Computer Security - Tutorial Sheet 3 - Solution
Network & Programming Security

Daniel Franzen

24th March 2013

Part A: Web Application Firewall

This question was partly inspired by the story of what happened to **HB Garry Federal** in 2011 (Google turns up plenty of news articles or see the Wikipedia page for a summary).

Here are some brief notes on the answers.

1. The HTTP protocol can be usefully restricted in many ways:

   - Check protocol conformance e.g. validating content or character encodings
   - limits size of request URI, query length
   - limit names or values of cookies
   - limit the number and size of headers
   - restrict use of POST and PUT or their transfer sizes

   Although a web server may allow configuration to restrict the HTTP protocol, configuring the restrictions in the WAF would allow the restrictions to be uniform and invariant across multiple web servers (it is mentioned that GBHarry is running at least two web apps), different web server version variations, etc. It also allows separation of security configuration from function hence better visibility of restrictions as they are gathered in one place.

2. Input normalisation allows the WAF to normalise URLs, character set encodings, HTML entities, back-slashes and the like. The reason is that pattern detectors then only need to work on the normalised patterns rather than any mix of formats, so they can be simpler and more efficient.

3. Negative policies are based on ruling out bad patterns, e.g., suspicious URLs, IPs known to be launch sites of attacks, etc. Postivie policies are based on only allowing traffic which fits the format specified by the application. The situation here is similar to that of anti-virus tools or intrusion detection systems.

   Some obvious points are:

   - Disadvantages of negative policies: have to be updated frequently or connect to online blacklists. The set of rules tends to grow over time and can become costly to check.
   - Advantages of negative policies: easy of configuration.
   - Disadvantages of positive policies: it could be very complicated to configure valid patterns for a sophisticated web application (an automatic learning based approach might be required). An version updates quite likely to change patterns significantly.
   - Advantages of positive policies: greater accuracy.

4. Possible responses by the WAF:

   - **router**: dropping a connection or blocking an IP address. Appropriate for a connection that has just been used to upload an attack vector, or an IP address which is connecting too frequently (DoS signature).
   - **WAF**: it might do rate-limiting
   - **application**: an example is disabling a user account if spam content is detected by the WAF.

5. The obvious missing feature categories are:

   - Configuration: the WAF should provide usable configuration mechanisms, perhaps out-of-the-box for commodity web applications, or automatically by providing a training phase.

- Monitoring: there should be a way for the WAF to raise an alarm in case a serious attack is detected, and at the least, it should provide a mechanism to monitor its status.
- Logging: the WAF should record audit logs of its activity, and perhaps of the validated web app traffic. All this data has to be protected against manipulation.

6. Some prudent considerations when deploying a WAF would be:

   - cost of configuration and maintenance (likely to exceed purchase price of tool and hardware);
   - additional risk, single point of failure: the WAF itself becomes a focus of attacks, and, especially if it implements security features like SSL encryption and authentication, may become a critical point of failure.
   - performance overhead: the WAF may slow access to the web site.

   The final decision might consider budgets and a risk assessment (how much business is lost by a security company that can't secure it's own website?) and they should certainly consider alternative solutions, perhaps such as using third-party cloud-hosted or externally managed applications.

Part B: Secure Programming

Question 3: Secure Programming

1. in class DBConnect(): **a hard-coded database server IP address in a constant String**
   A malicious user can use the javap -c DBConnect command to disassemble the class and discover the hard-coded server IP address. The output of the disassembler reveals the server IP address 129.23.45.77 in clear text.
   The solution retrieves the server url and the connection string from an external file located in a secure directory. Exposure is further limited by clearing the url from memory immediately after use.
   Note that the connection is still vulnerable to package sniffing, so additional authentication and encryption mechanisms have to be used.

```java
private String Connect() {
  try{
    char[] url = new char[100];
    BufferedReader br = new BufferedReader(new InputStreamReader(
    new FileInputStream("serveripaddress.txt")));
    // Reads the DB url into the char array,
    // Validate the url
    ...
    conn = DriverManager.getConnection (url);
    ...
    // Manually clear out the url
    // immediately after use
    for (int i = n - 1; i >= 0; i--) {
      url[i] = 0;
    }
    br.close();
  }  catch(SQLException ex) {...}
}
```

