University of Hawai'i Manoa

Assignment 2:

SDL: Requirements and Design

Team Alpha

Jay Watanabe

Michael She

Joseph Li

ICS 491

Dr. Barbara Endicott-Popovsky

July 12, 2016

HealthE

Introduction

Our Java applet HealthE will be designed to store user health data remotely on a web server. This paper will address its requirements, design, and threat modeling, among other things.

1. Security Requirements

Due to the fact that sensitive user data is stored and accessed on a server, the establishment of security requirements is needed. Two of its largest concerns include the assurance of health privacy and the integrity of data. In other words, *how should the user's sensitive data be only available by privileged access,* and *how may the data's integrity be kept incorruptible and protected?*

One answer is our applet shall adhere to Oracle's Java SE security page. Within Java there is built-in language security features enforced by the compiler and virtual machine, preventing hostile code from corrupting the runtime. Furthermore, our applet shall use cryptography (via Java Cryptography Architecture) for a wide range of cryptographic services such as digital signatures, embedding authentication and access control (via Java Security Architecture), using secure communications (via Java Secure Socket Extension) to authenticate peers over a network to protect the integrity and privacy of data, and contain a public-key infrastructure to handle secure certificates (using Java CertPath API, X.509 certificates, and CRLs).

A second answer is adherence to Java SE Secure Coding Guidelines. One fundamental principle of Java security is to "prefer to have obviously no flaws rather than no obvious flaws." (Java SE Secure Coding Guidelines). Adherence to these secure coding practices ensures that trusted code is memory safe and robust, executed securely, so that vulnerabilities are minimized.

Minimizing flaws prevents denial of service and unintended release of confidential information, protecting from injection attacks and validating input, and so on.

2.Bug Bars

The privacy bug bar for the end-user is the following:

Critical	● Lack of notice and consent ● Lack of user controls ● Lack of data protection ● Improper use of cookies ● Lack of internal data management and control
Important	● Lack of notice and consent ● Lack of user controls ● Lack of data protection ● Improper use of cookies
Moderate	● Lack of user controls ● Lack of data protection ● Improper use of cookies ● Lack of internal data management and control
Low	● Lack of notice and consent

The security bug bar for the client is as follows:

Critical	• Network Worms or *unavoidable* common browsing/use scenarios where the client is "owned" without warnings or prompts. • Elevation of privilege (remote): The ability to either execute arbitrary code *or* to obtain more privilege than intended

Important	• Common browsing/use scenarios where client is "owned" with warnings or prompts, or via extensive actions without prompts.
	• Elevation of privilege (remote)
	  ○ Execution of arbitrary code *with* extensive user action
	• Elevation of privilege (local)
	  ○ Local low privileged user can elevate themselves to another user, administrator, or local system.
	• Information disclosure (targeted)
	  ○ Cases where the attacker can locate and read information on the system, including system information that was not intended or designed to be

	exposed.
	• Denial of service
	  ○ System corruption DoS requires re-installation of system and/or components.
	• Spoofing
	  ○ Ability for attacker to present a UI that is different from but visually identical to the UI that users must rely on.
	• Tampering
	  ○ Permanent modification of any user data or data used to make trust decisions in a common or default scenario that persists after restarting the applet.
	• Security features: Breaking or bypassing any security feature provided

Moderate	• Denial of service
	  ○ Permanent DoS causes JVM to crash, requires cold reboot or causes Blue Screen/Bug Check.
	• Information disclosure (targeted)
	  ○ Cases where the attacker can read information on the system *from known locations*, including system information that was not intended or designed to be exposed.
	• Spoofing
	  ○ Ability for attacker to present a UI that is different from but visually identical to the UI that users *are accustomed to trust* in *a specific scenario*.

Low	• Denial of service
	  ○ Temporary DoS requires restart of applet.
	• Tampering
	  ○ Temporary modification of any data that does not persist after restarting the applet.

