1 CSC 495 — Assignment 3 — Due Tuesday, April 14

1. This problem deals with the clang output at http://cs.uncg.edu/faculty/srtate/495.s15/clang.html and in particular the report for `clamav-0.98.5`. Bring up the report for the first item in the API Bug Group, referring to line 711 of `libltdl/ltdl.c` — the claim is that `filename` could be NULL, leading to a null pointer dereference when `strcpy` is called. However, this actually can never happen. For this problem, write a clear argument showing why there cannot be a null pointer dereference at this line (do this similar to the argument we wrote in class on March 26). *Hint: Look at how the variables `filename` and `filenamesize` are initialized and how they change relative to one another in the while loop starting at line 697. Combining this with the test on line 701, you should be able to prove that `filename` will not only never be NULL on line 711, but that it is in fact pointing to a buffer large enough to hold a copy of `dir_name`.*

2. The `cppcheck` tool sometimes tries to guess the intentions of the programmer based on how the code is written, rather than exhaustively trying execution paths. One of the common warnings says "Possible null pointer dereference: xxxx - otherwise it is redundant to check it against null," where xxxx is a variable name. This message will have two indicated line numbers in the source file, where the first one is the questionable use of a variable (possibly a null pointer dereference), and the second one is a line in which the programmer tested whether the variable was null, with no change to the variable between these two spots. The reasoning is this: why would the programmer test whether the variable was null, unless there is a possibility that it could in fact be null. And if it could be null, it shouldn't be dereferenced.

    a. There are quite a few of these messages in the `cppcheck` output for `clamav-0.98.5` in the class sample static analysis outputs. In each of the locations listed below, there is in fact no possibility of a null pointer dereference, and it's the second part of the statement ("it is redundant to check it") that is the case. For this problem, you are to look at one of these locations in the code, and write up a clear argument about why that pointer usage cannot be a null pointer dereference. The location you should look at is determined by the first letter of your UNCG/cmpunix username, and the location is assigned in the following table.

Table 1: Assignment of cppcheck notices by first letter of username

c-d	`freshclam/manager.c:851`
e-i	`libclamav/readdb.c:232`
j	`sigtool/sigtool.c:2259`
k-p	`sigtool/sigtool.c:2348`
s-w	`win32/3rdparty/libxml2/SAX2.c:1170`

b. One of the messages is a little harder to reason about than the others, and requires looking at how a variable changes across two different functions, as well as thinking about how Boolean expression short-cutting works. Write up a clear argument for location `shared/actions.c:85`

c. If the check is redundant, can you think of any reason it is good to keep in the code?

3. For this problem, you are to run both the clang static analyzer and `cppcheck` on `getfile.c` that you wrote in Assignment 1. This will identify potential problems, and you should address each one by either fixing the problem or writing a justification of why this really isn't a problem. If you did not get this program written with your earlier assignment, you can use the one that is on `cmpunix` under `/csc495/hw3/getfile.c`.

   Before you start, make a copy of your original file as `getfile-orig.c` — this will allow me to compare your final version, after you make corrections, with the original code.

   a. You will first need to set up directories so that the clang static analyzer will be able to output its HTML report somewhere that you can access with a web browser. First create a subdirectory named `public_html` under your home directory using the `mkdir` command.

      Next, you need to run the clang static analyzer using the `scan-build-3.6` program — you will need to tell it where to output the report (`~/public_html`) and the command to use to build your program (using the `gcc` compiler. The following command does this:

      ```
      scan-build-3.6 -o ~/public_html gcc getfile.c
      ```

      Unfortunately, since the `cmpunix` system is set up by default to protect files so that other users can't read them, that means the web server on `cmpunix` won't be able to read them either. You need to change permissions so that the files can be read, and the easiest way to do that is with a command that says to recursively look at the files and directories in `~/public_html` and make them readable by all users:

      ```
      chmod --recursive a+r ~/public_html
      ```

      After running that command, you should be able to see the report by bringing up a web browser on your local machine, and going to the web page `http://cmpunix.uncg.edu/~username` (where you replace `username` with your actual user name). You'll actually see a directory listing with the scan results, where the name of the directory containing your report has the timestamp of when it was run. Go into that directory to see the actual report. For each error or warning, in your written assignment solution you should either describe how you fixed the problem, or justify that it isn't actually a problem.

   b. Run `cppcheck` using the following command:

      ```
      cppcheck --enable=warning getfile.c 2>cppcheck-errs.txt
      ```

This will output any errors and warnings to a file named `cppcheck-errs.txt`, which you can then look through to examine any problems. You should leave this file in your home directory so that I can find it when grading assignments. For each error or warning, in your written assignment solution you should either describe how you fixed the problem, or justify that it isn't actually a problem.

