Security
29 June 2015

- This exam consists of two parts:
  (1) eight short questions, worth 5 points each and
  (2) one problem consisting of five parts, worth 10 points each.
  The final grade is calculated as $(PointsAttained + 10)/10$.
- Mark every page with your name and student number.
- You are *not* allowed to use a calculator, books, notes, nor other additional material.
- Do not use pencil or red ink.
- Answer in English. Encrypted answers are not accepted!

1 Short Questions (40 points: 8 × 5 points)

Answer all of the questions below. Make sure you answer all parts of each question.

1. What is the 'glue' in a DNS response? Why is it necessary? Provide a (simple) example of a query and a response which involves glue. (Make sure to include the record types.)
2. A user can be authenticated based on something they *know*, such as a password or PIN. What are the 3 other categories/factors which can be used to authenticate someone? Provide an example of each of them.
3. Give four examples of (substantially different) techniques which might be used for defense-in-depth of a web server, and (briefly) explain what they each defend against.
4. What does it mean for a cryptosystem to be 'Shannon secure'? Can a public-key cryptosystem be Shannon secure? What about one based on one-time pads?
5. What is meant by the *same-origin policy*? (Provide an example of two different origins.) Where is it used? Does it provide protection against related-path attacks?
6. Give an example of three (different) types of web server misconfigurations which could leak information which might be useful to an attacker.
7. In a threat model, what is a 'use scenario'? Provide an example of a supported use and an unsupported use.
8. How is the secret key calculated, using the Diffie-Hellman key exchange method? Does this help against a man-in-the-middle attack?

2 Problem 1: Code Review (50 points: 5 × 10)

The source code on the next 3 pages (Listing 2) contains 5 different vulnerabilities. You have to find them (by reviewing the code), and for *each* of the 5 vulnerabilities, answer the following questions:

1. Name the vulnerabilty, and provide a short description. [2pt]
2. Explain (briefly) why the vulnerability is a problem. [1pt]
3. Specify where the problem is located in the code. [1pt]
4. Explain why the code has this vulnerability. [1pt]
5. Describe how the vulnerability could be exploited by an attacker. [2.5pt]
6. Explain how the vulnerability could be prevented by changing the code. [2.5pt]

The code is responsible for the checkout process for a webstore. It has to compute the total amount that a customer has to pay, and verify the payment information they provide.

It is written in JavaServer Pages (JSP), which is a technology which is used to dynamically generate HTML webpages (similar to PHP) using Java code. The scripts are enclosed in delimiters. The delimiters `<% ... %>` contain a fragment of Java code that is executed when the page is requested. (The delimiters `<%@ ... %>` are only used below to import the `Util` class.)

You are only being asked to do a security review; you can ignore any other problems with the code. You should consider both what is present in the code, as well as anything which might be *missing*, but you can ignore any problems outside the scope of this code (for example, there is no need to consider DoS attacks, nor server misconfigurations).

The rest of this page contains an **example**. Turn to the next page to see the code you should review.

For example, Listing 1 contains some vulnerable C code (you don't have to understand it):

Listing 1: Example (this is *not* the code you should review)

```
1  int main(int argc, char** argv) {
2    char * name[16];
3
4    if (argc != 2) return 1;
5    strcpy(name, &argv[1]);
6    printf ("Hello %s!\n", name);
7    return 0;
8  }
```

And here is an example answer for *one* vulnerability (remember, you must discuss all 5):

1. The listing contains a buffer-overflow vulnerability, which is a vulnerabilty that allows an attacker to write data beyond the bounds of an array and overwrite adjacent memory locations. These adjacent memory locations can hold sensitive information that allows an attacker to compromise the execution of the application or modify the behavior in application specific ways.
2. An attacker can craft an input, passed as the first argument, that overwrites the return address with an attacker specified address. This allows the attacker to redirect the execution when the main function returns and can result in arbitrary code-execution.
3. The vulnerability is introduced by the unsafe function strcpy , located on line 5, that doesn't perform bounds checking.
4. The vulnerability can be patched by using the safer strncpy function. The `strcpy(name, &argv[1])` function should be replaced by `strncpy(name, &argv[1], sizeof(name) - 1)` followed by `name[sizeof(name) - 1] = '\0'`, because `strncpy` doesn't NUL terminate the destination if the source string is greater than or equal to the length of the argument!

