Program Heuristics

Scan files in a directory and all sub directories

Scan for webshell signature matches

Scan for dangerous function matches

Scan for codes placed on the same line:

Scan for php:

- **Scanning for user input via $_GET, $_POST etc.**
- **Scanning for variables used in dangerous functions**
- **Scanning for user input assigned to variables**
- **Scanning for variables of user input assigned to other variables**
- **Scan for codes encoded using base64**
- **Properly identify user-defined functions, and identify dangerous functions used in them**

Scan for asp:

- **Scan for user input**
- **Scan for user input assigned to variables**
- **Scan for variables of user input assigned to other variables**
- **Scan for use of user input in dangerous functions**
- **Scan for user-defined functions, and identify dangerous functions used in them**

Scan for jsp:

- **Scan for user input**
- **Scan for user input assigned to variables**
- **Scan for variables of user input assigned to other variables**
- **Scan for use of user input in dangerous functions**

Preparation stage

1.

```
#ifdef _WIN32
            this->logfolder= this->startdir + "\\logs";
            this->shellFolder = this->startdir + "\\signatureLogs";
      #else
            this->logfolder= this->startdir + "/logs";
            this->shellFolder = this->startdir + "/signatureLogs";
      #endif

#ifdef _WIN32
            _mkdir(logfolder.c_str());
            _mkdir(shellFolder.c_str());
      #else
            mkdir(logfolder.c_str(), 0777);
            mkdir(shellFolder.c_str(), 0777);
      #endif
```

This is used to create the starting folders to contain the log files of the scan results. Depending on the OS used, either Linux or Windows, the folders are created differently to ensure portability.

2.

```
dangerInput.open("dangerFunction.txt");
      if(dangerInput) {
            while (getline(dangerInput, line)) {
                  if (line == "7")
                        type = 2;
                  if (line== "11")
                        type = 3;
                  switch(type) {
                        case 1: dangerFunction.push_back(line);
                                break;
                        case 2: aspFunction.push_back(line);
                                break;
                        case 3: jspFunction.push_back(line);
                        default:;
                  }
            }
      }
      else {
            cout << "ERROR OPENING DANGER FUNCTION FILE. EXITING" <<
endl;
            exit(0);
      }
      dangerInput.close();

      //retrieving web shell signatures from text file
      signatureInput.open("md5-signatures-w-names.txt");
      if(signatureInput) {
            while (getline(signatureInput, line)) {
                  signatures.push_back(line);
            }
      }
      else {
            cout << "ERROR OPENING SIGNATURE FILE. EXITING" << endl;
            exit(0);
      }
      signatureInput.close();
```

These codes are used to retrieve the dangerous function list and web shell signature list. They will be used for the scan of web shells. The dangerous functions of the different web languages are stored separately to increase efficiency and the neatness of the code.

<u>Scan files in a directory and all sub directories</u>

The scanning of directories and all sub directories are handled by the method **scanDirectory** of the **FileReader** class.

Key codes that handle the scanning directory functionality

1.

```
while ((dirp = readdir(dp)) != NULL) {
```

The above code keeps the program scanning until no more files or directories have been detected.

2.

```
if (!strcmp(dirp->d_name, "..") || !strcmp(dirp->d_name, ".")) {
    continue;
}
```

The above code prevents the scanner from scanning folders ".." and "." . Each directory contains these 2 default folders, even the folders ".." and "." themselves. Without the code above the scanner would enter an infinite loop, scanning further and further down in the those 2 folders.

3.

```
#ifdef _WIN32
    filepath = dir + "\\" + dirp->d_name;
#else
    filepath = dir + "/" + dirp->d_name;
#endif
```

The above code determines the full path of the file. Depending on whether Linux or Windows is used, the file path would be different. The program computes the file path differently depending on the OS used.

4.

```
if (stat(filepath.c_str(), &filestat)) continue;
```

This code is used to check for a valid file path. If the file path is invalid in any way, the program will skip that file and continue its operation.

5.

```
if (S_ISDIR(filestat.st_mode)) {
    if ((filepath != logfolder) && (filepath!= shellFolder)) {
        cout << "directory detected" << endl;
        counter++;
        scanDirectory(filepath, option, counter);
    }
}
```

The above code determines if the file that the program has detected is a directory. If it is, the program displays an appropriate message, and calls a recursive scanDirectory method on the detected directory. Counter is used to determine the level of the directory. For every sub-directory it goes in, the counter is increased. This helps to differentiate files with same names but in different directories.

