Project: Secure Coding – Vulnerability Scanning & Analysis

The purpose of this project is to use static analysis tools in order to find potential vulnerabilities within the code. In this project, two different static analysis programs will be used to scan for potential vulnerabilities: Cppcheck and Visualcodegrepper. The results of these two programs will be used to compare any similarities or differences of the potential vulnerabilities identified between Cppcheck and Visualcodegrepper. In addition, each potential vulnerability will include an in-depth analysis to determine whether it is a potential vulnerability or not.

1. Static Analysis Tools used:
Cppcheck and Visualcodegrepper

Cppcheck: This was the first tool we decided to use to perform the static analysis. The Cppcheck installation didn't really prompt us to select options, so the analysis was performed with the default settings, which was recommended on the website. The default settings Cppcheck used for this vulnerability scanning include checking for multiple types of errors (memory leaks, resource leaks), common warnings (undefined behaviors), and portability warnings.

Visualcodegrepper: This was the second tool we decided to use, but we were debating between using either this or IKOS. There were multiple options for this tool, but we went with the one which does a full scan, meaning it will look for comments, code errors and anything which could be a vulnerability (doesn't mean it will find 100% of them).

2. Why we chose these tools:

Cppcheck: We chose Cppcheck as it was the highest reviewed static analysis tool which we ran into while researching which tool to use. As advertised, it has a strong focus on detecting undefined behaviors, which could be useful to detect more potential vulnerabilities compared to other tools that detect more obvious errors and bugs. The undefined behaviors range from Dead pointers, division by zero, integer overflows to null pointer dereferences, out of bounds checking and uninitialized variables. Thus, we didn't really have to add anything extra or select certain settings since the tool, at default settings, already had everything configured. It focuses on finding only certain types of bugs and the developers recommend using other tools to find problems in other areas, which is usually the norm.

Visualcodegrepper: This was the second tool we chose after Cppcheck to focus on areas which Cppcheck doesn't really look over. Visualcodegrepper focuses on finding any bad functions, attempts to find range of around 20 common phrases within comments such as "ToDo" and "FixMe" (which could be potential vulnerabilities and they could contain hardcoded backdoors etc.), and a quality of life addition of a pie chart which shows the relative proportions of the code. This tool had multiple scan options/settings to use for analysis such as Code only, comments only, etc., but we chose to run a full scan since the scope of this project is to find as many potential vulnerabilities as possible within this code. Fullscan consists of scanning Comments, Code, and Dangerous Functions.

Screenshot of the analysis from Cppcheck:

CYSE 411 Project\cyse411project1.c	portability	56	Undefined behaviour, pointer arithmetic 'buf+byteswrote' is out of bounds.	3/30/2021
CYSE 411 Project\cyse411project1.c	error	177	Buffer is accessed out of bounds: buf	3/30/2021
CYSE 411 Project\cyse411project1.c	error	327	Buffer is accessed out of bounds: search	3/30/2021
CYSE 411 Project\cyse411project1.c	error	563	Resource leak: sock	3/30/2021
CYSE 411 Project\cyse411project1.c	error	569	Resource leak: sock	3/30/2021
CYSE 411 Project\cyse411project1.c	error	575	Resource leak: sock	3/30/2021
CYSE 411 Project\cyse411project1.c	warning	490	Size of pointer 'client' used instead of size of its data.	3/30/2021
CYSE 411 Project\cyse411project1.c	warning	316	Suspicious usage of 'sizeof' with a numeric constant as parameter.	3/30/2021

Results from the analysis done by Visualcodegrepper are attached as a separate text file (vcg.txt).

3. Potential Vulnerabilities:

Cppcheck

a. Buffer out of bounds: buf

CYSE 411 Project\cyse411project1.c	error	177	Buffer is accessed out of bounds: buf	3/28/2021

```
153  void readArticle(int sock, FILE *logfile, char *action)
154  {
155      FILE *file;
156      char buf[100];
157      char path[100];     //error: char path[100;
158
174      /* fgets for the size of the buffer (100), from the file
175         writing the article to the user each time! */
176
177      while (fgets(buf, 1000, file))
178      {
179          writeSock(sock, buf, strlen(buf));
180      }
```

Analysis: **Line 177** is a potential vulnerability due to how **fgets()** is used in this code. In **line 177**, **fgets()** allows an input size at most 1000 bytes of data within a file into **buf[100]** shown in **line 156**, which can store up to 100 bytes at most. Although **fgets()** has the capability to prevent buffer overflows by restricting the amount of data being inputted into the buffer via its second parameter (**int**), it would not work in this case as the maximum limit the user can input data into **buf[100]** is 1000 bytes, thus a buffer overflow can occur if the user is trying to input data from a file that contains more than 100 bytes.

