Robert Weiner

James Cogswell

CYSE 411-002

Term Project Phase 2

64 potential vulnerabilities were found. Of these, 37 were real vulnerabilities. This means a little over half of all potential vulnerabilities were real vulnerabilities. 41 of the potential vulnerabilities were found through the static analysis tools FlawFinder and Splint, and the remaining 23 were found through manual review. The static analysis tools we used were FlawFinder and Splint.

All Potential/Real Vulnerabilities Listed by Line Number (potential if not stated to be **REAL**):

58 (writeSock): while (byteswrote < len)

This is a potential vulnerability, but not a real one. Byteswrote is of type ssize\_t and len is of type size\_t. The implicit cast of this comparison does not have the potential for abuse as the len will never be negative as this argument is always the return of sizeof() or strlen() which return size\_t.

83 (readSock): while (byteswrote < len)

This is a potential vulnerability, but not a real one. Byteswrote is of type ssize\_t and len is of type size\_t. The implicit cast of this comparison does not have the potential for abuse as the len will never be negative as this argument is always the return of sizeof() or strlen() which return size\_t.

85 (readSock): ret = recv(sock, buf + bytesread, len - bytesread, 0);

This is a **REAL** vulnerability as recv does not append a null character if there is no space. The third argument to recv() is the max bytes to read, which here is len-bytesread. This would lead to recv() reading until the max size of the buffer, which would mean there would be no null terminator. Thus, the third argument to recv() should be len-bytesread-1 to ensure there is space. Additionally, before returning in readSock(), there should be a manual addition of the null terminator: buf[bytesread]=’\0’;

109 (writeArticle): char buf[1024];

This is not a vulnerability. While this can lead to vulnerabilities if improperly used, statically allocating buffers is not a vulnerability in itself.

110 (writeArticle): char path[1024];

This is not a vulnerability. While this can lead to vulnerabilities if improperly used, statically allocating buffers is not a vulnerability in itself.

115 (writeArticle): strcpy(path, ARTICLEPATH);

This is not a vulnerability as ARTICLEPATH is pre-defined and null terminated. If ARTICLEPATH was changed, it would be due to an arbitrary memory write vulnerability elsewhere in the program.

116 (writeArticle): strncat(path, &action[1], sizeof(path));

This is a **REAL** vulnerability because sizeof(path) is 1024 and it already has 12 bytes in path (ARTICLEPATH length + null byte) from the strcpy() func above. Can overwrite 12 bytes of memory.\*/

120 (writeArticle): fopen(&action[1], “w”);

This is a **REAL** vulnerability because the user can write a small malware c source file here to be compiled at line 329 to overwrite the ls command (or anything). This is also vulnerable to a symbolic link attack as the user has the ability to open any arbitrary file, and it's contents will be discarded, and then they can write anything within ~1kB. Can be used to crash the server, add themselves as a sudoer, etc.

154 (writeArticle): free(buf);

This is a **REAL** vulnerability as freeing a stack buffer can lead to undefined behavior, typically the program getting a segmentation fault and crashing. Thus, if any user were to use the command writeArticle, they could crash the program.

155 (writeArticle): free(path);

This is a **REAL** vulnerability as freeing a stack buffer can lead to undefined behavior, typically the program getting a segmentation fault and crashing. Thus, if any user were to use the command writeArticle, they could crash the program.

162 (readArticle): char buf[100];

This is a **REAL** vulnerability as fopen() later on is reading in 1000 bytes at a time, leading to a buffer overflow vulnerability. This can either be mitigated by changing the fopen() to read 100 bytes at a time, or by making this buffer 1000 bytes long.

163 (readArticle): char path[100];

This is a **REAL** buffer overflow vulnerability as action being fed in is 1024 bytes and path is only 100.

167 (readArticle): strcpy(path, ARTICLEPATH);

This is not a vulnerability as ARTICLEPATH is pre-defined and null terminated. If ARTICLEPATH was changed, it would be due to an arbitrary memory write vulnerability elsewhere in the program.

168 (readArticle): strcat(path, &action[1]);

This is a **REAL** vulnerability. This concatenates “./articles/” and the article name fed in by the user. This is a buffer overflow vulnerability since path is 1000 bytes, action is 1024, and path already contains the contents of ARTICLEPATH. Also, if action doesn't have null byte (due to vulnerability from readSock()); strcat doesn't check for this which can lead to unbounded string copy.

172 (readArticle): fopen(path, “r”);

This is a **REAL** vulnerability. It’s vulnerable to a user entering multiple ../ in their action to reach root, then can have any file they want shown to them, such as /etc/shadow. Also probably vulnerable to other attacks.