6.1

```
else if (S_ISREG(filestat.st_mode)) {
    i = filepath.size()-1;
    while (((filepath[i] != '/') && (filepath[i] != '\\')) &&  (i>=0)) {
        i--;
    }
    filename = filepath.substr(i+1, filepath.size()-1);
    ss << counter;
    #ifdef _WIN32
        logfile = logfolder + "\\" + filename + ss.str() + ".txt";
    #else
        logfile = logfolder + "/" + filename + ss.str() + ".txt";
    #endif
        remove(logfile.c_str());
        ss.str("");
```

If the file detected is a regular file, the code above catches it and handles it appropriately. The above code first retrieves the filename from the full path. It then generates the name of the log file depending on the filename, the level of the directory, and the OS used. It also removes any previous log file with the same name to ensure that the results are up-to-date.

6.2

```
fs.setLogfile(logfile);
fs.setShellLog(shellLog);
fs.setCurrentFile(filepath);
```

The above sets the attributes need for the fileScanner class to scan the file. It sets the log file to store the results, the shellLog to store results for the webshell signature scan, and it sets the file path of the file to be scanned.

6.3

```
if (option == "2")
    fs.webshellScan(signatures);
else if (option == "1") {
    if ((offset = filepath.find(".php", 0)) != string::npos) {
        format = ".php";
        fs.scanCurrentFile(dangerFunction, format);
    }
    else if ((offset = filepath.find(".asp", 0)) != string::npos) {
        format = ".asp";
        fs.scanCurrentFile(aspFunction, format);
    }
    else if ((offset = filepath.find(".aspx", 0)) != string::npos) {
        format = ".aspx";
        fs.scanCurrentFile(aspFunction, format);
    }
    else if ((offset = filepath.find(".jsp", 0)) != string::npos) {
        format = ".jsp";
        fs.scanCurrentFile(jspFunction, format);
    }
    else {
        format = ".php";
        cout << "Default scan: php" << endl;
        fs.scanCurrentFile(dangerFunction, format);
    }
}
```

The above code handles the options 1 and 2 of the users selection. If option 1 is chosen, the program scans the file according to the type of file format it is. The program is able to handle .php, .asp, .aspx, and .jsp file formats. If the file format does not match any of the above, it will scan it as a .php file as default, as php is more widely used. If option 2 is chosen, the program carries out a web shell signature scan, matching the signature to a compiled signature list of web shells that we have compiled.

6.4

```
else {
    fs.webshellScan(signatures);
    if ((offset = filepath.find(".php", 0)) != string::npos) {
        format = ".php";
        fs.scanCurrentFile(dangerFunction, format);
    }
    else if ((offset = filepath.find(".asp", 0)) != string::npos) {
        format = ".asp";
        fs.scanCurrentFile(aspFunction, format);
    }
    else if ((offset = filepath.find(".aspx", 0)) != string::npos) {
         format = ".aspx";
         fs.scanCurrentFile(aspFunction, format);
    }
    else if ((offset = filepath.find(".jsp", 0)) != string::npos) {
        format = ".jsp";
        fs.scanCurrentFile(jspFunction, format);
    }
    else {
        format = ".php";
        cout << "Default scan: php" << endl;
        fs.scanCurrentFile(dangerFunction, format);
    }
}
```

The above code is used when both options 1 and 2 are selected. The program will then carry out both the web shell scans and the dangerous functions scan on the same file.

Scan for webshell signature matches

The scanning for webshell signatures is handled by the **fileScanner** class, using the **webShellScan** method.

Key codes of webShellScan method

1.

```
char *a = new char[cfilename.size() + 1];
a[cfilename.size()] = 0;
memcpy(a, cfilename.c_str(), cfilename.size());
fileHash = md5.digestFile(a);
```

The code above first creates a char* variable to hold the bytes of the file name. After copying the bytes, it is then hashed using md5 to obtain the hashed signature of the file being scanned.

2.

```
for(unsigned int i = 0; i < signatures.size(); i++) {
    if((signatures[i].c_str()) == fileHash) {
```

The code above attempts to find a match for the signature of the file hash with a list of web shell signatures that we have obtained.

Scan for dangerous function matches

The scanning for dangerous functions is handled by the **fileScanner** class. The methods used are the **scanCurrentFile** method, and the different scan methods for the respective formats, e.g **scanPHP** method.