3. Security and Privacy Risk Assessments

In order to know what parts of our applet require security, we need to consider what type of information that attackers may attempt to gain access and require protection. The type of information in our applet is Personally Identifiable Information (PII), which if stolen could harm clients. The type of data we will store/transfer will be health data that will be very valuable if stolen. Our data will be stored on a server using the latest security patches and technologies, ensuring security and confidentiality. With that said, any data going in and out of the server must

be protected so that data integrity will be intact– not manipulated or intercepted in any way. Even if it were to be intercepted, that data should not be able to be read without having some sort of key to read it.

4. Design Requirements

The applet generally transfers data between a device and a server, thus the server will need to be able to handle a lot of data travelling in and out as well as a clean, secure connection to talk between. Each user must create an unique account and use that account to input their health data. The stored information is very important; therefore, data in transmit needs to be encrypted so that an outside source cannot access the data. The data flow will be designed in such a way that it uses a secure channel to allow a safe transfer of data between the server and the user's device.

The account creation design should not ask for too much personal information and should just simply require an email and password. The design of the user interface should be appealing, generally simple and user friendly. The login screen would just ask for the account email and password. Next after logging in, user data should be easily viewable and any new changes would be easy to make and input. Any and all input will need to be validated and sanitized to prevent any HTML and SQL code from being injected and accepted. General input errors in the login screen should never specify what is incorrect. Multiple failed attempts should lock the account and require some other sort of verification such as a security question or two-step validation by a secondary device. Multiple backup copies of the data of the server shall be made regularly in the case of any attacks where communication to the app was halted or data potentially may be corrupted.

Some examples of the information that the HealthE applet would keep track of include sleep times, calories consumed, workout routines, calories burned, etc. The GUI for sleep time could display a graph bar showing how much sleep the user gets per day with varying filters. As for calories consumed, a calorie counter would graphically display and keep track of what was eaten per day as entered by the user. Workout routines may be represented by a set schedule of what to do daily with how many calories that kind of exercise would burn and a log of what was actually accomplished.