Fix (Line 177): fgets(buf, **1000**, file); → fgets(buf, **sizeof(buf)-1**, file);

```c
    /* fgets for the size of the buffer (100), from the file
       writing the article to the user each time! */

    while (fgets(buf, sizeof(buf)-1, file))
    {
        writeSock(sock, buf, strnlen_s(buf, sizeof(buf)-1));
    }
```

This fix changes the maximum input size from 1000 bytes to the maximum size of **buf[100]** for fgets(), which is 99 bytes. This fix will mitigate vulnerability (a.) by preventing buffer overflows from occurring in **buf[100]**. Since the maximum input size is now equal to one less of the maximum size of what **buf[100]** can store (extra buffer space for the null character), buffer overflows cannot occur as the user is not able to input data outside the boundaries of **buf[100]**.

b. Buffer out of bounds: search

CYSE 411 Project\cyse411project1.c error 327 Buffer is accessed out of bounds: search 3/28/2021

```c
306 /* return 1 for success, 2 on bad username, 3 on bad password */
307 int authenticate(FILE *logfile, char *user, char *pass)
308 {
309     char search[512];
310     char path[1024];
311     char userfile[1024];
312     char data[1024];
313     FILE *file;
314     int ret;
324     /* look up user by checking user files: done via system() to /bin/ls|grep user */
325     logData(logfile, "performing lookup for user via system()!\n");
326     snprintf(userfile, sizeof(userfile)-1, "%s.txt", user);
327     snprintf(search, sizeof(userfile)-1, "stat %s`ls %s | grep %s`", USERPATH, USERPATH, userfile);
328     ret = system(search);
```

Analysis: Line 327 is a potential vulnerability because of how **snprintf()** is used in this code. In **line 327**, **snprintf()** allows an input size of at most 1024 bytes of data based on the buffer size of **userfile[1024]** (**line 311**) into **search[512]** (**line 309**), which can store up to 512 bytes at most. Although **snprintf()** has the capability to prevent buffer overflows by restricting the amount of data being inputted into a buffer via its second parameter (**size_t**), it would not work in this case as the maximum amount of bytes the user can input data into **search[512]** is 1024, thus a buffer overflow can occur if the user is trying to input data from **userfile[1024]** via the method parameter **char *user (line 307)** that contains more than 512 bytes.

Fix (Line 327): snprintf(search, **sizeof(userfile)-1**, “stat %s`ls %s | grep %s`", USERPATH, USERPATH, userfile); → snprintf(search, **sizeof(search)-1**, “stat %s`ls %s | grep %s`", USERPATH, USERPATH, userfile);

```c
324     /* look up user by checking user files: done via system() to /bin/ls|grep user */
325     logData(logfile, "performing lookup for user via system()!\n");
326     snprintf(userfile, sizeof(userfile)-1, "%s.txt", user);
327     snprintf(search, sizeof(search)-1, "stat %s`ls %s | grep %s`", USERPATH, USERPATH,
        userfile);
328     ret = system(search);
```

This fix changes the maximum input size from 1024 bytes (from **userfile[1024])** to 512 bytes (from **search[512]**) for **snprintf()**. This fix will mitigate vulnerability (b.) by preventing buffer overflows from occurring in **search[512]**. Since the maximum input size is now equal to the maximum size of what **search[512]** can store, buffer overflows cannot occur as the user is not able to input data outside the boundaries of **search[512]**.

c. Resource Leaks

CYSE 411 Project\cyse411project1.c	error	563 Resource leak: sock	3/28/2021
CYSE 411 Project\cyse411project1.c	error	569 Resource leak: sock	3/28/2021
CYSE 411 Project\cyse411project1.c	error	575 Resource leak: sock	3/28/2021

```
533 int setupSock(FILE *logf, unsigned short port)
534 {
535     int sock = 0;
536     struct sockaddr_in sin;
537     int opt = 0;
538 
539     if (signal(SIGCHLD, spawnhandler) == SIG_ERR)
540     {
541         perror("fork() spawn handler setup failed!");
542         return -1;
543     }
544 
545     memset((char *)&sin, 0, sizeof(sin));
546 
547     sin.sin_family = AF_INET;
548     sin.sin_port = htons(port);
549 
550     sock = socket(AF_INET, SOCK_STREAM, 0);
551 
552     if (sock == -1)
553     {
554         logData(logf, "socket() failed");
555         return -1;
556     }
557 
558     opt = 1;
```

Line 563:

```
560     if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) == -1)
561     {
562         logData(logf,"setsockopt() failed");
563         return -1;
564     }
```