Key codes for the scanning of dangerous functions functionality

1.

```
void FileScanner::scanCurrentFile(vector<string> signature, string
format){
    if (format == ".php")
        scanPHP(signature);
    else if ((format == ".aspx") || (format == ".asp"))
        scanASP(signature);
    else if (format == ".jsp")
        scanJSP(signature);
    else;
}
```

The above code shows the **scanCurrentFile** method. The method calls the appropriate scan methods based on the file format.

2.

```
fileInput.open(cfilename.c_str());
```

```
    if(fileInput.is_open()) {
        while(getline(fileInput, line)) {
            content.push_back(line);
        }
        fileInput.close();
    }
    else cout << "Unable to open file.";
```

The code above is used to obtain the file contents for easy access. It opens the file based on the file name. The contents are then retrieved and stored in a vector.

3.

```
while(lineno < content.size()) {
```

This is the first loop to scan through the file contents. This causes the program to scan through each line of file, until the end of the file contents.

4.

```
while(i<signature.size()) {
```

This is the second loop which loops through the danger functions list. As the program scans through each line, it checks the line for the use of any of the dangerous functions listed.

5.

```
if (signature[i] == "1") {
    category = 1;
     score = 10;
}
else if (signature[i] == "2") {
    category = 2;
    score = 9;
}
else if (signature[i] == "3") {
    category = 3;
    score = 5;
}
else if (signature[i] == "4") {
    category = 4;
    score = 1;
}
else if (signature[i] == "5") {
    category = 5;
    score = 4;
}
else if (signature[i] == "6") {
    category = 6;
    score = 6;
}
else if (signature[i] == "7") {
    category = 7;
}
```

This code logic separates the functions into different categories. This is done as functions belonging to different categories may have varying levels of dangers, and may warrant different scores assigned to them.

6.

```
while ((offset = content[lineno].find(signature[i], wall)) !=
string::npos) {
    startPos = offset;
    offset = offset + signature[i].size();
    wall = offset+1;
```

This code is the loop that searches for a match of a dangerous function. As the loop goes through the dangerous function list, it checks the current line of the file content for the use of any dangerous function. This loop is also used to **scan codes placed on the same line**. This is achieved by using the “wall” variable.

PHP scanning heuristics

The following are the key codes used to scan for dangerous functions for PHP files.

Scanning for user input via $_GET, $_POST etc.

```
if ((content[j][offset1] == '$')) {
    offset1++;
    variableDone = true;
    if (content[j][offset1] == '_') {
        r.addDangerFunctions(signature[i], score, lineno);
        scanned = true;
```

First the program checks for the presence of the '$' character. This suggests the use of a variable or user input. Then the program checks if the next character is a '_' character. Once that has been determined, this means that a user input is currently being used. Some examples of user input are $_GET, $_POST and $_REQUEST. As this code is executed on the event of a dangerous function being detected, this means that user input is used in a potentially dangerous function. The program would then add it to the results list and assign it an appropriate score.

Scanning for variables used in dangerous functions

```
variable = "$";
while ((content[lineno][offset]>=65 && content[lineno][offset]<=90)
||(content[lineno][offset]>=97 && content[lineno][offset]<=122) ||
(content[lineno][offset]>=48 && content[lineno][offset]<=57) ||
(content[lineno][offset] == '_')) {

      variable.append(1, content[lineno][offset]);
      offset++;

}
offset--;
```

In the event that a '_' was not detected, this would mean that a variable was used in the dangerous function. The code above is used to obtain the variable name, which will be used for further scanning to see if the variable was assigned a value from user input.

Scanning for user input assigned to variables and scanning for variables of user input assigned to other variables

1.

```
while (j<content.size() && !scanned) {
while (((offset1 = content[j].find(variable,wall1)) !=
string::npos)) {
```

These 2 loops are used to scan for the contents of the variable used in the dangerous function. They will keep scanning until an assignment of a value to the variable is detected.

2.

```
    else if (content[j][offset1] == '=') {
```

The main thing that the program tries to detect is the '=' operation. This would mean that the variable is now being equated to a value.