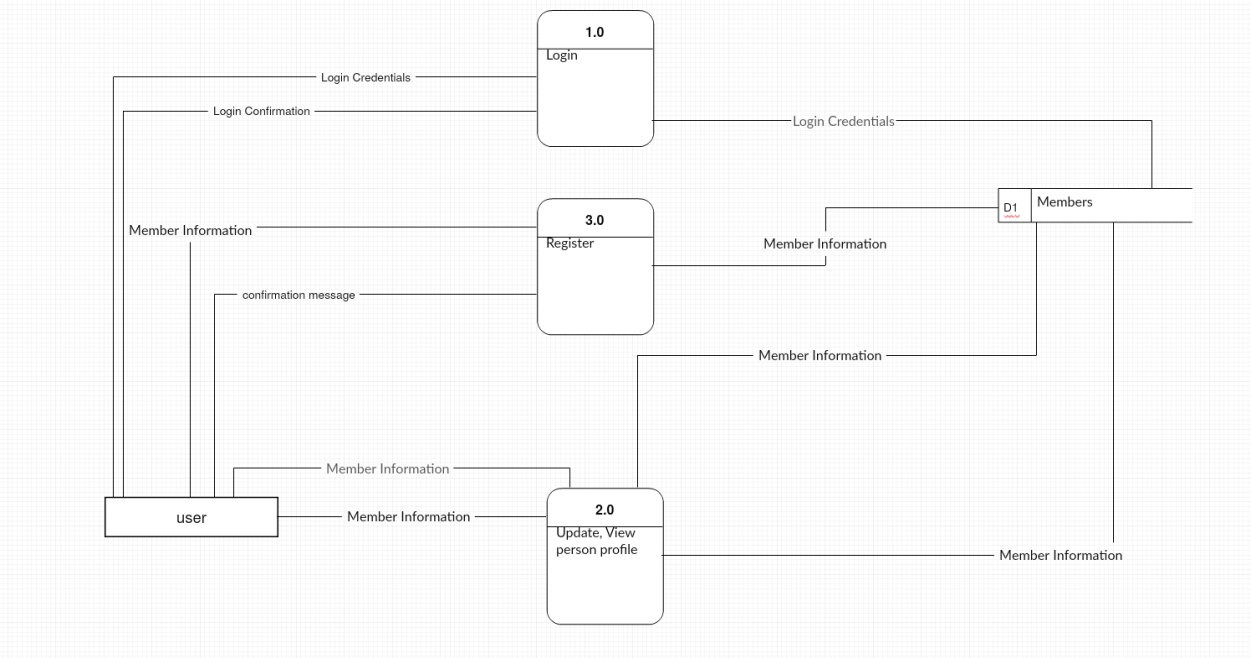
5. Attack Surface Analysis / Reduction

We will have three levels of access. Level 0 users are all users that are not logged into the service. Level 0 users will have no direct access to the database and are only able to access the login and register processes. They are able to compare login credentials and register as a new user in the members database. They are not permitted to alter anything else, aside from creating a new entry. Level 1 users are users with a registered profile. They have access to the view and edit profile process. Through this process they may view and edit data in the sections corresponding to their login credentials. This user would still not be capable of directly accessing the member database. The final Level 2 user is equivalent of an administrator account, which has access to the entire members database directly, permitted to delete users and modify accounts. There would only be a single Level 2 user account.

The attack surface of our applet is concerned with how user levels are controlled, how access to the database is managed, how the data is transferred to the database, how the processes are operated, and how logins are authenticated. With user levels clearly defined, no one should be able to access unpermitted areas according to their assigned level. Since database access is a

very commonly attacked area (take for instance how prevalent buffer overflows are!), it is of utmost importance that we ensure our database can be accessed only as we want it to be. All data must be secured while transferring back and forth from the database, processes, and user. Lastly, login authentication is a another commonly attacked area, so we must make sure how we verify users is also secure.

6. Threat Model



The diagram above is our applet's threat model. Our greatest threats mostly come from users accessing data and profiles not linked to their specific account. These threats relate to the Spoofing and Tampering portions of the STRIDE threat model.

Spoofing could potentially occur if an attacker mimicked the login of another user and access their profile. This would allow the attacker access to information of another user, which is not allowed. This means that we must always maintain proper authentication during and after login.

Furthermore, there is also a potential for attackers to tamper with the members database. Users should only be able to modify data in their own respective profile through the approved listed processes. Thus, proper user access and privileges must be maintained.

References

Java SE Security. (n.d.). Retrieved July 16, 2016, from http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

Secure Coding Guidelines for Java SE. (n.d.). Retrieved July 16, 2016, from http://www.oracle.com/technetwork/java/seccodeguide-139067.html

University of Hawai'i Manoa

Assignment 3:

SDL: Implementation

Team Alpha

Jay Watanabe

Michael She

Joseph Li

ICS 491

Dr. Barbara Endicott-Popovsky

July 22, 2016

HealthE Implementation

1. Approved Tools For Use

Tool	Required Version	Comments
Java SE JRE	8u101	- Includes basic tools needed to run java applications - Includes Java Virtual Machine, browser plugin, standard Java libraries, and a configuration tool
NetBeans	8.1	- Java IDE for running applet and performing unit tests; also useful for designing GUI with JavaFX using SceneBuilder
PostgreSQL	9.6	- Creates and runs database required for application

The Java SE JRE is the base development package provided by Oracle, needed to run Java applications. Our specific application is designed using PostgreSQL, an open-source relational database management system. NetBeans is our preferred IDE tool for the development process, due to its beefy feature set and ample support from Oracle; additionally, it has a multitude of JavaFX examples readily available. Older JRE versions and older PostgreSQL versions could potentially be usable with our Java applet, but it was designed with these tools and versions in mind.