Line 569:

```
566     if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) == -1)
567     {
568         logData(logf, "bind() failed");
569         return -1;
570     }
```

Line 575:

```
572     if (listen(sock, 10) == -1)
573     {
574         logData(logf, "listen() failed");
575         return -1;
576     }
```

Analysis: **Lines 563, 569, and 575** may not be potential vulnerabilities due to the fact that the socket connection is not closed if the setup fails. Although it will not cause a buffer overflow, malicious users can still access the socket connection that was left open and gain certain information about the system, which could be dangerous. If anything, leaving a socket connection open could cause a DoS of said system due to the wasted resources being used to maintain the failed socket connection.

d. Different variable usage

CYSE 411 Project\cyse411project1.c warning 490 Size of pointer 'client' used instead of size of its data. 3/28/2021

```
485     int clientfd = 0;   //error: Int clientfd = 0;
486     struct sockaddr_in *client = (struct sockaddr_in*)malloc(sizeof(struct sockaddr_in));
487     socklen_t clientlen = 0;
488     pid_t offspring = 0;
489
490     memset(client, 0, sizeof(client));
```

Analysis: **Line 490** may not be a potential vulnerability because the **client** is a defined pointer within the code **(line 486)** and has a predetermined value declared to it. **memset()** requires the user to input three arguments, **str**, **int** and **size** which represent the pointer to block of memory to fill, the value to be set and the number of bytes to be set respectively. In this case, the pointer **client** is being used with the value set to 0. However, **sizeof()** argument only takes an arbitrary data type as an input, so using **client** in this case will only allocate the size of a pointer, which is 4 bytes in a 32-bit system, and not the size of the data pointed to by **client**. A user cannot exploit this since the pointer is not an input, and they cannot change it unless they get access to the source code itself. However, it is a bad practice for coding.

Fix (Line 490): memset(client, 0, **sizeof(client)**); → memset(client, 0, **sizeof(&client)**);

```
485     int clientfd = 0;   //error: Int clientfd = 0;
486     struct sockaddr_in *client = (struct sockaddr_in*)malloc(sizeof(struct sockaddr_in));
487     socklen_t clientlen = 0;
488     pid_t offspring = 0;
489
490     memset(client, 0, sizeof(&client));
```

This fix changes the argument for **sizeof()** from a pointer to the data pointed by the pointer. This fix will mitigate a bug that exists in (d.) by providing the entire memory of **client** in order for **memset()** to be used properly. In this case, the fix will allow **memset()** to "clear" the memory of **client** by setting all existing values within **client** to 0.

e. Numeric constant sizeof() parameter

CYSE 411 Project\cyse411project1.c warning 316 Suspicious usage of 'sizeof' with a numeric constant as parameter. 3/28/2021

```
307 int authenticate(FILE *logfile, char *user, char *pass)
308 {
309     char search[512];
310     char path[1024];
311     char userfile[1024];
312     char data[1024];
313     FILE *file;
314     int ret;
315
316     memset(path, 0, sizeof(1024));
```

```
    /* open file and check if contents == password */
    file = fopen(path, "r");

    if (!file)
    {
        logData(logfile, "fopen for userfile failed\n");
        return 2;
    }
```

Analysis: **Line 316** may not be a potential vulnerability because similar to how **memset()** was misused in (d.), the argument used as input for **sizeof()** in this case is just an **int** value, so the **sizeof()** would allocate only **4 bytes** rather than the entire **1024**. The size of the char array **path[1024]** is larger than 4 bytes, so no buffer overflow can occur. It also cannot be exploited as it is hard coded within the function itself. However, it is a bug since **path[1024]** is used for authenticating the user by reading the user's file to check for the right password. Since **memset()** does not "clear" the entire **path[1024]** buffer by setting the entire buffer data to 0 and instead only "clear" the first four bytes of **path[1024]** in this case, it may cause issues with authenticating future users. For instance, if the previous user's data remains leftover in **path[1024]** when the next user authenticates, in the event where the next user's data does not override all of the previous user's data and instead the two data merges together, the program may fail to authenticate the next user due to concatenating the previous user's leftover data with the current user's data.

Fix (Line 316): memset(path, 0, **sizeof(1024)**); → memset(path, 0, **sizeof(path)**);

```
307 int authenticate(FILE *logfile, char *user, char *pass)
308 {
309     char search[512];
310     char path[1024];
311     char userfile[1024];
312     char data[1024];
313     FILE *file;
314     int ret;
315 
316     memset(path, 0, sizeof(path));
```

This fix changes the argument for **sizeof()** from an integer to a buffer (**path[1024]**). This fix will mitigate a bug that exists in (e.) by providing the entire memory of **path[1024]** in order for **memset()** to be used properly. In this case, the fix will allow **memset()** to "clear" the memory of **path[1024]** by setting all existing values within **path[1024]** to 0.

