2012

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

STUDY & IMPLEMENTATION OF AUTHENTICATION BEST PRACTICE TECHNIQUES FOR TWO LEGACY SYSTEMS

Matthew Laird

A Capstone Project Submitted to the
University of North Carolina at Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Information Systems & Operations Management
Department of Computer Science

University of North Carolina at Wilmington

2012

Approved by

Advisory Committee

_____	_____
Dr. Douglas Kline	Dr. Laurie Patterson

_____
Dr. Tom Janicki, Chair

Accepted By

_____
Dean, Graduate School

ABSTRACT

The goals of this research project are to 1) explore the concept of "sign in" authentication and related best practices; to 2) investigate the operations of two authentication systems (the Cameron School of Business Web portal (myCSB), and Entropy, used at the University of North Carolina Wilmington); and to 3) design and implement a revised system to centralize the login process of the aforementioned systems. Additionally, the concept of third-party authentication will be explored, and the systems will be assessed for their adherence to authentication best practices. Authentication is an integral element of modern computing systems. People now spend a great deal of time interacting with computers and systems that employ identity verification techniques to protect personal and private information. Centralizing this process offers more user convenience and simplifies future work on the systems. This paper uses related literature to explore authentication as an idea and as a practice, in order to more effectively study the CSB portal and Entropy, and ultimately design and develop a strong foundation for a new centralized authentication process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I: INTRODUCTION

Computing technologies have improved by leaps and bounds over the past century. System applications to assist individuals have become more ubiquitous in society; it has become ordinary for people to trust them with personal and private information. In order to protect this information, efficient, dependable authentication systems are an absolute necessity. It is worth investigating authentication techniques (such as centralization) for the purpose of understanding and steadily improving them over time. This affords users a safer and simpler experience.

The objectives of this paper are as follows. First, the general idea of authentication is explored to emphasize its value and provide obligatory background for subsequent sections of this document. More importantly, best practices and standards are outlined, as authentication techniques have changed over time. Second, the authentication features of two current Web-based existing systems—the University of North Carolina at Wilmington's (UNCW) Entropy grading system, and Cameron School of Business Web portal (myCSB)—are evaluated. This evaluation is necessary to accomplish the third objective of this paper: using systems analysis techniques to create a design for centralizing the authentication aspect of myCSB and Entropy systems, and ultimately preparing a working prototype to pave the way for future implementation. Additionally, third-party authentication techniques such as OpenID are explained, and strategies for their implementation are delineated.

This project builds on a wealth of information that is available regarding authentication. Certain standards have been reached in the computing community through

extensive trial and error. Much of this information is freely available online, including various techniques that have been applied to accomplish the goals of this project.

The remainder of this paper is structured as follows. Chapter II contains background on authentication, on the systems to be evaluated, and on the problems and goals of this project. In chapter III, several systems analysis methodologies are explored, including the methodology chosen for this project. Possible alternative implementations are covered in chapter IV. Chapter V consists of information regarding the selected implementation method for this task, while the actual process of this implementation is covered is chapter VI. Chapters VII and VIII focus on the third-party authentication analysis and best practices assessment of the current systems, respectively. A summary of this project is detailed in chapter IX, including the project timeline, lessons learned, and future work.

CHAPTER II: BACKGROUND

2.1 *Background on "Sign-In" Authentication*

Typically, when one considers authentication, verification of user identity is what immediately comes to mind. This is expected, due to the increasing need for refined security practices in current computing technologies. Authentication can be defined as an "access control that verifies an individual is the person he or she claims to be." (Shelly, Cashman, & Vermaat, 2007) Authentication may also be viewed as "any protocol or process that permits one entity to establish the identity of another entity." (Mallow)

The concept of authentication has existed in some form for millennia. Based on the preceding second definition, one can surmise that authentication can be performed by humans. This kind of authentication is based in the physical world, and involves determining a person's identity based on certain inherent aspects—examining their face, body language, and demeanor, listening to their voice, or even investigating their handwriting. These methods of identity verification have been used throughout humankind's history.

However, there is an element of personal affiliation with those methods. What if nothing is known about the authentication target? Leaving aside biometrics, passwords have a long tradition of providing authentication, particularly in a military context. For instance, couriers would relay vital information between high-ranking military leaders. These message-bearers couldn't be identified solely by their physical traits—so, an agreed-upon password was used (often in conjunction with a physical object) to verify their identity as a valid courier with an official dispatch.

Authentication—especially password-based authentication—has now become an integral component of modern computing. Businesses store a great deal of users' personal and private information, including credit card numbers and account balances. Reliable storage is only part of the equation—the other part of the equation is ensuring that anybody who accesses this information *is who they say they are*. Authentication is compulsory not only for businesses and educational institutions, but for everyday users, as well. Users frequently interact with systems in which their identity must be proven, such as ATMs. It has become even more common for users to possess many, many accounts on the Web. Each of these accounts—some of them relatively frivolous (social networking), some crucial (banking)—likely require a username and password, or some variant of this arrangement.