2. Unsafe Functions

Deprecated Class/Function	Replacement	Notes

java.rmi.RMISecurityManager (class)	java.lang.SecurityManager	- Implements a security policy
java.lang.SecurityManager (numerous functions)	java.lang.SecurityManager. checkPermission()	- Numerous check functions within class have deprecated, checkPermission() used instead

Security manager allows for implementing of a security policy, which polices operations within an application. The policy is able to determine whether an operation is safe, in a secure context, and can allow or prevent operations from running. Since the RMISecurityManager class is deprecated in the current version of Java, the SecurityManager class should be used instead. Numerous check methods within SecurityManager have also become deprecated, and replaced by the checkPermission() method instead.

3. Static Analysis

Name	Version	Purpose
Additional Java Hints NetBeans Plugin	1.5.0	Checks code style, general code cleanliness
FindBugs Integration NetBeans Plugin	1.28	Performs broad range, general security checks
VisualCodeGrepper	2.1.0	Searches for insecure code, OWASP standards, common security issues

We decided to go with Additional Java Hints, FindBugs, and VisualCodeGrepper due to their ease of use and the range of issues they can cover. Each of these tools are quick to download and setup, able to be used immediately. Additional Java Hints and FindBugs are both

available as plugins for Oracle's NetBeans, our IDE of choice, downloadable through the IDE directly.

After installation, it is possible to run both of them, along with the other built in code analysis tools, at once to generate a list of potential issues. VisualCodeGrepper is a standalone program, however it is easy to designate code to scan and provides a clear, easily understood list of issues. It is a top choice for checking Java code amongst developers.

The purpose of Additional Java Hints is style and code cleanliness. It warns of deprecated functions, joins literals, replaces +, etc. This helps clean up the code, and increases the quality. This tool doesn't often return dangerous bugs or critical issues, but manages to keep code simpler, more functional, and clearer. Checkstyle was also considered in this place, it however was more of a cosmetic tool. Checkstyle does much of the same style checks, but does not really affect the content of the code, which is why we decided to go with Additional Java Hints.

FindBugs does as exactly what it sounds like– it finds bugs within your code. FindBugs searches byte code, not source code, and performs a deeper search than a style checker. It covers a broad range, including checks for many common security risks like SQL injection, hardcoded passwords, etc. VisualCodeGrepper has a lot of overlap with FindBugs, but it searches source code. Like FindBugs, VisualCodeGrepper performs a variety of complex searches, covering issues like buffer overflow, as well as OWASP recommendation violations. The OWASP checks stood out to us as we recently went over what OWASP is and its credibility in the software security community. Another plus to these two tools is they come recommended by the CERN computer security team, CERN being a leading scientific research center.

References

Deprecated List (Java Platform SE 8). (n.d.). Retrieved July 22, 2016, from https://docs.oracle.com/javase/8/docs/api/deprecated-list.html

Static Code Analysis Tools. (n.d.). Retrieved July 22, 2016' from https://security.web.cern.ch/security/recommendations/en/code_tools.shtml

V2.1.0,. "VisualCodeGrepper V2.1.0". (n.d.). Retrieved July 22, 2016' from https://sourceforge.net/projects/visualcodegrepp/?source=typ_redirect

University of Hawai'i Manoa

Assignment 4:

SDL: Verification

Team Alpha

Jay Watanabe

Michael She

Joseph Li

ICS 491

Dr. Barbara Endicott-Popovsky

July 25, 2016

HealthE Verification

1. Dynamic Analysis Tools

For our dynamic analysis tool we considered Spock and JUnit, however ended up choosing to go with JUnit. Spock is an alternative to JUnit testing framework that is growing in popularity. Spock relies on a dynamic language called Groovy, though it is still based in the Java platform. In addition to Groovy being generally easier to read than vanilla Java, Spock includes easier parameterizing of tests, additional test related syntax, and overall revolutionize the approach to testing that is associated with JUnit. For most purposes it appears that Spock would be the more efficient, intuitive tool, but it would also require learning Groovy and adapting to this different tool, which does not seem feasible within the time constraints of this project. This ultimately tipped the scales into JUnit's favor.