2. in class UserManagement, loginCheck(): **no input validation: log injection attack**
   The example code logs the user's login name when an invalid request is received. No input sanitization is performed. An unsanitised input could lead to a log injection attack. A standard log message when username is alice might look like this:

```
20 Feb 2012, 1:14:30 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login failed for: alice
```

If the username that is used in a log message was not david, but rather a multiline string like this:

```
alice
20 Feb 2012, 1:16:45 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login succeeded for: administrator
```

the log would contain the following misleading data:

```
20 Feb 2012, 1:16:45 PM  java.util.logging.LogManager$RootLogger log
SEVERE: User login failed for: alice
May 15, 2011 2:25:52 PM java.util.logging.LogManager log
SEVERE: User login succeeded for: administrator
```

Perform input sanitisation: validate the username input before logging it to prevent log injection attacks

```
  private void loginCheck (String username,
                String passwordHash) throws IOException {
    bool loginSuccessful = checkCredentialsValidity(username, passwordHash);
    if (!Pattern.matches("[A-Za-z0-9_]+", username)) {
      // Unsanitised username
      logger.severe("User login failed for unauthorized user");
      // no access ..
    }
    else if (loginSuccessful) {
      logger.severe("User login succeeded for: " + username);
      /* the user is directed to her/his own pages */ ...
    } else {
      logger.severe("User login failed for: " + username);
      /* restricted access to public pages */ ...
    }
  }
```

3. in class UserManagement, resetPassword(): **Generate weak random numbers**
This code uses the insecure java.util.Random class. The seed to initialize is hard-coded to `123`. Consequently, the sequence of numbers is predictable. An attacker could find the vulnerability by generating more than one new user. It is not very hard for an attacker to access/guess usernames of the existing users. using identical password, the attacker can login in as existing users and access the user profiles, sensitive info e.g. credit card info ...
Use the java.security.SecureRandom class to produce high-quality random numbers:

```
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
...
private void resetPassword (String  newPass){
  try {
    SecureRandom number = SecureRandom.getInstance("SHA1PRNG");
    Character c;
    newPass = null;
    for (int i = 0; i < 9; i++) {
      // Generate another random integer in the range of [0, 255]
      int n = number.nextInt(256);
      c = (char) n;
      newPass = newPass +  c.toString() ;
    }
  } catch (NoSuchAlgorithmException nsae) {
    // Forward to handler
  }
}
```

4. in class Purchase, getDate(): **returning private mutable class member's reference**
   An untrusted caller can manipulate a private Date object because returning the reference exposes the internal mutable component beyond the trust boundaries of class Purchase.
   This solution returns a clone of the Date object from the getDate() accessor method. While Date can be extended by an attacker, this is safe because the Date object returned by getDate() is controlled by class Purchase and is known to be a nonmalicious subclass.

```
  public Date getDate() {
    return (Date)d.clone();
  }
```

5. in class Purchase, createXMLQuery(): **input validation vulnerability; XML injection**
   A malicious user might input the following string instead of a simple number in the quantity field.

```
0</quantity></item><item><decription>Widget</description><price>1.0</price><quantity>50
```

   so the result query would be:

```
  <item>
    <description>Widget</description>
    <price>500.0</price>
  <quantity>0</quantity>
  </item>
  <item>
    <decription>Widget</description>
    <price>1.0</price>
    <quantity>50</quantity>
  </item>
```