3.

```
if ((content[j][offset1] == '$')) {
    offset1++;
    variableDone = true;

      //if '_' is detected, userinput confirmed
    if (content[j][offset1] == '_') {
        r.addDangerFunctions(signature[i], score, lineno);
        scanned = true;
    }
    //else, retrieve the variable assigned to the current variable
    else {
        wall2 = offset1-1;
        variable1 = "$";
        while ((content[j][offset1]>=65 && content[j][offset1]<=90)
||(content[j][offset1]>=97 && content[j][offset1]<=122)
||(content[lineno][offset1]>=48 && content[lineno][offset1]<=57)
||(content[j][offset1] == '_')) {
            variable1.append(1, content[j][offset1]);
            offset1++;
        }
        //scan the new variable to check if it is user input
            variable = variable1;
            previous = j;
            j=0;
            wall1=0;
        }
        break;
    }
```

The codes shown above are the same as the first 2 code snippets. If a "$_" is detected, this would mean that the variable detected has a user input assigned to it. If only a '$' was detected, the variable would be kept and the search would be continued using that new variable.

Scan for codes encoded using base64

```
//if quotes is detected, means that variable
// is assigned a hardcoded value, not user input

    else if ((content[j][offset1] == '\"')|| (content[j][offset1] ==
'\'') ) {

    //however if function detected is base64_decode
    //retrieve the contents within the quotes
        if (signature[i] == "base64_decode") {
            quote = content[j][offset1];
            variable1 = "";
            if (offset1 < content[j].size())
                offset1++;
            else {
                j++;
                offset1=0;
            }
            while (content[j][offset1] != quote) {
                variable1 = variable1+content[j][offset1];
                if (offset1 < content[j].size())
                    offset1++;
                else {
                    j++;
                    offset1=0;
                }
            }
            //decode the contents & write it to a temporary file
            //and scan it using the scanCurrentFile function

            decoded = base64_decode(variable1);
            cout << "base64_decode operation in file \"" << cfile <<
"\"" << endl;
            cout <<  "Line number " << lineno << endl;
            cout << endl;
            logger.open(logfile.c_str(), ios::out | ios::app);
            logger << "base64_decode operation in file \"" << cfile
<< "\"" << endl;
            logger <<  "Line number " << lineno << endl;
            logger << endl;
            logger.close();
            afile.open(decodedFilename.c_str(), ios::out);
            afile << "base64_decode operation in file \"" << cfile
<< "\"" << endl;
            afile <<  "Line number " << lineno << endl;
            afile << endl;
            afile << decoded;
            afile.close();
            cfilename = decodedFilename;
            scanCurrentFile(signature, ".php");
            //remove temporary file after scan
            remove(decodedFilename.c_str());
            cfilename = cfile;
    }
    scanned = true;
    variableDone = true;
}
```

The above code is used to check for quotes. For other dangerous functions or for when checking for assignment of variables, if quotes are detected, it usually means that the argument provided is hardcoded by the owner. However in the case of a base64_encode operation, it is necessary to check the contents of the argument as it could contain dangerous functions within. The above code checks if the detected function is base64_encode. If it is, obtain the contents within the quotes and decode it using a base64 decode method. The contents are then written to a temporary file to be scanned. The file is then removed after the scan is done.

Properly identify user-defined functions, and identify dangerous functions used in them

```
else if ((content[j][offset1] == ',')||(content[j][offset1] == ')'))
{
    //move counter till '(' is detected
    while (!(content[j][offset1] == '(')) {
        offset1--;
        if (offset1 < 0) {
            j--;
            offset1 = content[j].size() - 1;
        }
        //if "=>" is found, means it’s not a function but
        //a foreach operation

        if (content[j][offset1] == '>') {
            offset1--;
            if (content[j][offset1] == '=') {
                scanned = true;
                variableDone = true;
                break;
            }
        }
}
    //if '(' reached, obtain function name
    if (!variableDone) {
        functionName = "";
        offset1--;
        while (content[j][offset1] != ' ') {
            functionName = content[j][offset1] + functionName;
            offset1--;
            //if end of line is reached before a space, means
            //it’s not a user-defined function header
            if (offset1 < 0) {
                    break;
               }
               function = true;
           }
           //continue to check if there us the "function" keyword
           if (function) {
               variable1 = "";
               offset1--;
               for (int h=0; h<8; h++) {
                   variable1 = content[j][offset1] + variable1;
                   offset1--;
                   if (offset1 < 0) {
                       break;
                   }
               }
               found = false;
               //if "function" keyword is found, check if signature
               //list already has it
               if (variable1 == "function") {
                   for (int h=signature.size()-1; h>=0; h--) {
                       if (signature[h] == functionName){
                       found = true;
                       break;
                   }
               }
               //if not, add it to the list of dangerous functions
               //and results, as well as giving a warning
               if (!found) {
                   signature.push_back(functionName);
                   phpComments.push_back(functionName);
                   phpComments.push_back("This is a user-defined
function that contains the use of a dangerous function.\n" "Please
take care in using it.");
                   r.addDangerFunctions(functionName, 2, lineno);
                   scanned = true;
                }
            }
        }
    }
    //variableDone = true;
}
```