For more detailed information about Web-based authentication as a practice, please refer to chapter 4.3 of this document: *Authentication Best Practices & Standards*.

2.2 *About the Existing Systems*

This research project focuses heavily on the Cameron School of Business Web portal [https://csbapp.uncw.edu/csbsi/], and the Entropy grading service [https://csbapp.uncw.edu/entropy/]. Both of these websites were created in support of the Cameron School of Business, and the UNCW ISOM (Information Systems/Operations Management) department. It should be noted that both the CSB portal and Entropy were incrementally developed by UNCW students, and continue to grow and change according to the needs of the users. For example, the systems have begun to make more use of internet cookies (containing more personal information) than originally.

Both systems are regularly used by students taking classes that fall under the ISOM umbrella; additionally, myCSB has a wider range of users. While the sites are utilized by many of the same people—both at the student and faculty level—they offer different functionalities. Entropy functions as a "class management" system, allowing instructors to keep track of classes and students, grade effectively, administer quizzes, automatically grade assignments, and more. The CSB portal is aimed at providing more general services, such as business week sign-up, advising time reservations, mentor surveys, CSB scholarship applications and faculty/staff directory listings. Many of these utilities are available for the benefit of external users—business week guests and CEN mentors, for example—who do not have their own UNCW credentials. Currently each system handles authentication separately.

2.3 *Systems Specifications*

Upon examination, it was found that the authentication procedure is handled similarly in both systems, with some minor differences. Both were developed in ASP.NET/Visual Basic in Visual Studio, and interact with MS SQL Server databases. For each website, several aspects were studied on the code side: login, logout, cookies, lost and forgotten passwords, user-changed passwords, secure HTTP (HTTPS), encryption, and timeouts. On the database side, the following aspects were examined: any tables essential to authentication, stored procedures (SPs or sprocs), and encrypted stored data. **Figure 1** displays the current systems architecture for authentication in the two systems.

myCSB

Client—The login form is located on a root page called *Default.aspx*. The source code for this page reveals that the form is a single entity called *Login1*, containing a username textbox, password textbox, "remember me" checkbox, and sign-in button.

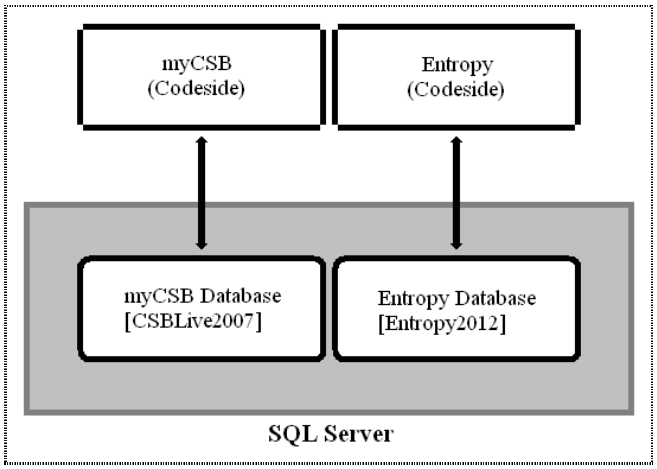


Figure 1 - Current Systems Architecture for Authentication

As with all .aspx designs, this page has underlying code written in VB (*Default.aspx.vb*). In the code, a variable designated *u* is defined as an instance of the *Person* object; the specifications for this object are located in a folder called *App_Code*. When the default page is visited, a load method runs that checks for HTTPS, and resets the session variable. When the user clicks the sign-in button, a method designated *Login1_Authenticate* runs. It first calls another method named *getLoginInfo*, which performs the RC4 encrypted password matching with the database. If *getLoginInfo* returns true, the user is successfully authenticated, and method *sendToMenus* sets a cookie before redirecting the user to the proper page based on their role (student or faculty), defined in the *Person* instance.

The logout function is handled through a root-level page called *Logout.aspx,* and operates by destroying the session, then using ASP.NET's built-in Forms authentication to sign out and redirect the user. The page to retrieve lost passwords is also at root level

(*forgotPassword.aspx*); this page does not use HTTPS as no password information is sent over the web, the password is sent via an email. Because it is a function that all legitimate users have, regardless of role, the page for changing passwords (*chgPassword.aspx*) is located in the *common* folder of the site; this page does use HTTPS. Persistent login is enabled for myCSB, but is not used on the production site; it is used on the intern test site that is currently under development.