JUnit is very popularly used for Java testing, so finding general use tutorials was easy. Using JUnit in NetBeans proved to simplify certain test cases but, in other test cases, it seem to overcomplicate the process of testing. We were able to verify that data was correctly processed in class methods through repeated testing of classes methods automatically. However, we found it complicated the testing of the SleepController class, producing confusing Initializer errors. We were not sure if controllers were normally tested either as part of a MVC design, but perhaps the errors were due to an attempt to test a class not meant to be tested.

2. Fuzz Testing

For the fuzz testing, we decided to test for SQL injection, try random inputs, and our attempt to break the UI for the graph portion. In general, we chose these examples as they are fairly common issues that are important to keep in mind in any programming endeavor.

The first test we performed was attempting the SQL injection. SQL injection should always be checked when working with and SQL databases, as it is one of the most common and prevalent vulnerabilities. Currently we have a database setup using postgresql, and have usernames and passwords set up within a table in this database. Our login system consists of comparing the user input to this table in order to verify whether the user has entered correctly the information corresponding to an existing account. Initially when we had coded this system, we took the input and hardcoded the username and passwords into an SQL statement to check against the table. After going through the lab and material regarding this in class, we have since rewrote the code to use prepared statements to prevent any unwanted manipulation of our database. Prepared statements serve to prevent any SQL injection by inserting the input directly into the database, preventing any attempted SQL commands from being executed. We attempted to insert simple SQL commands, for example by entering “password OR 1=1” into the password box. These attempts were all prevented by the prepared statements.

The second test we performed was entering random inputs into our application. This currently includes our login screen, as well as a graph to display hours slept/day. This is another issue that should always be checked, how an application handles different or unexpected inputs. For the login portion, it handled any inputs we could throw at it. This was handled by the prepared statements, along with a catch for SQL exception. For the graph data collection, we have a calendar date picker control in which you can choose a date on a calendar, or enter your own date in the form of mm/dd/yyyy into a text-field, and a text-field for inputting the number of hours slept. The date picker control should be able to handle sorts of inputs, as to prevent any improper characters. It should remove letters and special characters, so only a proper date can be entered. The hour textbox handles not only integers but also numerical inputs with decimals. It

works by allowing you to type whatever you want, it will however throw an exception if the entry is not a number and will not take the input.

Our third test was attempting to break or cause the display to respond oddly. We decided to test this because the UI needs to remain functional regardless of how the application is used, otherwise it doesn't matter how well the rest of the application is designed. The interface is the only part a user should ever see, and will have immense impact on how users respond to and remember an application. Through our tests we found three issues with the graph.

The first two issues was with the calendar box. We have yet to deal with these issues, but we have plans as how to manage in the future. The first problem is that when you add days out of chronological order, they will still be displayed in the order they are added. This issue can possibly be fixed by building an entirely new graph every time a new day is added, rather than adding on to a single graph.

The next issue occurs when a user adds a day that has already been added. Our solution is that the graph shall update the day with the new number of hours of sleep. Furthermore, it could also generate a brand new graph using the newly updated series data.

The third issue came when inputting incredibly large numbers for the hours slept, the graph become unreadable due to the hours axis expanding to accommodate the numbers. This issue is a simple fix, we limited the hours input to 24, as nobody can sleep more than 24 hours a day anyway.

3. Attack Surface Review

The attack surface tools that we employed were FindBugs, Additional Java Hints, and VisualCodeGrepper. After initially using these tools, most issues have been dealt with. The main vulnerabilities found were related to SQL injection, which we have since amended. Additional

Java Hints served as a reminder during the actual coding process, with convenient reminders relating to style, making certain things protected, areas where short hand expression can be used, etc. This is nice, though many of the suggested changes did not seem strictly needed. FindBugs and VisualCodeGrepper provide a nice safety net in regards to things like the SQL vulnerabilities mentioned, OWASP standards we may not be aware of, and potentially dangerous code bits, etc.