The above code checks for functions. It is used at the same time when scanning for the '=' operation. If a ',' or ')' is found instead, this means that the variable is used within a function. The program will then try to find the "function" keyword. If it is detected, it would mean that the function is a user-defined function. In that case, the program will flag the user-defined function as a dangerous function. At the same time, it is added to the results list of dangerous functions with a score of 2 as a warning to the user.

ASP Scanning Heuristics

The following are the key codes used to scan for dangerous functions for asp files.

Scan for user input

1.

```
//obtain user input from a form
if (signature[i] == "<input") {
    found = false;
    variable = "";
    while ( lineno < content.size()) {
    //determine the type of form, may ignore some types
    if((offset1=content[lineno].find("type=",wall))!= string::npos){
        offset1 = offset1+5;
        if (content[lineno][offset1] == '\"') {
            stack.push_back('\"');
            while (stack.size() != 0) {
                offset1++;
                if (content[lineno][offset1] == '\"')
                    stack.pop_back();
                else
                    variable.append(1, content[lineno][offset1]);
        }
    }
    else {
        while ((content[lineno][offset1]>=65 &&
        content[lineno][offset1]<=90) ||
        (content[lineno][offset1]>=97 &&
        content[lineno][offset1]<=122) ||
        (content[lineno][offset1] == '_') &&
        (offset < content[lineno].size())) {

            variable.append(1, content[lineno][offset1]);
            offset1++;
        }
    }
    //types to ignore
    if (variable == "submit") {
        found = false;
        break;
    }
    else if (variable == "reset") {
        found = false;
        break;
    }
    else {
    //find the assignment of the variable
        variable = "";
        while ( lineno < content.size()) {
            if ((offset1 = content[lineno].find("name=", wall)) !=
            string::npos) {
                offset1 = offset1+5;
                wall = offset+1;
                found = true;
                break;
            }
            else if ((offset1 = content[lineno].find("id=",
            wall)) != string::npos) {
                offset1 = offset1+3;
                wall = offset+1;
                found = true;
                break;
            }
            else {
                lineno++;
                wall=0;
            }
        }
        if (found)
            break;
        }
    }
    else {
        lineno++;
    }
}
//if found, obtain the variable name
if (found) {
//obtain variable name if enclosed in quotes
    if (content[lineno][offset1] == '\"') {
        stack.push_back('\"');
        while (stack.size() != 0) {
        offset1++;
        if (content[lineno][offset1] == '\"')
            stack.pop_back();
        else
            variable.append(1, content[lineno][offset1]);
    }
}
//obtain variable name normally
else {
    while ((content[lineno][offset1]>=65 &&
    content[lineno][offset1]<=90) ||
    (content[lineno][offset1]>=97 &&
    content[lineno][offset1]<=122) ||
    (content[lineno][offset1] == '_') &&
    (offset < content[lineno].size())) {
        variable.append(1, content[lineno][offset1]);
        offset1++;
    }
}
//store the variable and its lineno
variables.push_back(variable);
vline.push_back(lineno);
userInput.push_back(variable);
```

The above code handles user input obtained from a web form. Upon detection of the <input command, it then determines the form type. If it is of a submit type or reset type, skip them. Otherwise, proceed to obtain the variable name and add it to a variables list, if haven't already.