Visualcodegrepper

1 - strcpy():

MEDIUM: Potentially Unsafe Code - strcpy
Line: 111 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions.

```
        strcpy(path, ARTICLEPATH);
```

MEDIUM: Potentially Unsafe Code - strcpy

Line: 161 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions.

```
strcpy(path, ARTICLEPATH);
```

<u>Parts of code where the error is identified:</u>

```
101        FILE *file;
102        char *p;
103        size_t x, y;
104        int complete = 0;
105        char buf[1024];      //error: chr buf[1024];
106        char path[1024];
107
108        // char* buf  = (char*)calloc(1024, sizeof(char));
109        // char* path = (char*)calloc(1024, sizeof(char));
110
111        strcpy(path, ARTICLEPATH);
```

```
155        FILE *file;
156        char buf[100];
157        char path[100];      //error: char path[100;
158
159        logData(logfile, &action[1]);
160
161        strcpy(path, ARTICLEPATH);
```

<u>In line 28, ARTICLEPATH is defined, making it a constant throughout the code.</u>

```
27    #define USERPATH "./users/"
28    #define ARTICLEPATH "./articles/"
29    #define LISTCOMMAND "ls ./articles/ > list.txt"
```

<u>Analysis:</u> **Lines 111 and 161** may not be potential vulnerabilities because even though the function **strcpy()** can be exploited to cause buffer overflow conditions, the usage of **strcpy()** within this method itself cannot really be exploited as the arguments being used in this case are **path[100] (for line 157)** and **path[1024] (for line 106)**, which are defined character arrays, and **ARTICLEPATH**, which is a defined constant within the code. **strcpy()** is vulnerable to buffer overflows due to the function having no restrictions on the amount of data being inputted into the buffer since the string to be copied can be larger than the size of the target buffer, thus leading to an overflow. To cause a buffer overflow in the case for **path[100]** in **line 157**, one will need to input data larger than **path[100]**, thus the user will need to input data larger than 100 bytes into **path[100]** as that is the most it can store. However, **ARTICLEPATH** is a defined constant within the code in **line 28** to which its data size is 11 bytes (not including the null character), thus it will fit into **path[100]** as an input. In addition, the input will always remain constant to which the user will not be able to modify **ARTICLEPATH** unless they gain access to the source code.

<u>Fix (Lines 111 and 161):</u> **strcpy**(path, ARTICLEPATH); → **strncpy**(path, ARTICLEPATH, **sizeof(path)**);

```
101        FILE *file;
102        char *p;
103        size_t x, y;
104        int complete = 0;
105        char buf[1024];      //error: chr buf[1024];
106        char path[1024];
107
108        // char* buf  = (char*)calloc(1024, sizeof(char));
109        // char* path = (char*)calloc(1024, sizeof(char));
110
111        strncpy(path, ARTICLEPATH, sizeof(path));
112        strncat(path, &action[1], sizeof(path)-11);
```

```
155        FILE *file;
156        char buf[100];
157        char path[100];      //error: char path[100;
158
159        logData(logfile, &action[1]);
160
161        strncpy(path, ARTICLEPATH, sizeof(path));
162        strncat(path, &action[1], sizeof(path)-11);
```

This fix changes the function used from **strcpy()** to **strncpy()** and adds an input size restriction of the two **path** buffers via **sizeof(path)** (100 bytes for **path[100]**; 1024 bytes for **path[1024]**. This fix will aim to resolve bad coding practices that exists in #1 by avoiding the use of **strcpy()** preventing buffer overflows from occurring in both **path** buffers.

2 - strncat():

STANDARD: Potentially Unsafe Code - strncat

Line: 112 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions.

```
    strncat(path, &action[1], sizeof(path));
```

```
101        FILE *file;
102        char *p;
103        size_t x, y;
104        int complete = 0;
105        char buf[1024];      //error: chr buf[1024];
106        char path[1024];
107
108        // char* buf  = (char*)calloc(1024, sizeof(char));
109        // char* path = (char*)calloc(1024, sizeof(char));
110
111        strcpy(path, ARTICLEPATH);
112        strncat(path, &action[1], sizeof(path));
```

In line 269, the character array action[1024] is defined

```c
int userFunctions(FILE *logfile, int sock, char *user)
{
    char action[1024];
    size_t len;

    if (0 == strncmp(user, "admin", 5))
    {
        adminFunctions(logfile, sock);
        return 0;
    }
```