Listing 2: JSP code to be reviewed

```
<%@ page import="work.security-home.Utils">

<%!
    public String decrypt(String data) {
      Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
      SecretKeySpec secretKey = new SecretKeySpec(SECRET_KEY, "AES");
      cipher.init(Cipher.DECRYPT_MODE, secretKey);
      return new String(cipher.doFinal(Base64.decodeBase64(data)));
    }

    /*
      The session is stored in encrypted form in a cookie with the
      following format:

      username=STRING&password=STRING&mode=STRING

      The password is stored using the unix password scheme.
      The mode can be either the string production or development, which
      allows developers to test the code by skipping the creditcard processing.
    */
    public Hashtable<String, String> getSession() {
      Cookie[] cookies = request.getCookies();
      if (cookies != null) {
          for (int i = 0; i < cookies.length; i++) {
              if (cookies[i].getName().equals("session")) {
                  String rawSession = decrypt(cookies[i].getValue());
                  return Utils.parseSession(rawSession);
              }
          }
      }
      return null;
    }

    public String escapeDbParam(String s) {
           String escape = "\x00\n\r\\'\"\x1a";

           for (int i = 0; i < escape.length(); i++) {
               s = s.replace(escape.charAt(i), "\\" + escape.charAt(i));
           }

           return s;
    }

    public boolean isAuthenticated() {
           Hashtable<String, String> session = getSession();
           String fmt = "SELECT * FROM users WHERE "
                      + "username = '%1s' AND password = '%2s'";
           String query = String.format(fmt,
                                        session.get("username"),
                                        session.get("password"));

           ResultSet rs = Utils.executeQuery(query);

           return Utils.getRowCount(rs) == 1;

    }
```

```
60      public int getDiscount(String coupon) {
61             String fmt = "SELECT * FROM coupons WHERE coupon = '%1s'";
62             String query = String.format(fmt, escapeDbParam(coupon));
63             ResultSet rs = Utils.executeQuery(query);
64
65             if (Utils.getRowCount(rs) == 1) {
66                rs.getInt("discount");
67             }
68
69             return 0;
70      }
71
72      public int computeTotalAmount() {
73             int amount = Utils.toInt(request.getParameter("amount"));
74             int donations = Utils.toInt(request.getParameter("donations"));
75             int discount = getDiscount(request.getParameter("coupon"));
76
77             int total = amount + donations - discount;
78
79             // To prevent a negative total amount (which happens if the
80             // discount > amount + donations), we return 0 in that case.
81             if ( total < 0 )
82                return 0;
83             else
84                return total;
85
86      }
87
88      public String escapeShellParam(String s) {
89             String escape = "&|><*?‘$(){}[]!#";
90
91             for (int i = 0; i < escape.length(); i++) {
92                 s = s.replace(escape.charAt(i), "\\" + escape.charAt(i));
93             }
94
95             return s;
96      }
97
98      public inDebugMode() {
99        Hashtable<String, String> session = getSession();
100       if (!session) reponse.sendRedirect(INTERNAL_ERROR_PAGE);
101
102       String mode = session.get("mode");
103       if (mode.equals("development")) return true;
104       return false;
105     }
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
```

```
121     public boolean isValidCC() {
122       // Skip verification if we are in debug mode.
123       if (inDebugMode()) return true;
124
125       String cardNumber = request.getParameter("cardnumber");
126       String expirationDate = request.getParameter("expirationdate");
127       String issuer = request.getParameter("issuer");
128       String cvv = request.getParameter("cvv");
129       String firstname = request.getParameter("firstname");
130       String lastname = request.getParameter("lastname");
131
132       // Because the Luhn algorithm implementation we have created isn't
133       // yet fully tested, we use our legacy CC verification tool to perform
134       // the validation
135       String fmt = "ccvalidate '%1s' '%2s' '%3s' '%4s' '%5s' '%6s'";
136
137       // Since we do not trust the users input, escape the string we are
138       // going to execute!
139       String command = String.format(fmt,
140                                      escapeShellParam(cardNumber),
141                                      escapeShellParam(expirationDate),
142                                      escapeShellParam(issuer),
143                                      escapeShellParam(cvv),
144                                      escapeShellParam(firstname),
145                                      escapeShellParam(lastname);
146
147       // This executes the command on the (UNIX) host machine.
148       Runtime rt = Runtime.getRuntime();
149       Process p = rt.exec(command);
150
151       // When the ccvalidate tool returns 0, the cc is valid.
152       return p.waitFor() == 0;
153     }
154
155     public void performCheckout() {
156       // First, check if we are authenticated.
157       if (!isAuthenticated()) {
158         reponse.sendRedirect(LOGIN_PAGE);
159       }
160
161       // Then, verify the provided creditcard information.
162       if (!isValidCC()) {
163         reponse.sendRedirect(INVALID_CC_PAGE);
164       }
165
166       // Finally if we are autenticated and provided valid CC
167       // information, finalize the checkout.
168       int amount = computeTotalAmount();
169       finalizeCheckout(amount);
170     }
171
172     performCheckout();
173 }
174 %>
175
176 <html>...</html>
```