BAKEXT + 1] = '\0';
(void) strcat (pathtail, BAKEXT);"
```

The usages of strcat() in ispell.c face the same security concerns of buffer overflowing as in defmt.c and buildhash.c. The size of the destination buffer are not verified to be large enough to concatenate the arguments passed in the function calls (the length of these arguments are also variable, since some of them are user inputted); this should happen in some form before calls to strcat().

Recommendations and Conclusions

Security patches

1. Ispell should not be run with files that contain null characters.

2. The server should ensure that the file provided as input should only contain ASCII characters, even when used in interactive mode.
3. The server should not directly use any user input string as the path to the user specified dictionary on the command line as would lead to a buffer overflow.
4. The server should carefully handle ispell crashes.
5. Command line input to ispell by the server must be checked for length.
6. Replace usage of memcpy() with memmove().
7. Verify that the size of the destination buffer for the destination string is large enough for calls to strcpy() and strcat(), whether explicitly in code, or by tracing logic to verify that the allocated buffer size is large enough.

Recommendations for Future Evaluations

In future efforts, a possibly more in depth understanding of the code could provide us with more accurate feedback. In the time we had reviewing the code, we had obtained a higher level of understanding about the code. In the time we had, we had to rely on a lot of tools and programs to help us better understand the source code. These programs help a great deal in the analysis of programs but they are not always accurate or comprehensive.

Our analysis focused on some well known attacks, however with more time we might have been able to summon up some more creative ideas for attacking the system beyond the surface level. Something we did not have a chance to do was to live test ispell in the context of the web server system, performing end-to-end testing. There might be other security vulnerabilities that may arise when the entire system is running, that wouldn't be an issue when testing individual components alone.

A fresh review might be required upon any massive changes to the core of the system. While a security check for every update would be nice, it would not necessarily be feasible. Any changes to the more vital portions of code such as code with a lot of dependencies would require a security analysis. Because this portion of the code is so vital to the security of the program and the system, it is important to review before publicly released.

Lessons learned

1. While using C, avoid using functions known to cause buffer overflows, like sprintf or strcpy. Always verify, either in code or by hand (and then note in code comments), that destination buffers are large enough to handle anything passed in as input. Otherwise, verify that the only data that can fit into the specified buffer will be read and written, or none at all will be.
2. User input must be validated before it is used--whether read, written, copied into a buffer, or used as an argument to another function, which may or may not verify the length of that input.

3. One limitation of fuzz testing, in which only errors that have the program crash due to error signals like SIGSEGV can be caught (logic errors that result in errors like unintended file access and truncated buffers due to improper string representation will not be caught).
4. Results of static code analysis can be used to prevent similar vulnerabilities in the future. Running automated tools on code and preemptively addressing warnings can reduce vulnerabilities and increase the security of the overall system, before attacks can happen.
5. Designing and evaluating a secure system is easier with documentation. If we had access to design documentation that identified intended entry points into the system, we might have had a concrete place to start our security evaluation, and insight into the thought process of designing the system. We also might have been able to spot vulnerabilities based on the intention of the design, instead of having to read through all of the source code to understand where potential vulnerabilities could occur.
6. In addition to tools that identify common problems by looking at static code, researching common problems and reading through documented bugs, we found that splitting up the work to evaluate a system and codebase of this size made it significantly easier to handle, especially given a time limit.
7. Of the issues that we discovered, they all seemed to have relatively straightforward fixes. For example, writing in verifications so that buffer overflow cannot occur. We didn't discover any complex attack approach that required a multi-step security patch, but that doesn't mean it doesn't exist. If we had more time, we might have tried more complicated strategies to look for security vulnerabilities.