Example of action being used as a user-defined input. Can be seen being used on line 214 in system().

```c
211  void command(FILE *log, int sock, char *action)
212  {
213      logData(log, "executing command: %s", &action[1]);
214      system(&action[1]);
215  }
```

Analysis: **Line 112** is a potential vulnerability. Compared to **strcat()**, which does not check for the size of the data to be copied, **strncat()** requires a third argument, “count”, which requires the user to specify the size/limit of the data input to be copied. However, in this case, **path[1024]** already has 11 bytes of data stored beforehand from **strcpy(path, ARTICLEPATH)**, thus the use of **strncat()** with the “count” parameter being **sizeof(path)** will cause a buffer overflow due to how the user-defined data input, **&action[1]**, can be at most 1024 bytes (from **sizeof(path)**) but only 1013 bytes of **path[1024]** buffer space remains due to **ARTICLEPATH** taking the first 11 bytes of **path[1024]**. This can be exploited by a user through inputting **&action[1]** to concatenate a string larger than 1013 bytes, which is the amount of space available in the target buffer **path[1024]**, which can cause an overflow. In addition, **&action[1]** can be seen as a user-defined input in **line 214** within **system()**. As **system()** is within the **void command()** method, one can observe that **system()** is trying to execute a command, which requires users to input a string in order for the command to be executed.

Fix (Line 112): strncat(path, &action[1], **sizeof(path)**); → strncat(path, &action[1], **sizeof(path)-11**);

```c
101      FILE *file;
102      char *p;
103      size_t x, y;
104      int complete = 0;
105      char buf[1024];     //error: chr buf[1024];
106      char path[1024];
107
108      // char* buf  = (char*)calloc(1024, sizeof(char));
109      // char* path = (char*)calloc(1024, sizeof(char));
110
111      strcpy(path, ARTICLEPATH);
112      strncat(path, &action[1], sizeof(path)-11);
```

This fix changes the input size restriction from 1024 bytes (the entire buffer size of **path[1024]**) to 1013 bytes for **strncat()**. This fix will mitigate vulnerability #2 by preventing buffer overflows from occurring in **path[1024]**. Because there is existing data in the first 11 bytes of **path[1024]** from **ARTICLEPATH**, **path[1024]** has 1013 bytes of buffer space left, thus **strncat()** will concatenate the input data and the existing data in **path[1024]** with the remaining buffer space. Since there is 1013 bytes left of buffer space within **path[1024]**, restricting the input size to **sizeof(path)-11** will limit the maximum input size to 1013 bytes in which the input data will fill the entire **path[1024]** if necessary, thus buffer overflows cannot occur as the user will not be able to input data outside the boundaries of **path[1024]**.

3 - strcat():

MEDIUM: Potentially Unsafe Code - strcat

Line: 162 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions.

```
strcat(path, &action[1]);
```

```
155     FILE *file;
156     char buf[100];
157     char path[100];     //error: char path[100;
158
159     logData(logfile, &action[1]);
160
161     strcpy(path, ARTICLEPATH);
162     strcat(path, &action[1]);
```

```
267 int userFunctions(FILE *logfile, int sock, char *user)
268 {
269     char action[1024];
270     size_t len;
271
272     if (0 == strncmp(user, "admin", 5))
273     {
274         adminFunctions(logfile, sock);
275         return 0;
276     }
```

Analysis: **Line 162** is a potential vulnerability because compared to the usage of **strncat()**, there is no restriction on the amount of bytes to be written, so the usage of **strcat()** in this context can be used to facilitate a buffer overflow condition by inputting a string to be concatenated that is larger than the size of the target buffer (in this case, **path[100]**, which can store up to 100 bytes). A user can exploit this by inputting **&action[1] (explanation to why this is a user-defined function is mentioned in the above part regarding strncat())** to concatenate a string larger than 100 bytes, which is the amount of space available in the target buffer **path[100]**, to cause an overflow.

Fix (Line 162): **strcat**(path, &action[1]); → **strncat**(path, &action[1], **sizeof(path)-11**);

```
155     FILE *file;
156     char buf[100];
157     char path[100];     //error: char path[100;
158
159     logData(logfile, &action[1]);
160
161     strcpy(path, ARTICLEPATH);
162     strncat(path, &action[1], sizeof(path)-11);
```

This fix changes the function used from **strcat()** to **strncat()** and adds an input size restriction of 89 bytes. This fix will mitigate vulnerability #3 by preventing buffer overflows from occurring in **path[100]**. Similar to vulnerability #2, there is existing data in the first 11 bytes of **path[100]** from **ARTICLEPATH**, thus **path[100]** has 89 bytes of buffer space left. Since there is 89 bytes left of buffer space within **path[100]**, restricting the input size to **sizeof(path)-11** will limit the maximum input size to 89 bytes in which the input data will fill the entire **path[100]** if necessary, thus buffer overflows cannot occur as the user will not be able to input data outside the boundaries of **path[100]**.