Database—There are a handful of tables required for myCSB authentication. These tables are as follows: tblPerson, tblPersonDesiredEmail, tblPersonUnivEmail, tblPersonRoles, validPersonTypes, and validPersonRoles. The tblPerson table contains the most obvious vital information that makes up an instance of a Person, including their first and last names, university ID, email, and password (which is encrypted and stored as binary via RC4). The peripheral tables contain information that all applies to all users (such as their role), or information that may fall under a single category with multiple types (such as address).

There are several authentication-related stored procedures. To enable current students, alumni, faculty and external CSB mentors to sign in, there must be various methods of identity verification. SP *uspLoginPartA* performs the actual matching, and checks for one of two login identifiers: a number beginning with “850”, or an applicable email address. If a match is found, the user role is read and added to the cookie, and the user is directed to the proper menu; if the match is unsuccessful, a flag of -1 is returned from the sproc and authentication fails. Two other SPs—*uspPasswordChange2008* and *uspPasswordRetrieve*—allow users to change and recover their passwords, respectively.

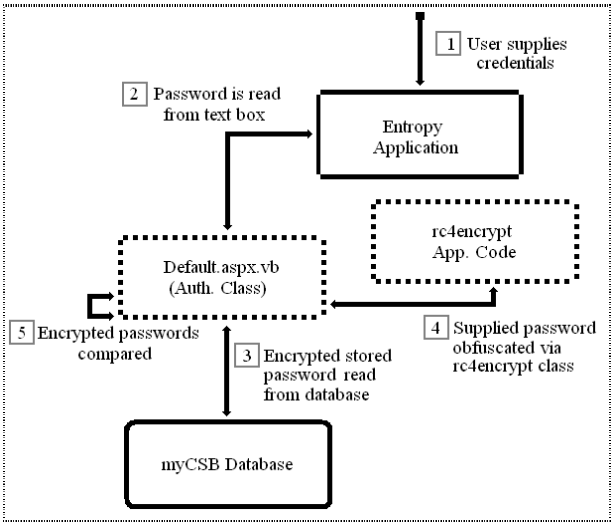


Figure 2 - Simplified RC4 Encryption Process

Entropy

Client—The code side of Entropy is similarly structured to that of myCSB. As before, the landing page is at root level, called *Default.aspx*. This page includes the login form; unlike myCSB, the Entropy login form is not a single entity containing multiple authentication-related controls, but instead several separate controls that are referenced individually.

Again, *Default.aspx.vb* contains the underlying code. In the code, a variable designated *u* is defined as an instance of the *Person* object; the specifications for this object are located in a folder called *App_Code*. When the landing page is visited, a load method runs that checks for HTTPS, and resets the session variable. When the user clicks the sign-in button, a method designated *uiSubmit_Click* runs. This method first runs another method, *getLoginID*, which communicates with the SQL database and performs the RC4 encrypted password matching. **Figure 2** displays a simplified version of the encryption process. After a successful match, *uiSubmit_Click* creates a cookie for the

user, and redirects the user to the proper page based on their role (student or faculty), defined in the *Person* instance.

The logout function is handled through a root-level page called *Logout.aspx,* and operates by abandoning the session (using forced expiration), then using ASP.NET's built-in Forms authentication to sign out and redirect the user. The page to retrieve lost passwords (*forgot.aspx*) is in the *user* folder; this page uses HTTPS, though it is not necessary, as no passwords are actually transferred. The page for changing passwords (*pwdChange.aspx*) is also located in the *user* folder of the site; this page uses HTTPS. Because this application handles grades, persistent login is not available.

Database— There are a few tables required for Entropy authentication. These tables are as follows: tblPerson, tblPersonRoles, and validRoles. The tblPerson table contains the most obvious vital information that makes up an instance of a Person, including their university ID, name, and password (which is RC4 encrypted). The other tables focus on the possible role of the user—whether they are student or faculty-level.

There are several authentication-related stored procedures. SP *uspLogin* performs the primary matching, and checks for an identification based on a valid university ID. If a match is found, the user role is determined via the tblPersonRoles table. SPs *uspPasswordChange* and *uspPasswordSetNew* allows users to change and reset (a temp password is emailed to the user; this is procedure for forgotten passwords on Entropy) their passwords, respectively.

2.4 *Problems*

There is a great deal of overlap in the user (person tables) base of Entropy and myCSB. However, authentication is handled separately for each site (and entirely

independently from the UNCW domain). This creates more difficulty for users, since they must remember multiple passwords. This problem is exacerbated by the fact that Entropy and myCSB have dissimilar "active" periods; myCSB has fewer peaks of activity over a wider range of time, while Entropy may receive very frequent use, or none at all (this depends on the class and instructor partiality). Furthermore, these systems are the subject of numerous student projects. When working with both systems, complications may arise due to their total sovereignty. The decision to avoid using UNCW domain authentication is necessary, since students involved in the development of the system cannot have access to the internal UNCW domain. Lastly, software standards evolve and change rapidly; in only a few short years, certain practices may become less efficient or entirely obsolete. Both myCSB and Entropy should be examined to ensure they align with authentication best practices.