2.

```
else if ((signature[i] == "request.cookies") ||
(signature[i] == "request.form") ||
(signature[i] == "request.querystring")) {
    if (signature[i] == "request.cookies")
        variable = "request.cookies";
        else if (signature[i] == "request.form")
        variable = "request.form";
     else if (signature[i] == "request.querystring")
         "request.querystring";
         //obtaining the full request statement
         while ((content[lineno][offset] != ' ') &&
         (offset < content[lineno].size())) {
             if (content[lineno][offset] == '(') {
                 stack.push_back(content[lineno][offset]);
                 variable.append(1, content[lineno][offset]);
                 while (stack.size() != 0) {
                     offset++;
                     if (content[lineno][offset] == ')') {
                         stack.pop_back();
                         variable.append
                         (1, content[lineno][offset]);
                     }
                     else
                         variable.append
                         (1, content[lineno][offset]);
                     }
            }
            else
                break;
        }
        found = false;
        assign = true;
        //add to user input list if haven't already
            if (variablesUsed.size() !=0) {
                for (int j=0; j<variablesUsed.size(); j++) {
                    if (variablesUsed[j] == variable) {
                         found = true;
                         break;
                    }
                }
                if (!found) {
                    if (function) {
                        functionVariable.push_back(variable);
                        variablesUsed.push_back(variable);
                        fline.push_back(lineno);
                    }
                    else {
                        variables.push_back(variable);
                        vline.push_back(lineno);
```

```
              userInput.push_back(variable);
              variablesUsed.push_back(variable);
          }
      }
  }
  else {
      if (function) {
          functionVariable.push_back(variable);
          variablesUsed.push_back(variable);
          fline.push_back(lineno);
      }
      else {
          variables.push_back(variable);
          vline.push_back(lineno);
          userInput.push_back(variable);
          variablesUsed.push_back(variable);
      }
 }
 //if user input detected is new, try to obtain the
 //variable it is assigned to
 if (!found) {
     offset = startPos;
     //checking if user input is assigned to a variable
     while (content[lineno][offset] != '=') {
         offset--;
         if ((offset<0) ||
         (content[lineno][offset] == ',') ||
         (content[lineno][offset] == '(')) {
             assign = false;
             break;
         }
}
//if variable was assigned to a variable, obtain that
//variable
if (assign) {
    offset--;
        if (content[lineno][offset] == ' ')
            offset--;
            variable = "";
            //2nd check to see if user input was assigned
            //to a variable
            while ((content[lineno][offset] != ' ') &&
           (offset >=0)) {
               variable = content[lineno][offset] +
               variable;
               offset--;
               if (content[lineno][offset] == '.') {
                   assign = false;
                   break;
               }
           }
           //add obtained variable to the list, if haven't
           //already
           if (assign) {
               found = false;
```

```
                            if (variablesUsed.size() !=0) {
                                for (int j=0; j<variablesUsed.size();
                                j++) {
                                    if (variablesUsed[j] == variable) {
                                        found = true;
                                        break;
                                    }
                                }

if (!found) {
    if (function) {
        functionVariable.push_back(variable);
        variablesUsed.push_back(variable);
        fline.push_back(lineno);
    }
    else {
        variables.push_back(variable);
        vline.push_back(lineno);
        userInput.push_back(variable);
        variablesUsed.push_back(variable);
    }
}
else {
    if (function) {
        functionVariable.push_back(variable);
        variablesUsed.push_back(variable);
        fline.push_back(lineno);
    }
else {
    variables.push_back(variable);
    vline.push_back(lineno);
    userInput.push_back(variable);
    variablesUsed.push_back(variable);
}
```

This part of the code is obtaining user input through the request operations. Upon detecting a request operation, it first obtains the full statement of the request, and adds it to the variable list if haven’t already. It then checks if the user input was assigned to any variable. If it was, it adds it to the variable list as well.