4 - fopen():

STANDARD: Potentially Unsafe Code - fopen
Line: 116 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
Function used to open file. Carry out a manual check to ensure that user cannot modify filename for malicious purposes and that file is not 'opened' more than once simultaneously.
```
file = fopen(&action[1], "w");
```

STANDARD: Potentially Unsafe Code - fopen
Line: 166 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
Function used to open file. Carry out a manual check to ensure that user cannot modify filename for malicious purposes and that file is not 'opened' more than once simultaneously.
```
file = fopen(path, "r");
```

STANDARD: Potentially Unsafe Code - fopen
Line: 200 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
Function used to open file. Carry out a manual check to ensure that user cannot modify filename for malicious purposes and that file is not 'opened' more than once simultaneously.
```
list = fopen("list.txt", "r");
```

STANDARD: Potentially Unsafe Code - fopen
Line: 338 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
Function used to open file. Carry out a manual check to ensure that user cannot modify filename for malicious purposes and that file is not 'opened' more than once simultaneously.
```
file = fopen(path, "r");
```

STANDARD: Potentially Unsafe Code - fopen
Line: 587 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
Function used to open file. Carry out a manual check to ensure that user cannot modify filename for malicious purposes and that file is not 'opened' more than once simultaneously.
```
logf = fopen("logfile.txt", "w");
```

```
108     // char* buf  = (char*)calloc(1024, sizeof(char));
109     // char* path = (char*)calloc(1024, sizeof(char));
110
111     strcpy(path, ARTICLEPATH);
112     strncat(path, &action[1], sizeof(path));
113
114     logData(logfile, "user writing article: %s", path);
115
116     file = fopen(&action[1], "w");
```

```
161     strcpy(path, ARTICLEPATH);
162     strcat(path, &action[1]);
163
164     logData(logfile, "user request to read article: %s", path);
165
166     file = fopen(path, "r");
```

```
194     /* i wish i had more time! i wouldnt have to write
195        this code using system() to call things! */
196
197     memset(buf, 0, sizeof(buf));
198     system(LISTCOMMAND);
199
200     list = fopen("list.txt", "r");
```

```
335     snprintf(path, sizeof(path)-1, "%s%s", USERPATH, userfile);
336
337     /* open file and check if contents == password */
338     file = fopen(path, "r");
```

```
583     int sock;
584     FILE *logf;
585
586     /* setup log file */
587     logf = fopen("logfile.txt", "w");
```

Analysis: **Lines 116, 166, 200, 338, and 587** may not be potential vulnerabilities because the path is coded to point to **ARTICLEPATH**, which cannot be changed without accessing the source code. This narrows down the user to only read files present inside the **ARTICLEPATH**, which only contain articles written by other users. In order to exploit this, a user must be able to change the path to point towards another directory that would contain more sensitive information such as password files; however, in this case, the user cannot modify the directory path through **strcat()**.

5 - strlen():

STANDARD: Potentially Unsafe Code - strlen

Line: 179 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. For critical applications, particularly applications accepting anonymous Internet connections 'wraparound' errors.

```
writeSock(sock, buf, strlen(buf));
```

STANDARD: Potentially Unsafe Code - strlen

Line: 204 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. For critical applications, particularly applications accepting anonymous Internet connections 'wraparound' errors.

```
writeSock(sock, buf, strlen(buf));
```

```
174     /* fgets for the size of the buffer (100), from the file
175        writing the article to the user each time! */
176
177     while (fgets(buf, 1000, file))
178     {
179         writeSock(sock, buf, strlen(buf));
180     }
181
182     fclose(file);
183
184     return;
185 }
```

```
194     /* i wish i had more time! i wouldnt have to write
195        this code using system() to call things! */
196
197     memset(buf, 0, sizeof(buf));
198     system(LISTCOMMAND);
199
200     list = fopen("list.txt", "r");
201
202     while (fgets(buf, sizeof(buf)-1, list))
203     {
204         writeSock(sock, buf, strlen(buf));
205     }
```

In line 156, the character array buf[100] is declared, which has 100 bytes of space available.

```
153 void readArticle(int sock, FILE *logfile, char *action)
154 {
155     FILE *file;
156     char buf[100];
157     char path[100];     //error: char path[100;
```