References

Dziworski, Jakub. "Junit Vs Spock + Spock Cheatsheet". *JAKUB DZIWORSKI*. N.p., 2016. Web. 29 July 2016.

Winder, Russel. "So Why Is Spock Such A Big Deal?". *ACCU Professionalism in Programming*. N.p., 2016. Web. 26 July 2016.

University of Hawai'i Manoa

Assignment 5:

SDL: Release

Team Alpha

Jay Watanabe

Michael She

Joseph Li

ICS 491

Dr. Barbara Endicott-Popovsky

August 7, 2016

HealthE Release

1. Incident Response Plan

Our Privacy Escalation Team will be composed of the following:

- *Escalation Manager:* Joseph Li

- The escalation manager deals with ordering, structuring, and bringing to the attention of relevant parties information regarding incidents requiring their attention. They analyze the scenario and lead the team in resolving the issues.

- *Legal Representative:* Jay Watanabe

  - The legal representative oversees legal affair regarding the escalation incident. They provide any relevant information and advice in how to resolve and legal matters in the best interest of all involved parties.

- *Public Relations Representative:* Michael She

  - The public relations representative deals with interaction between the company and the general public. They work to maintain the positive image of the company and ensure proper communication between users and the company.

Contact:

Email Address: HealthESupp@gmail.com

Privacy Escalation Procedure:

First Determine:

- Source of escalation
- Impact and spread of escalation
- Incidents leading to escalation
- Summary of situation and information known
- Plan to respond and address the issues

- List of affected parties
- Employees responsible for area of escalation

Escalation Resolution May Include:

- Management of internal issues
- Communication and training to prevent future issues
- Adjustment of internal policies
- Publication of troubleshooting literature
- Informing of affected user base
- Updating of any relevant documentation

Resolutions should:

- Resolve the concerns of the reporting party
- Resolve any user concerns
- Work towards preventing similar events in the future

2. Final Security Review

As we looked at our applet for a final security review and our threat model, we saw the greatest threat was from user accessing the data not linked to their account. As it currently stands, this is true. Since each password is salted and hashed, hacking into another person's account is more challenging to crack.

Many of the warnings in Additional Hints were to change certain packages public or protected. It was not as useful as we hoped. We found FindBugs more useful and better, for it discovered more substantial bugs, such as null exceptions pointers and the like, which actually saved time and prevent terrible crashes. Still, FindBugs suggested solutions that were at times

worded confusingly, particularly for example with regards a serializable class and finding an "non-primitive instance field". We got rid of the extra Serializable code, since it turned out it was not needed, and code ran seemingly smoother. We were unsure of the overall effects of changing classes to protected, so only the red alerts were taken seriously.

Perhaps we were overambitious originally, but there are still many things we lacked, after all was coded and done. There was so much to learn; we omitted security policy and Security Manager's permission checking, and written logs were neglected as well. We find these are not a large problem on a small scale applet as this, but it may an issue with scaling up the applet. For instance, the data source connection would ideally be done remotely to a server rather than a local host (which was easier to test, since none of us had experience with setting up remote clients), but so many security issues occur by simply exposing data through the Internet.

Our experience and time limitations caused us to believe we did the best we could in the time given. Certainly the applet was useable but it felt like a sprint, not a marathon, and so it is not as perfect in form as we would like. Salting and hashing was implemented. There were compromises, rewritings of code, and further reviewing of code. In the end, we give the project a grade of *passed FSR with exceptions*. Such exceptions are what was listed above. If the applet program were any larger, we definitely would tighten down on security more before considering release. With all things considered, this applet is ready for release.

3. Release and archive link

Release link: https://github.com/josephyli/HealthE/releases/tag/v1.0

User Guide link: https://github.com/josephyli/HealthE/wiki/User-Guide