3.

```
else if (signature[i] == ".text") {
    variable = ".text";
    while ((content[lineno][offset] != ' ') && (offset >=0)) {
        variable = content[lineno][offset] + variable;
        offset--;
    }
    //add variable obtained into user input list, if haven't already
    found = false;
    assign = true;
    if (variablesUsed.size() !=0) {
        for (int j=0; j<variablesUsed.size(); j++) {
            if (variablesUsed[j] == variable) {
                found = true;
                break;
            }
        }
        if (!found) {
            if (function) {
                functionVariable.push_back(variable);
                variablesUsed.push_back(variable);
                fline.push_back(lineno);
            }
            else {
                variables.push_back(variable);
                vline.push_back(lineno);
                userInput.push_back(variable);
                variablesUsed.push_back(variable);
            }
         }
    }
    else {
        if (function) {
            functionVariable.push_back(variable);
            variablesUsed.push_back(variable);
            fline.push_back(lineno);
        }
        else {
            variables.push_back(variable);
            vline.push_back(lineno);
            userInput.push_back(variable);
            variablesUsed.push_back(variable);
        }
    }
    // if user input is new, check if it is assigned to a variable
    if (!found) {
        while (content[lineno][offset] != '=') {
            offset--;
            if ((offset<0) || (content[lineno][offset] == ',') ||
            (content[lineno][offset] == '(')) {
                assign = false;
                break;
            }
        }
        //if it is, obtain the variable
            if (assign) {
                offset--;
                if (content[lineno][offset] == ' ')
                    offset--;
                    variable = "";
                    while ((content[lineno][offset] != ' ') &&
                    (offset >=0)) {
                        variable = content[lineno][offset] +
                        variable;
                        offset--;
                        //2nd check to see if user input is assigned
                        //to a variable
                        if (content[lineno][offset] == '.') {
                            assign = false;
                            break;
                        }
                    }
                    //if variable is obtained, add it to the user
                    //input list, if haven't already
                    if (assign) {
                        found = false;
                        if (variablesUsed.size() !=0) {
                            for (int j=0; j<variablesUsed.size();
                            j++) {
                                 if (variablesUsed[j] == variable) {
                                     found = true;
                                     break;
                                 }
                             }
                             if (!found) {
                                 if (function) {
                                 functionVariable.push_back
                                 (variable);
                                 variablesUsed.push_back(variable);
                                 fline.push_back(lineno);
                             }
                             else {
                                 variables.push_back(variable);
                                 vline.push_back(lineno);
                                 userInput.push_back(variable);
                                 variablesUsed.push_back(variable);
                             }
                         }
                     }
                     else {
                         if (function) {
                             functionVariable.push_back(variable);
                             variablesUsed.push_back(variable);
                             fline.push_back(lineno);
                         }
                         else {
                             variables.push_back(variable);
                             vline.push_back(lineno);
                             userInput.push_back(variable);
                             variablesUsed.push_back(variable);
                         }
                     }
                }
            }
        }
    }
```

This part of the code scans for user input obtained from a text box. It obtains variable name of the text box and adds it to the variable list. It then checks if it was assigned to any variable, and add it to the list as well if it was.

Scan for variables of user input assigned to other variables

```
while ((offset = content[lineno].find(variablesUsed[i], wall)) !=
string::npos) {
    startPos = offset;
    wall = offset+1;
    assign = true;
    //1st check to determine if the user input was assigned to
    //another variable
    while (content[lineno][offset] != '=') {
        offset--;
        if ((offset<0) || (content[lineno][offset] == ',') ||
        (content[lineno][offset] == '(')) {
            assign = false;
            break;
        }
    }
    if (assign) {
        offset--;
        if (content[lineno][offset] == ' ')
            offset--;
            variable = "";
            //obtained variable it was assigned to
            while ((content[lineno][offset] != ' ')&& (offset >=0)){
                variable = content[lineno][offset] + variable;
                offset--;
                //2nd check to determine if the user input was
                //assigned to another variable
                if (content[lineno][offset] == '.') {
                    assign = false;
                    break;
                }
            }
```

The code scans for the use of user input that were previously scanned and stored in a variables list. Once detected, it then checks if they were assigned to other variables. If so, it obtains the variable and stores it in the variable list as well.

Scan for use of user input in dangerous functions

```
for (int j=0; j<variablesUsed.size(); j++) {
    if ((offset = content[lineno].find(variablesUsed[j], 0)) !=
    string::npos) {
```

These are the main loops to check if user input was used in a dangerous function. If a dangerous function is detected, these 2 loops would be executed to check if any user input was used in them.

Scan for user-defined functions, and identify dangerous functions used in them

```
if ((signature[i] == "end function")||(signature[i] == "end sub")) {
    function = false;
    functionVariable.erase(functionVariable.begin(),
    functionVariable.end());
    fline.erase(fline.begin(), fline.end());
}
else if ((signature[i] == "function") || (signature[i] == "sub")) {
    if (content[lineno][offset] != ' ') {
        continue;
    }
    else if (content[lineno].find("{", 0) != string::npos) {
        continue;
    }
    else {
        offset++;
        functionName = "";
        while (content[lineno][offset] != '(') {
            functionName = functionName + content[lineno][offset];
            offset++;
        }
        variable = "";
        while (content[lineno][offset] != ')') {
            variable = variable + content[lineno][offset];
            offset++;
            if ((content[lineno][offset] == ',') ||
            (content[lineno][offset] == ')')) {
                functionVariable.push_back(variable);
                fline.push_back(lineno);
            }
        }
        function = true;
    }
}
```