Analysis: Lines 179 and 204 may not be potential vulnerabilities because the length of **buf[100]** was already predetermined on **line 156**. The **strlen()** command uses **buf[100]** as the maximum size to check the length of the string used during the call for **writeSock()** method. The parameter **strlen(buf)** restricts the string to be less than 100 bytes (99 since we do not consider the NULL character), and since integer overflow occurs if the parameter is larger than the allocated memory space, which in this case, it cannot happen since the size of the parameter **buf** was already predetermined. There is no way to exploit this as the size of **buf[100]** is declared within the **readArticle()** method itself, so it is hard coded. No user can change the size of **buf[100]** without getting access to the source code itself.

Fix for Line 177 (bad coding practice): fgets(buf, **1000**, file); → fgets(buf, **sizeof(buf)-1**, file), was already done in **(a.)** of the project.

Fix (Lines 179 and 204): **strlen**(buf) → **strnlen_s**(buf, sizeof(buf)-1)

```
174     /* fgets for the size of the buffer (100), from the file
175        writing the article to the user each time! */
176
177     while (fgets(buf, sizeof(buf)-1, file))
178     {
179         writeSock(sock, buf, strnlen_s(buf, sizeof(buf)-1));
180     }
```

```c
194     /* i wish i had more time! i wouldnt have to write
195        this code using system() to call things! */
196
197     memset(buf, 0, sizeof(buf));
198     system(LISTCOMMAND);
199
200     list = fopen("list.txt", "r");
201
202     while (fgets(buf, sizeof(buf)-1, list))
203     {
204         writeSock(sock, buf, strnlen_s(buf, sizeof(buf)-1));
205     }
```

This fix changes the function used from **strlen()** to **strnlen_s()**, which adds an input size restriction of 99 bytes to **buf[100]**. This fix aims to resolve bad coding practices that exists in #5 with using **strlen()** as the function does not have any restrictions to ensure a null character will be present in **buf[100]**. **strnlen_s()** provides a second parameter that restricts the input size of the string in order to ensure a null character is present to prevent overflow from occurring.

6 - Suspicious Comment:

SUSPICIOUS COMMENT: Comment Indicates Potentially Unfinished Code -
Line: 318 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
`FIXME: hard coded admin backdoor for password recovery */`

```c
318     /* FIXME: hard coded admin backdoor for password recovery */
319     if (memcmp(pass, "baCkDoOr", 9) == 0)
320     {
321         return 1;
322     }
```

Analysis: Line 319 is a potential vulnerability because there is a hard-coded backdoor implemented into the code. If the malicious user is able to view the source code or obtain the hard-coded password via other means through the system, it will be able to bypass all security measures and access the system.

Fix: Removing the hard-coded backdoor password is the best way to mitigate this vulnerability.

7 - system():

MEDIUM: Potentially Unsafe Code - Application Variable Used on System Command Line
Line: 198 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
The application appears to allow the use of an unvalidated variable when executing a system command.
`system(LISTCOMMAND);`

MEDIUM: Potentially Unsafe Code - Application Variable Used on System Command Line
Line: 214 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c
The application appears to allow the use of an unvalidated variable when executing a system command.
`system(&action[1]);`

MEDIUM: Potentially Unsafe Code - Application Variable Used on System Command Line

Line: 328 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

The application appears to allow the use of an unvalidated variable when executing a system command.

```
ret = system(search);
```

```
189     char buf[100];
190     FILE *list;
191
192     logData(logfile, "user has requested a list of articles");
193
194     /* i wish i had more time! i wouldnt have to write
195        this code using system() to call things! */
196
197     memset(buf, 0, sizeof(buf));
198     system(LISTCOMMAND);
211 void command(FILE *log, int sock, char *action)
212 {
213     logData(log, "executing command: %s", &action[1]);
214     system(&action[1]);
324     /* look up user by checking user files: done via system() to /bin/ls|grep user */
325     logData(logfile, "performing lookup for user via system()!\n");
326     snprintf(userfile, sizeof(userfile)-1, "%s.txt", user);
327     snprintf(search, sizeof(userfile)-1, "stat %s`ls %s | grep %s`", USERPATH, USERPATH,
        userfile);
328     ret = system(search);
```

In line 29, LISTCOMMAND is defined as a global constant:

```
28  #define ARTICLEPATH "./articles/"
29  #define LISTCOMMAND "ls ./articles/ > list.txt"
```

Analysis: The **system()** function executes system commands via **/bin/sh** (the directory path where the shell code is stored), which is dangerous when executing user-generated commands inputted by the user if it is malicious. **Line 198** may not be a potential vulnerability because it uses **LISTCOMMAND**, which is a defined global constant **(in line 29)**, meaning no user is able to change it unless they gain access to the source code, making this case of **system()** not exploitable. **Lines 214 and 328** are potential vulnerabilities because they both use the **system()** function to run an input given by the user. Since **system()** executes commands from a shell via **/bin/sh**, the user has the potential to manipulate the environment variables within the system, which allows the potential for the user to exploit the system call executed from the shell to invoke other system commands that the user may not have permission to do so.