183 (readArticle): while(fgets(buf, 1000, file))

This is a **REAL** vulnerability because buf[] is 100 bytes and reads in 1000 bytes from file, leading to a buffer overflow vulnerability. Fgets() null terminates if the stated maximum bytes is reached, so there is no one-off error or unterminated string.

185 (readArticles): writeSock(sock, buf, strlen(buf));

This is not a real vulnerability as writeSock() will only read up until the length provided by strlen(buf). Even though strlen() does not include a null character in it’s count, this is not a vulnerability as the implementation of writeSock() negates this one-off vulnerability.

195 (listArticles): char buf[100];

This is not a real vulnerability as fgets() is reading in only the sizeof(buf)-1 at a time, so there is no potential for overflow.

196 (listArticles): memset(buf, 0, sizeof(buf));

This is not a real vulnerability as sizeof(buf) will return the total size of buf[], meaning there is no writing outside of array bounds.

204 (listArticles): system(LISTCOMMAND);

This is a **REAL** vulnerability because if there is an executable named “ls” in the local directory, it will be executed instead of normal “ls”. See writeArticle() and authenticate() vulnerabilities for how the attacker could do this.

206 (listArticles): fopen(“list.txt”, “r”);

This is a **REAL** vulnerability. If the custom ‘ls’ program the attacker creates writes a secret file to list.txt, this will open that file for printing to screen, mostly an extension to system() vulnerability above it on line 204.

208 (listArticles): while (fgets(buf, sizeof(buf)-1, list))

This is a **REAL** vulnerability. The return of fopen() is never checked, so if the file does not exist, fgets() will try to read from the address NULL. Thus, if an attacker (or even normal user) enters a file name they know does not exist, the program will crash from a segmentation fault.

210 (listArticles): writeSock(sock, buf, strlen(buf));

This is not a real vulnerability as writeSock() will only read up until the length provided by strlen(buf). Even though strlen() does not include a null character in it’s count, this is not a vulnerability as the implementation of writeSock() negates this one-off vulnerability.

220 (command): system(&action[1]);

This is a **REAL** vulnerability as it opens a vulnerability to insider attacks as an admin can execute ANY command from root level, not just those pertaining to this program.

226 (addUser): char buf[1024];

This is not a real vulnerability as the only thing that uses buf is snprintf() which only copies in sizeof(buf)-1 bytes, so there is no buffer overflow potential.

228 (addUser): strchr(&action[1], ‘:’);

This is not a real vulnerability because to enter the addUser() function they would already have administrator access and could therefore execute any arbitrary root command. An attacker attempting to exploit strchr() here would be far more complicated and unnecessary.

237 (addUser): snprintf(buf, sizeof(buf)-1, "echo %s > %s%s.txt", &p[1], USERPATH, &action[1]);

This is a **REAL** vulnerability because it is susceptible to a string formatting attack where the user can enter their own text as a command into action[]. Though, it is worth noting, only the admin can access this function, however if they themselves do not properly check the user’s provided username or password, this vulnerability can still occur.

Additional **REAL** vulnerability for this line: The max size of the formatted string is larger than the buffer, leading to truncation of the string after formatting as snprintf() stops transferring bytes after the max size provided. The return value from snprintf() should be checked compared to sizeof(buf) and if larger, the string is useless. If the formatted string were executed (currently there is no system() call with the string), then this would lead to an improper command being executed.

243 (adminFunctions): char action[1024];

This is not a real vulnerability as readSock() only reads in the amount of bytes that action can hold.

248 (adminFunctions): memset(action, 0, sizeof(action));

This is not a real vulnerability as sizeof(action) will return the total size of action[], meaning there is no writing outside of array bounds.

270 (userFunctions): char action[1024];

This is not a real vulnerability as readSock only reads in the amount of bytes that action can hold.

273 (userFunctions): strncmp(user, “admin”, 5)

This is a **REAL** vulnerability because it only checks the first five bytes of the user-defined username, and if those first five characters are ‘admin’ (e.g., if an attacker made their username adminhackerman or adminimizor), they would be granted admin privileges. There are no checks to prevent this from happening.

282 (adminFunctions): memset(action, 0, sizeof(action));

This is not a real vulnerability as sizeof(action) will return the total size of action[], meaning there is no writing outside of array bounds.

310 (authenticate): char search[512];

This is a **REAL** vulnerability since in line 328, the snprintf() function can overflow the search buffer due to the maximum bytes being read in set to sizeof(userfile)-1 and userfile[] is 1024 bytes. Therefore, search can be overflowed by 511 bytes.

311 (authenticate): char path[1024];

This is not a real vulnerability since it’s used in lines 336 and 339 where, in line 336, it limits the amount of data being written into the path buffer to sizeof(path)-1 and in line 339, it only opens the file for reading.

312 (authenticate): char userfile[1024];

This is not a real vulnerability since it’s used in line 327 where the limit of snprintf() is set to sizeof(userfile)-1.