4. In this question, you will work with the basics of the HTTP protocol, and demonstrate why it is hopeless to hide secrets in different parts of HTTP or to base decisions on trust in honest browser functioning. There are multiple ways to solve each of the parts below, and hints are given for one particular solution. The descriptions assume you are using the Google Chrome browser, but Firefox works pretty much the same as far as these questions go.

   To start this question, use Google Chrome to access http://cmpunix.uncg.edu/hw3 — note that if you start this problem and come back to it later, always start here, and don't jump to an intermediate page directly. Each question below will reveal some "secret" that could be classified as "Steve's ultimate pronouncements on nerd culture" — that might not make sense now, but it should as you get through these questions. For each part below, in your written homework solution give a clear description of actions you took to answer the question, and give the secret that you discover (feel free to disagree with the secret, but you must justify your disagreement!).

   a. When you click on the "Question 4 page" link, there will be a secret value returned in a custom HTTP header field. You can't view HTTP headers without making a special effort to see them, and I would recommend using the Web Developer's tools that are built in to all recent versions of Google Chrome. First type Shift-Ctrl-I to bring up the developer tool console (if this doesn't work, you may have to go into the Chrome menu to find the tools). Next, select the "Network" tab in the tools console, and then click the "Question 4 page" link in the main browser window. After that page loads, click on the "hw3q4.php" entry in the network panel and look through the Response Headers for one that contains "CSC 495" in the name. That's your first secret!

   b. Cookies are another place where some naive developers try to hide secrets. There are multiple ways to examine cookies, but if you are using the Chrome developer's tools just click on "Resources" in the developer frame, and drill down under "Cookies" in the left-hand resources pane. Again, look for a cookie with "CSC 495" in the name. That's your second secret!

   c. After completing parts a and b, you should be at a web page with a button at the bottom labeled "Click Here to Go On." For this part, you are going to see how this works and how to tamper with its behavior. In the Chrome developer's tools, click on the magnifying glass, which is a tool that lets you select an element in the main web page to examine. After selecting this tool, as you move the mouse over the web page, different parts will be highlighted, so hover over the button so that only the text in the button is highlighted and click there. That should show you the HTML form that this button belongs to in the developer window. Here's an interesting trick: You can change the HTML that is loaded into the browser in the developer window. To do this, double click on an HTML field or attribute, and type in new values. To succeed at this part, you will need to change a field in the form, and the name of the field gives you a big hint as to what you should change it to. If you make the correct modification and then click the button, you'll get to a page that reveals the next secret.

d. After completing the last challenge, you'll be teased that there is another secret waiting to be revealed. To do this, you'll tamper with another part of the HTTP request: the cookies. Cookies seem hard to change directly in the browser: You can't just go into the cookies resource pane and change them, so how do you do it? Can the server trust the values it receives? Again, there are several ways to do this, but the most powerful technique to know about is how to directly execute JavaScript statements in your browser. Here's a web page that will give you a "how to": http://nategood.com/quickly-add-and-edit-cookies-in-chrome (If you have difficulty making that work, enter the JavaScript at the prompt in the "Console" tab of developer tools rather than in the address bar — I think that's easier anyway). After you change the cookie (you'll have to figure out which one!), reload the page and the final secret should be revealed.

e. In parts a-d, you saw that various normally-invisible parts of the HTTP protocol can be examined using the right tools, and values sent between browser and server can be tampered with so that a web server should never trust that a browser is behaving honestly. For this question part, try to come up with a way that a server could embed data into an HTTP interaction in such a way that it is both private and tamper-proof. You don't need to know anything more about HTTP than what you used in parts a-d, but think about the basic ideas of cryptography that we are discussing in class. Describe your solution here — I don't expect a perfect solution, but am more interested in how you think through this problem.

5. The browser same-origin policy (SOP) consists of several different policies, including an SOP for DOM access and an SOP for cookie access. The SOP for DOM access is based on the triple (protocol, host, port), while the SOP for sending cookies to sites involves the domain and path. Cookies designated "secure" are sent by the browser over HTTPS only. In current browsers, reading document.cookie in an HTTP context does not reveal secure cookies.

   Suppose the SOP for DOM access was defined using only host and port (as was the case in Safari until Safari 3.0) and did not include the protocol.

   a. A "network attacker" is someone who can intercept and modify network communications. Explain how such an attacker could steal facebook.com cookies that are designated "secure." Hint: A facebook user may login using a form served over https, but then receive a facebook.com page served over http. Consider injecting Javascript into the page served over http.

   b. A "web attacker" can set up a malicious web site and entice a browser to visit any site, but cannot intercept or forge network packets. Assume the user is willing to visit any site set up by the web attacker, but the web attacker can only set up web sites at domains other than facebook.com. For this part let's assume that the facebook.com site has no XSS vulnerabilities. Under these assumptions, is it possible for a web attacker to steal facebook.com cookies that are designated "secure?" Describe an attack or explain why you believe none exists.

Credit: The same-origin policy question is from CS155 at Stanford University