Work breakdown

Name, UID	**Bugs/Techniques**	**Parts of Report**
Asavari Limaye 605224431	Live Testing Fuzz Testing User Input Manual Code Review Used the tools zzuf, valgrind, and address sanitization to find errors. Found the dictionary path and temp folder vulnerability by live testing and manual code review.	1. Original Schedule for Security Evaluation 2. Tools and Approaches Used - Live and Fuzz Testing 3. User Input - Segmentation Fault 4. Buffer Overflow: a. Filltable b. Path to Dictionary c. Temp folder Environment Variable 5. Recommendations and Conclusions

Davis Gomes 104907446	Manual Code Review Code Analysis Tools Internet Research Split time between internet research and looking through the code to find bugs.	1. Known security issues 2. Buffer Overflow 3. Null Reference 4. Memory Leak 5. Buffer Overflow 6. Recommendations and Conclusions
Kaela Polnitz 504751829	Static Automated Code Analysis Manual Code Review Used tools Flawfinder, Splint, and CPPCheck to do static C code analysis.	1. Static Automated Code Analysis 2. Buffer Overflow 3. Storage Release Management 4. Incompatible Types 5. Memory Leaks
Hannah Nguyen 304787645	Manual Code Review Internet Research Skimmed codebase hierarchy for important files and their functions. Searched and read through the code looking for common security vulnerabilities in C.	1. Summary 2. Manual Code Review 3. File Opening and Reading 4. Buffer reading/copying 5. String manipulation 6. Lessons Learned 7. Recommendations and Conclusions 8. Report formatting, revising, compilation, and submission
Dennis Zang 704766877	Live / Fuzz Testing Manual Code Review Use of AFL, gdb, and strace to Review of ispell man page to figure out potential vulnerabilities. Manual reading through code.	1. Live Testing and American Fuzzy Lop 2. Manual Debugging and Tracing through Program / Code 3. Potential Command-Line Options That May Lead to Vulnerabilities 4. Finding Viable Approaches to Apply Formal Testing

Work Breakdown - Effectiveness of Method and Tools used

Asavari Limaye

I chose to approach the evaluation of security using fuzz and live testing because the easiest way for an attacker to attack a system is by using it and without any access to the source code. The user interface of ispell must be secure and it must handle invalid user inputs correctly for it to be secure.

The tool zzuf is available on ubuntu and can be installed easily. however, its functionality is limited and it could not easily be used for fuzz testing when ispell is opened in interactive mode. To test ispell in interactive mode, I used another fuzzing tool called American Fuzzy Lop. This can automatically generate fuzzed input but also cannot fuzz ispell in interactive mode. This tool also produces comprehensive reports and is very powerful. The tool valgrind was used to get more details about the execution and the reason for a crash. Address sanitization is a feature available with certain compilers, which finds possible invalid memory accesses and other memory errors. This tool had limited utility as it stops execution when the first error is detected.

After using these tools, I manually examined the source code to find the relevant parts. Live testing and fuzzing tools speed up the process of finding vulnerable parts of the source code.

Davis Gomes

Unfortunately, not many security issues for ispell existed online. I did a pretty extensive search to find problems that had been discussed online. Important bugs, if there are any addressed, are probably classified. I was able to find some patched bugs that gave me ideas on how to approach looking through the source code. Overall, this internet's effectiveness on finding a solution was not especially helpful for ispell.

I somewhat pivoted my approach after exploring the internet by doing a combination of looking through the source code and using code review tools. This method proved to be much more effective and expedient. Through these methods I was able to collect a lot more information on the system and thus find more security issues.

Kaela Polnitz

During this search, most of the focus was on the C files contained in the iSpell program. To evaluate the program in its entirety, it would have been good to also take a look at the scripts and header files separately with their own analysis.

In addition, it would have been better to use multiple environments. For this part, the tools were used on Mac OS. It is possible that different errors or warnings could arise depending on what operating system you are using (Linux, Windows, etc.)

Hannah Nguyen

Manual code review was initially a daunting strategy for a codebase of this size. Beyond “grep” and search functions, and using the Internet to research what were common security vulnerabilities in C, there were really no other tools to find security issues, other than manually reading and tracing code. Gaining a higher level understanding of the source code’s hierarchy and organization helped a lot in knowing where to begin looking. Identifying important files and functionalities to ispell were effective strategies because if an attacker were trying to get into the system, it would probably be an attempt to interfere with ispell’s functionality, which happens in the important files. Tangential features and/or files are less crucial to the system, and thereby less likely to have both exploitable and useful vulnerabilities. Doing research for common issues also helped a lot in knowing *what* to look for during code review.

This approach probably would have been more effective given more time and given more context into the organization of the codebase and access to documentation or firsthand sources behind the design of the system. I only had time to gain a general understanding of how the different files interacted with each other and entry-points into the system. A more in-depth (and laborious) manual code review of every single line of code would likely result in more potential security issues to delve into. However, simply identifying major files to focus on and looking for common vulnerabilities yielded a substantial number of issues to look into and fix already. None of the ones found via manual code review appeared to be fatal issues, but it’s possible that there is/are issues that will only be found with a deeper dive into the code and system design.