313 (authenticate): char data[1024];

This is not a real vulnerability since it’s used in line 348 where the limit of fgets() is set to sizeof(data)-1.

317 (authenticate): memset(path, 0, sizeof(1024));

This is a **REAL** vulnerability since the function is setting only four bytes of the path[] array to zeros. This is known since sizeof(1024) returns 4 bytes due to 1024 being an integer and integers are 4 bytes in size. It should be noted that this is not as severe as other vulnerabilities in this program and is clearly not intended to work this way. The original programmers would have likely intended to include sizeof(path) here instead of sizeof(1024). To the untrained eye, this may seem to set 1024 bytes to zero, but that is not the case.

320 (authenticate): if (memcmp(pass, "baCkDoOr", 9) == 0)

This is a **REAL** vulnerability as this is a hard coded backdoor password. Any backdoor will be a vulnerability. In addition, if an attacker enters their password as “baCkDoOr” they would gain user privileges without even having an account, which they could then privilege escalate to admin using other vulnerabilities.

327 (authenticate): snprintf(userfile, sizeof(userfile)-1, “%s.txt”, user);

This is a **REAL** vulnerability because the user can perform a string format attack up to 1024 characters in size.

Additional **REAL** vulnerability: User input is not sanitized, thus they can do a command injection attack. Example: the user can enter into userfile “name`;sudo rm -rf --no-preserve-root /;” to remove everything on the machine.

Additional **REAL** vulnerability for this line: The max size of the formatted string is larger than the buffer, leading to truncation of the string after formatting as snprintf() stops transferring bytes after the max size provided. The return value from snprintf() should be checked compared to sizeof(userfile) and if larger, the string is useless. If the formatted string were executed (currently there is no system() call with the string), then this would lead to an improper command being executed.

328 (authenticate): snprintf(search, sizeof(userfile)-1, “stat %s`ls %s | grep %s`”, USERPATH, USERPATH, userfile);

This is a **REAL** buffer overflow vulnerability because the buffer is accessed out of bounds. The variable ‘search’ is 512 bytes and the data being read in from userfile is 1024 bytes. This line is also susceptible to string formatting attacks, however this attack would really take place on line 327. The fact the user entered data is not properly sanitized by this line also leads to the vulnerability in line 329, where the user can run a command injection attack.

Additional **REAL** vulnerability for this line: The max size of the formatted string is larger than the buffer, leading to truncation of the string after formatting as snprintf() stops transferring bytes after the max size provided. The return value from snprintf() should be checked compared to sizeof(userfile) and if larger, the string is useless. If the formatted string were executed (currently there is no system() call with the string), then this would lead to an improper command being executed.

329 (authenticate): system(search);

This is a **REAL** vulnerability due to the user’s input not being validated before snprintf() which leads to command injection vulnerability potential. The user gets to determine the user file being accessed so if they were to end the quote and insert a semicolon, they could write any command they wanted into the variable ‘search’. This would then be executed by the system(search) command. Example username that would execute command injection to compile the c code written to overwrite “ls” in the writeArticles() vulnerability mentioned above:

name` ; gcc -o ls ./articles/ls.txt;

336 (authenticate): snprintf(path, sizeof(path)-1, “%s%s”, USERPATH, userfile);

This is a **REAL** vulnerability to string format attacks. However, the attack would have already been executed on lines 327 or 328 by the time it reaches this point.

339 (authenticate): fopen(path, “r”);

This is not a real vulnerability because you can’t overwrite the file as it is opened in read mode and you can’t view the file as it is never printed to screen. You also couldn’t have it point to another user's file to gain authorized access because you'd need to know their password for the check afterwards on line 353. The userfile that is put into the path variable has to exist for execution to reach here, so in a way the user’s input has been validated before executing this line.

353 (authenticate): memcmp(data, pass, 3);

This is a **REAL** vulnerability because it’s only comparing the first three characters of the user’s input to the actual correct password. Therefore, an attacker would only need to brute force the first three characters of the password to gain access. For example, if the real password was ‘*HaRd2GuEsS!*’, then an attacker only needs to guess ‘*HaR*’ correctly to be granted access. This reduces the key space of the password from 951023 to 953. (there are 95 printable ascii characters, and the password can be up to 1023 bytes long due to data[1024])

372 (findarg): memcpy((char \*)&size, ptr1, 4);

This is not a real vulnerability because it’s only copying 4 bytes into a data type of size 8 bytes (size\_t data type). Thus, not vulnerable to buffer overflow.

382 (findarg): ptr1 += size;

This is a **REAL** vulnerability as this will lead to the program crashing every time this function is called. This is faulty logic, as it should be ptr1 += 4; as the program is trying to check every 4 bytes, but instead it is adding the integer representation of a four character block, which will almost certainly be a massive number leading to the program referencing address spaces it is not allowed to, leading to a crash.