   Even when it is not possible to perform such an attack, the attacker may be able to inject special characters, such as comment blocks and CDATA delimiters, which corrupt the meaning of the XML.
   Depending on the specific data and command interpreter or parser to which data is being sent, appropriate methods must be used to sanitize untrusted user input. This compliant solution uses whitelisting to sanitize the input. In this compliant solution, the method requires that the quantity field must be a number between 0 and 9.

```
private void createXMLQuery(BufferedOutputStream outStream,
              String quantity) throws IOException {
  /* Write XML string if quantity contains numbers only.
  Blacklisting of invalid characters can be performed
  in conjunction.*/
  if (!Pattern.matches("[0-9]+", quantity)) {
    // Format violation
  } else {
    String xmlString = "<item>\n<description>Widget</description>\n" +
            "<price>500</price>\n" +
            "<quantity>" + quantity + "</quantity></item>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
  }
}
```

   A more general mechanism for checking XML for attempted injection is to validate it using a Document Type Definition (DTD) or schema. The schema must be rigidly defined to prevent injections from being mistaken for valid XML.

6. in class Inventory, checkInventory(): **Memory leak**
   The vector object in the code example leaks memory. The condition for removing the vector element is

mistakenly written as n > 0 instead of n >= 0. Consequently, the method fails to remove one element per invocation and quickly exhausts the available heap space.
This can be fixed easyly by changing the second loop condition to n >= 0.

```java
public void checkInventory(int count) {
  for (int n = 0; n < count; n++) {
    vector.add(Integer.toString(n));
  }
  // ckeck any mismatch in the inventory list
  checkMismatch();
  ...
  for (int n = count - 1; n >= 0; n--) { // Free the memory
    vector.removeElementAt(n);
  }
}
```

Part C: Access Control

1. This kind of security hierarchy can be represented in a lattice:
   (a) The security lattice is depicted in Figure 1. Each node represents one security level.
   (b) The observer is watching. That means he needs to be on the level $(T, \{CS\}) \vee (S, \{CS\})$ which is $(E, \{CS\})$.
   (c) In this case write access is needed, thus the level needs to be $(T, \{CS\}) \wedge (E, \{CT\})$. Here this is $(S, \{\})$.
   (d) A partial order is a reflexive, transitive and antisymmetric binary relation.
   The reflexivity $A \leq A$is needed so that every subject can read and write the objects on her security level.
   Transitivity requires that if $A \leq B$ and $B \leq C$ then $A \leq C$; that is, if indirct information flow is possible from A to C via B, then we should allow direct information flow from A to C.
   Antisymmetry requires that if $A \leq B$ and $B \leq A$, then $A = B$. Given the reflexive and transitive requirements, antisymmerty merely eliminates redundant security classes. In other words there is no point in having two different security labels if objects whith these labels are restricted to having exacly the same information flows.

2.

	Exercise 1.2	Last tutorial	Exam ideas
Bob	{read,write}	{write}	{write}
Luke	{read}	{read,write}	{write}
David	{read}	{read}	{read,write}

3. • Current requested actions:

$$\begin{aligned} b = [&(David, Problem - Thread, write), & (1)\\ &(David, Problem - Thread, read), & (2)\\ &(Luke, Problem - Thread, write), & (3)\\ &(Luke, Problem - Thread, read), & (4)\\ &(Bob, Problem - Thread, write), & (5)\\ &(Bob, Problem - Thread, read)] & (6)\end{aligned}$$

   • Access Marix:
   $M =$

	Ex. 1.2	Last tut.	Exam	Problem-Thread
Bob	{read,write}	{write}	{write}	{read,write}
Luke	{read,write}	{write}	{write}	{read,write}
David	{read,write}	{write}	{write}	{read,write}

- Current security levels for Subjects

  $$\begin{aligned} L_C = [&Bob \mapsto (Student, \{CS\})], \\ &Luke \mapsto (Student, \{CS\})], \\ &Divid \mapsto (Student, \{CS\})] \end{aligned}$$

  Maximal security level for Subjects

  $$\begin{aligned} L_S = [&Bob \mapsto (Student, \{CS\}), \\ &Luke \mapsto (Tutor, \{CS\}), \\ &David \mapsto (Examiner, \{CS\})] \end{aligned}$$