Dennis Zang

The original approach that I planned to take was to first look through the code manually for a general understanding, then do some live testing for potential errors using tools like AFL and gdb, then finally perform formal analysis using Frama-C using the errors I found. However, I did have to take detours due to various limitations.

For formal testing, as mentioned in the specs, it is expensive to perform the analysis (computation-wise and effort-wise) of the program. Given only about a weeks worth of time, reading through all of the source code, knowing how every variable and function in ispell interact, and annotating each function in at least 7 files with ACSL would prove to take too long in the given time. So, a more reasonable approach would be to find which errors came up in live testing, and prove those particular errors with formal testing to see where the code fails. However, our live testing only yielded one error that is simple enough that formal testing would prove unnecessary to demonstrate.

Of course, to get the detail for the actual bug, we need to run separate tools using the inputs logged by AFL. To to this, I had to run gdb and strace on multiple accounts, along with manually read through the code, as expected. Using debugging tools to understand the nature of the bug and program proved to be an additional step for manual code analysis. Of course, we found the

bug of mishandling the null character at the beginning of a string, but performing a formal analysis on a single bug and less than 20 lines of code would be superfluous.

While I would still argue that the planned approach would be reasonable in bug testing and fixing, the full procedure proved to be unnecessary in this lab. What I can say though is that the program is likely to have been well-tested (besides the obvious error of mishandling null characters), due to the lack of variety in the bugs we collaboratively found by live testing.

Supplementary materials

Notes on source code hierarchy

buildhash.c

```
  buildhash.c - make a hash table for ispell
  important functions:
    int main (int argc, char* argv[]));
    static void output();
    static void filltable();
    void* mymalloc(unsigned int size);
    void* myrealloc(void* ptr, unsigned int size, unsigned int oldisze);
    void myfree(void* ptr);
    static void readdict();
    static unsigned int newcount();
  files/buffers:
    char* Dfile; // dictionary file
    char* Hfile; // hash(output) file
    char* Lfile; // language file
    char Sfile[MAXPATHLEN]; // statistics file
```

tree.c

```
  tree.c - a hash style dictionary for user's personal words
  important functions:
    void treeinit(char* p, char* LibDict);
    static FILE* trydict(char* dictname, char* home, char* prefix, char*
suffix);
    static void treeload(FILE* dictf);
    void treeinsert(unsigned char* word, int wordlen, int keep);
    static struct dent* tinsert(struct dent* proto);
    struct dent* treelookup(ichar_t* word);
    void treeoutput();
    void* myalloc (void* ptr);
 files/buffers:
   static char personaldict[MAXPXATHLEN];
   static FILE* dictf;
```

parse.y

```
  yacc source file: program used to generate source code parser
```

defines tokens(eg. FLAG, SUFFIXES, WORDCHARS), types(file, headers, tables, rules)

fields.c

important functions:

field_t* fieldread(FILE* file, char* delims, int flags, unsigned int maxf);

reads one line of given file into buffer, break into fields, return to caller

field_t structure returned must eventually be freed with fieldfree()

field_t* fieldmake(.), field_t* fieldparse(.), int fieldwrite(.), void fieldfree(.);

local.h.bsd

sample #define statements, processed by shell scripts

ispell.c

ispell.c - An interactive spelling corrector.

functions:

static void usage();

int main();

static void dofile(char* filename);

static FILE* setupdefmt (char* filename, struct stat* statbuf);

static void update_file(char* filename, struct stat* statbuf);

static void expandmode(int printorig);

char* last_slash(char* file);

proto.h

xgets.c

xgets() is similar to gets(), except read from included files

README

ispell.1X

Magiclines

ijoin.c

"join" command, reimplements UNIX except fields can't be separated by newline, can handle unlimited line length

tgood.c

Table-driven good.c (which checks to see if a word/its root word in dict)

ispell.h

local.h.solaris

sq.c

munchlist.X

correct.c

Manages higher-level aspects of spell-checking

lookup.c

check if a word in the dictionary

addons/

addons/nextispell/

addons/nextispell/README

improved release of former internationalspell

need ispell to use this spell server (invokes ispell, communicates via pipe)

addons/nextispell/configure.h.template

addons/nextispell/services.template

addons/nextispell/Makefile

addons/nextispell/configure

addons/nextispell/configureTeX

addons/nextispell/nextispell.m

addons/nextispell/xspell.shar

unsq.c

icombine.c

icombine: combine multiple ispell dictionary entries into a single entry with the options of all entries

functions:

int main();

static void usage();

static void combinedict();

static void combineout();

exp_table.h

Makekit

Make an ispell distro kit.