Fix: The most efficient way to fix vulnerabilities related to **system()** calls is to create an entirely new function which does the same job as what the **system()** call in the code is doing. However, it was recommended not to do it as it could take a significant amount of time to create such functions.

8 - memcpy():

MEDIUM: Potentially Unsafe Code - memcpy

Line: 371 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions and other memory mis-management situations.

```
memcpy((char *)&size, ptr1, 4);
```

MEDIUM: Potentially Unsafe Code - memcpy

Line: 417 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions and other memory mis-management situations.

```
memcpy((char *)&segmentcount, ptr1, 4);
```

MEDIUM: Potentially Unsafe Code - memcpy

Line: 436 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions and other memory mis-management situations.

```
memcpy((char *)&segnext, ptr1, 4);
```

MEDIUM: Potentially Unsafe Code - memcpy

Line: 439 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions and other memory mis-management situations.

```
memcpy((char *)&argsize, ptr1, 4);
```

MEDIUM: Potentially Unsafe Code - memcpy

Line: 441 - C:\Users\Jimmy's PC\Downloads\CYSE 411 Project\cyse411project1.c

Function appears in Microsoft's banned function list. Can facilitate buffer overflow conditions and other memory mis-management situations.

```
memcpy(ptr2, ptr1, argsize);
```

```
362     char *ptr1;
363     char *found = NULL;
364     char type = 0;
365     size_t size;
366 
367     ptr1 = argbuf;
368 
369     while (1)
370     {
371         memcpy((char *)&size, ptr1, 4);
```

```
402     /* read in data */
403     memset(buffer, 0, sizeof(buffer));
404     len = readSock(sock, buffer, sizeof(buffer));
405     logData(logfile, "handling connection");
406 
407     if (len == -1)
408     {
409         return;
410     }
411 
412     /* parse protocol */
413     ptr1 = buffer;
414     ptr2 = argbuf;
415 
416     /* get count of segments */
417     memcpy((char *)&segmentcount, ptr1, 4);
```

```
431     memset(argbuf, 0, sizeof(argbuf));
432
433     for (segloop = 0; segloop < segmentcount; ++segloop)
434     {
435         logData(logfile, "adding segment %i", segloop+1);
436         memcpy((char *)&segnext, ptrl, 4);
437         logData(logfile, "next segment offset %i", segnext);
438         ptrl += 4;
439         memcpy((char *)&argsize, ptrl, 4);
440         logData(logfile, "argsize: %i", argsize);
441         memcpy(ptr2, ptrl, argsize);
442         ptr2 += argsize;
443         ptrl += segnext;
444     }
```

Analysis: Similar to **strcpy()**, **memcpy()** does not check for boundaries of the memory location when copying from one memory location to the other, thus if the source memory location is larger than the destination memory location, it can cause an overflow that can override other memory locations with malicious data. In this context, the tool detected 5 different usages of **memcpy()**, and all 5 of them (**Lines 371, 417,436, 439, and 441**) are not potential vulnerabilities because the size of an address in a 32-bit system is 4 bytes, and the allocated size with the **memcpy()** usage is 4, so no buffer overflow is happening.

Conclusion

From the results between Cppcheck and Visualcodegrepper, one can observe that the two static analysis tools only share one similar potential vulnerability of the code (albeit after the in-depth analysis), thus both programs provide drastically different sets of potential vulnerabilities. The cause of this outcome is due to how each static analysis tool measures and identifies potential vulnerabilities: Cppcheck identifies vulnerabilities through resource leaks and out-of-bound buffers while Visualcodegrepper identifies vulnerabilities through the functions the code uses (strcpy(), system(), etc.). However, the two static analysis tools provide a good, general overview of how a code can undergo vulnerability scanning in many different ways. In addition, the in-depth analysis of each potential vulnerability suggests that a vulnerability can be exploited if the variable is used as an input at some point in the code, thus a potential vulnerability can exist if such input can be altered by the user to their advantage. Furthermore, if a user cannot interact with the vulnerable code via user input nor make any changes to the existing inputs, the vulnerability is determined to be not potential in which the code cannot be exploited.