  Security levels for objects

  $$\begin{aligned} L_O = [&Ex.1.2 \mapsto (Student, \{CS\}) \\ &Lasttut. \mapsto (Tutor, \{CS\}) \\ &Exam \mapsto (Examiner, \{CS\}) \\ &Problem - Thread \mapsto (Student, \{CS\})] \end{aligned}$$

In order to say whether this transition is safe we have to evaluate, whether both states are safe. That means validating the ss-property, the $\star$-property and the DS-property.
The implicit represented start state is safe: All subject have their maximum clearence, everybody is only writing at his own level, nobody is apending-down or reading-up ($\rightarrow$ ss-property, $\star$-property). The permission matrix is accordingly (DS-property).
In the new state we adjusted the permission matrix to match the security needs (DS-property) and all operations are only on one thread ($\rightarrow$ ss-property, $\star$-property).
Thus the transition is safe.

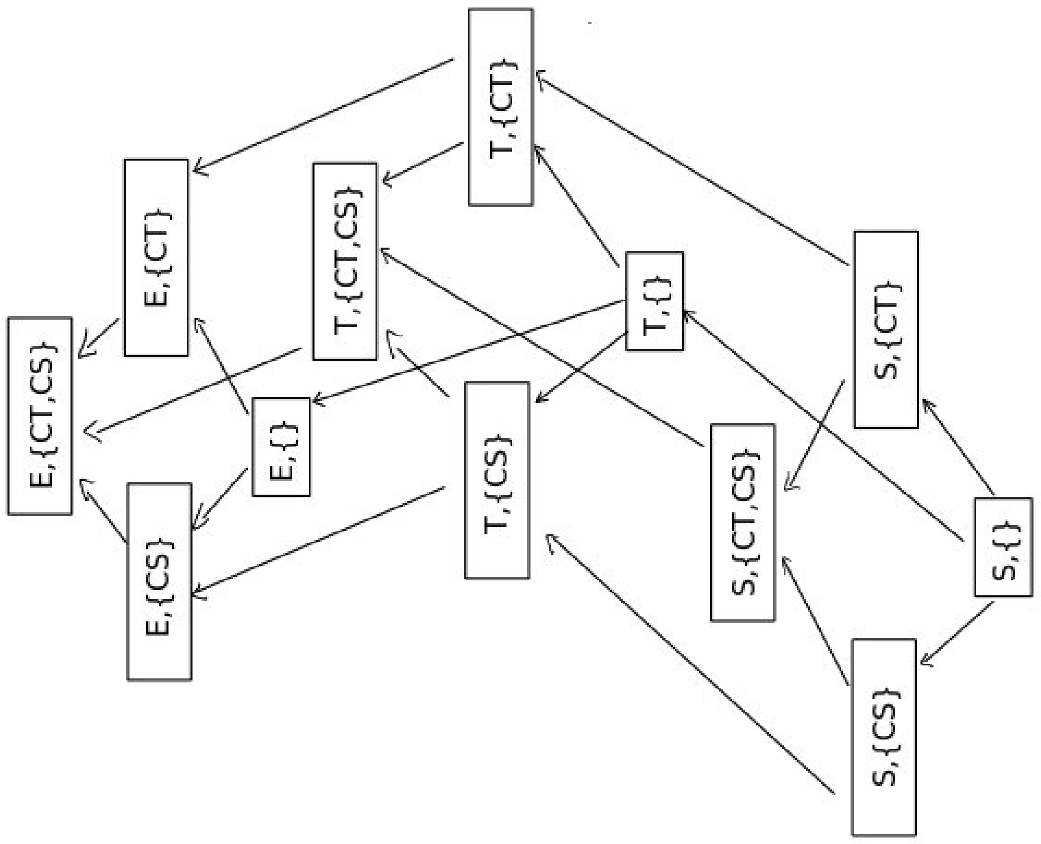

Figure 1: C.1a: Security Lattice