deformatters/

deformatters/README

sample deformatters to write filters for ispell -F

deformatters/defmt-sh.c

deformatters/Makefile

deformatters/defmt-c.c

hash.c

hash.c - a simple hash function for ispell

Contributors

sq.1

man page for sq and unsq

sq: compresses a sorted list of words (a dictionary) [sq.c]

unsq: uncompress the output of sq [unsq.c]

defmt.c

Handle formatter constructs, mostly by scanning over them.

subset.X

combine and resolve dictionaries, proper subsets of each other

findaffix.X

find affixes (prefix or suffix) to use with ispell

Makepatch

ispell patch kit

config.X

configuration file for ispell

makedent.c

(makedent: make dictionary entry)

functions:

```
int makedent(unsigned char* lbuf, int lbuflen, struct dent* ent);
long whatcap(ichar_t* word);
int addvheader(struct dent* ent);
int combinecaps(struct dent* hdr, struct dent* oldp, struct dent* newp);
static int combine_two_entries(...).
void upcase/lowcase/chupcase();
static int issubset(struct dent* ent1, struct dent* ent2);
static void combineaffixes(struct dent* ent1, struct dent* ent2);
stringcharlen(), strtoichar(), ichartostr(),
```

languages/

these contain dictionaries and configuring different languages

languages/altamer/

languages/altamer/Makefile

languages/american/

languages/american/Makefile

languages/norsk/

languages/norsk/Makefile

languages/Where

languages/british/

languages/british/Makefile

languages/dansk/

languages/dansk/Makefile

languages/fix8bit.c

languages/francais/

languages/francais/Makefile

languages/deutsch/

languages/deutsch/Makefile

languages/Makefile

languages/svenska/

languages/svenska/Makefile

languages/nederlands/

languages/nederlands/Makefile

languages/english/

languages/english/altamer.2

languages/english/msgs.h

Messages header file: text strings written by any C program in ispell

languages/english/english.1

files with number extensions contain words for the dictionary

languages/english/altamer.0

languages/english/british.2

languages/english/altamer.1

languages/english/english.2
languages/english/british.0
languages/english/american.1
languages/english/english.aff
languages/english/makefile
languages/english/american.2
languages/english/english.0
languages/english/american.0
languages/english/english.3
languages/english/british.1
languages/portugues/
languages/portugues/Makefile
languages/espanol/
languages/espanol/Makefile
makedict.sh
Make a beginning dictionary file for ispell, using an existing speller

WISHES
things that are TODO in ispell. may contain security issues
Makefile
pc/
pc/djterm.c
pc/local.emx
pc/README
files to build Ispell on MS-DOS and MS-Windows systems.
pc/local.djgpp
pc/makeemx.bat
pc/cfgmain.sed
pc/cfglang.sed
pc/configdj.bat
pc/make-dj.bat
tryaffix.X
Try out affixes to see if they create valid roots
zapdups.X
local.h.macos
local.h.linux
term.c
deal with termcap, and unix terminal mode settings
fields.h
Structures used by the field-access package.
exp_table.c
expanded table implementation
good.c

good.c - see if a word or its root word is in the dictionary.

local.h.cygwin

iwhich

Report which version of a command is in use

version.h

ispell.5X

local.h.generic

CHANGES

fields.3

dump.c

Ispell's dump mode

References

1. https://fuzzing-project.org/
2. Zzuf - http://manpages.ubuntu.com/manpages/bionic/man1/zzuf.1.html
3. American Fuzzy Lop - http://lcamtuf.coredump.cx/afl/
4. cppcheck - http://cppcheck.sourceforge.net/
5. https://my.ece.utah.edu/~kstevens/ispell-faq.html
6. https://www.cvedetails.com/vulnerability-list/vendor_id-774/product_id-1323/Itcorp-Ispell.html
7. https://fossies.org/linux/misc/ispell-3.4.00.tar.gz/index_o.html