390 (handleConnection): char buffer[1024];

This is not a real vulnerability because wherever the buffer is used in this function, the data being written to this buffer is restricted to a size that is less than 1024 bytes.

391 (handleConnection): char argbuf[1024];

This is not a real vulnerability because it’s only being read in data from buffer[] which is the same size [1024].

404 (handleConnection): memset(buffer, 0, sizeof(buffer));

This is not a real vulnerability as sizeof(buffer) will return the total size of buffer[], meaning there is no writing outside of array bounds.

418 (handleConnection): memcpy((char \*)&segmentcount, ptr1, 4);

This is not a real vulnerability because it’s only copying 4 bytes into a data type of size 8 bytes (size\_t data type). Thus, not vulnerable to buffer overflow.

432 (handleConnection): memset(argbuf, 0, sizeof(argbuf));

This is not a real vulnerability as sizeof(argbuf) will return the total size of argbuf[], meaning there is no writing outside of array bounds.

437 (handleConnection): memcpy((char \*)&segnext, ptr1, 4);

This is not a real vulnerability, though it is terrible and broken coding. This copies 4 chars into a size\_t variable which is large enough to hold them. However, this will lead to size\_t being a massive number when being accessed as a size\_t.

440 (handleConnection): memcpy((char \*)&argsize, ptr1, 4);

This is not a real vulnerability, though it does contribute to the real vulnerability in line 442. This is due to the same reason as line 440, however argsize is accessed as a size\_t in line 442.

442 (handleConnection): memcpy(ptr2, ptr1, argsize);

This is a **REAL** vulnerability because argsize is a massive number generated by line 440. This memcpy copies that many bytes which will be far more bytes than the program space holds. This will lead to the OS terminating the program for accessing address space it is not authorized to access.

449 (handleConnection): user = findarg(argbuf, USERNAME);

This is a **REAL** vulnerability. The return from findarg() is never checked to see if it is NULL. If the username is not found, a NULL pointer will be returned, which when fed into the snprintf() on line 452 will lead to a segmentation fault. Thus, the user is able to crash the program by entering improper input.

450 (handleConnection): pass = findarg(argbuf, PASSWORD);

This is a **REAL** vulnerability. The return from findarg() is never checked to see if it is NULL. If the username is not found, a NULL pointer will be returned, which when fed into the snprintf() on line 452 will lead to a segmentation fault. Thus, the user is able to crash the program by entering improper input.

452 (handleConnection): snprintf(buffer, sizeof(buffer)-1, “User attempting to authenticate %s”, user);

This is a **REAL** string format vulnerability because the user has direct control over the input to string formatting and can memory write.

472 (handleConnection): snprintf(buffer, sizeof(buffer)-1,"user: %s failed to login with password %s", user, pass);

This is a **REAL** string format vulnerability as the user has direct control over the input to string formatting, and thus can memory write.

487 (mainLoop): struct sockaddr\_in \*client = (struct sockaddr\_in\*)malloc(sizeof(struct sockaddr\_in));

This is a potential vulnerability as the return of malloc() is not checked for a NULL pointer. This is not a real vulnerability as even if this leads to a segmentation fault, the user does not have the ability to interact with the program yet.

588 (main): fopen(“logfile.txt”, w”);

This is not a real vulnerability because this part executes long before a user is able to manipulate it. It’s a potential vulnerability because fopen() is sometimes vulnerable to race conditions, symlink attacks, force open device files, etc. but in this context it is not vulnerable.

619 (logData): char buffer[4096];

This is not a real vulnerability as 4096 bytes is more than enough space to hold any arguments that are being passed into logData at any place in the program, so there is no buffer overflow potential.

622 (logData): vsnprintf(buffer, sizeof(buffer)-1, format, arguments);

This is a **REAL** vulnerability as the user can conduct a string format attack. Generally this attack would be executed at other lines before logData() is called, however it is possible there are lines where the user input could reach here before another string format vulnerability is reached.

624 (logData): fprintf(logfile, "LoggedData [Proccess:%i]: %s\n", getpid(), buffer);

This is a **REAL** string format vulnerability as the user has direct control over what is placed into buffer[] in many cases of logData() being called, however this vulnerability would have really been executed on line 622 as that string format vulnerability comes first in execution.

625 (logData): fflush(logfile);

This is a **REAL** vulnerability as logfile is not checked for if it is a NULL pointer before executing this line. Due to the buffer overflow and string format vulnerabilities found elsewhere in the program, there is a real possibility of logfile being NULL. If it is, all output streams are flushed, not just logfile, or even just ones in the program space. The program likely would have crashed on line 624 due to this possibility, but there is a chance execution could have gotten to this line anyway.