The code above first detects the start of a user-defined function. Once detected, it proceeds to retrieve the argument names used in the function. At this point, a function tag is now marked true, thus until the end of the function is detected, all dangerous function detections will be considered under the user function.

```
if (function) {
    if (signature[signature.size()-1] == functionName) {
        aspComments[aspComments.size()-1] =
        aspComments[aspComments.size()-1] + "\nIt also contains
        the use of dangerous file system functions using the
        arguments of the function.";
    }
    else {
        signature.push_back(functionName);
        functionName = "User-defined Function \"" +functionName +
        "\"";
        aspComments.push_back(functionName);
        aspComments.push_back("This is a user-defined function. It
        contains the use of dangerous file system functions using
        the arguments of the function.");
    }
}
```

This code is executed when the function tag is true. It handles the detection of a dangerous function differently. When detected, it adds the user function to the results list instead of the dangerous function detected.

<u>JSP Scanning Heuristics</u>

The following are the key codes used to scan for dangerous functions for jsp files.

Scan for user input

1.

```
if (category == 11) {
    assign = false;
    k = lineno;
    while ((content[k][offset] != '=') && (content[k][offset] !=
    '(')) {
        offset--;
        if (offset < 0 ) {
            k--;
            offset=content[k].size()-1;
        }
    }
    //checking for assignment
    if (content[k][offset] == '=')
        assign = true;
    else
        assign = false;
    //if it was assigned to a variable, obtain the variable
    if (assign) {
        if (content[k][offset] == ' ') {
            offset--;
        if (offset < 0 ) {
            k--;
            offset=content[k].size()-1;
        }
    }
    variable = "";
    found = false;
    while (content[k][offset] != ' ') {
        variable = content[k][offset] + variable;
        offset--;
        if (offset < 0 ) {
            k--;
            offset=content[k].size()-1;
        }
    }
    //add the obtained variable to the user input list, if haven't
    //already
    if (userInput.size() !=0) {
        for (int j=0; j<userInput.size(); j++) {
            if (userInput[j] == variable) {
                found = true;
                break;
            }
        }
        if (!found) {
            userInput.push_back(variable);
        }
    }
    else {
        userInput.push_back(variable);
    }
}
```

This part scans for reader objects like “Scanner” or “BufferedInputReader”. Category 11 holds the object names of the scanner or bufferedinputreader objects that were created and detected by the program. If the use of such an object is detected, the program obtains the variable the user input was assigned to and adds it to the user input list.

2.

```
else if (category == 12) {
    k = lineno;
    //obtain variable
    while (content[k][offset] != '=') {
        offset--;
        if (offset < 0 ) {
        k--;
        offset=content[k].size()-1;
    }
}
offset--;
if (offset < 0 ) {
    k--;
    offset=content[k].size()-1;
}
if (content[lineno][offset] == ' ') {
    offset--;
    if (offset < 0 ) {
        k--;
        offset=content[k].size()-1;
    }
}
variable = "";
while (content[k][offset] != ' ') {
    variable = content[k][offset] + variable;
    offset--;
    if (offset < 0 ) {
        k--;
        offset=content[k].size()-1;
    }
}
//for getParameter method, the variable holds the user input
//but for scanner and bufferedreader, the variable is a scanner
//object that will be used to obtain the user input
if (signature[i] == "getParameter") {
    userInput.push_back(variable);
}
else {
    found = false;
    if (variables.size() !=0) {
    for (int j=0; j<variables.size(); j++) {
        if (variables[j] == variable) {
            found = true;
            break;
        }
    }
    if (!found) {
        variables.push_back(variable);
        signature.insert(signature.begin()+1, variable);
        if (signature[i] == "new Scanner")
            variable = variable + ".next";
        else if (signature[i] == "new BufferedReader")
            variable = variable + ".read";
        userInput.push_back(variable);
        }
    }
    else {
        variables.push_back(variable);
        signature.insert(signature.begin()+1, variable);
        if (signature[i] == "new Scanner")
            variable = variable + ".next";
        else if (signature[i] == "new BufferedReader")
            variable = variable + ".read";
        userInput.push_back(variable);
        }
    }
}
```

This part checks for the creation of the scanner objects, as well as the use of the getParameter method. It obtains the object name created and adds it under category 11 so as to be used for scanning for user input. If getParameter was detected, the variable is directly added to the user input list instead.

Scan for user input assigned to variables and Scan for variables of user input assigned to other variables

Scan for use of user input in dangerous functions

For scanning of user input assigned to variables and the actual scanning of the use of user input in dangerous functions, it is the same as in the ASP format scanning, only difference being some of the syntax differences between asp and jsp.