2.5 *Project Goals & Benefits*

Students, faculty, CSB alumni, and CSB volunteers act as the key stakeholders for this project. By centralizing the authentication process, users need only remember a single password (in addition to their university ID), thereby adding an element of user convenience to these regularly-used services. Streamlining the authentication process will simplify future projects of which both systems are the subject. Because this project also covers authentication best practices and standards, myCSB and Entropy can be examined to ensure they incorporate them.

CHAPTER III: SYSTEMS ANALYSIS OVERVIEW

The implementation portion of this project is focused on software (ASP.NET websites in Visual Studio, working in tandem with SQL database). As such, several software process models were examined. These models can be applied to support the development of software.

3.1 *The Waterfall Model*

The waterfall model is perhaps the most well-known of the various system development approaches. This approach is highly plan-driven, falling on the "predictive" side of the Systems Development Life Cycle (SDLC) scale. In other words, the waterfall approach requires that each phase of development be entirely planned out—start to finish—prior to beginning the actual work. This process assumes the each phase of development can be tackled sequentially, with no need to revisit preceding stages. The waterfall model requires rigid planning, and may produce strong results on paper; nonetheless, due to human error and unanticipated problems, the "pure" waterfall model is ineffective. It is more practical to use a customized waterfall model that allows for more flexibility. (Satzinger, Jackson, Burd, 2010)

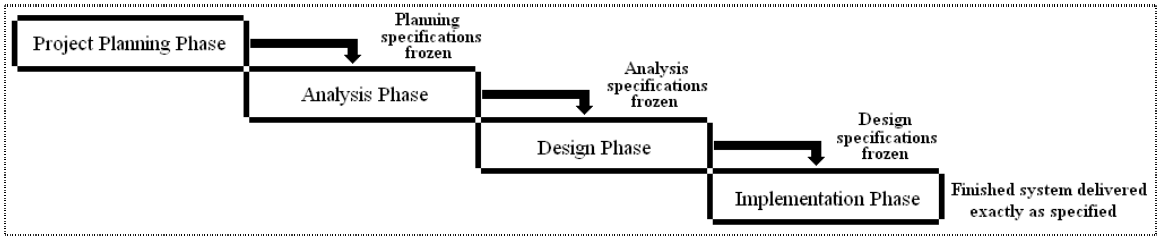


Figure 3 - Waterfall Model (Satzinger et al., 2010)

3.2 *Incremental Development*

The foundation of incremental development is customer involvement and feedback. This process attempts to approach projects intuitively, mirroring the way people naturally solve problems. (Sommerville, 2010) Incremental development involves creating an early initial prototype, or simplified working implementation. This early version normally contains the most critical functionalities. This way, it can be exposed to users and stakeholders, who provide imperative feedback. This is done repeatedly, with features being gradually added, modified and removed from the central infrastructure, until a suitable final product is accomplished. While fast and relatively cheap, the incremental development approach works best for smaller undertakings, including personal projects and e-commerce. (Sommerville, 2010)

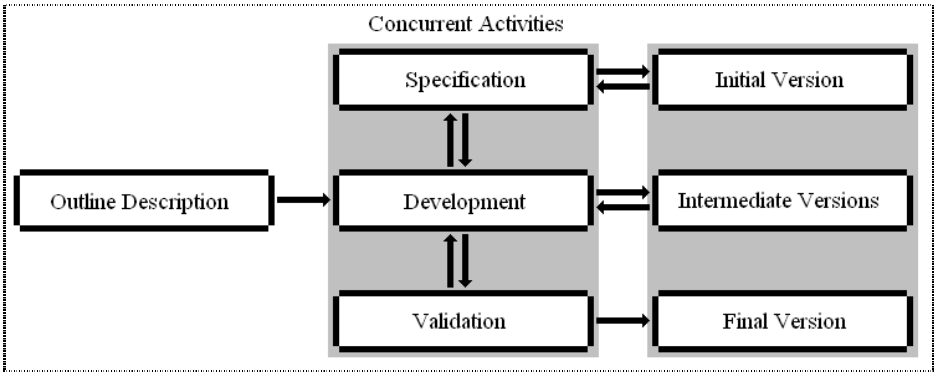


Figure 4 - Incremental Development (Sommerville, 2010)

3.3 *Spiral Model*

While the waterfall model falls on the rigid, "predictive" side of the SDLC scale, the spiral model sits on the "adaptive" end. This model does not approach a software project sequentially, but iteratively, which involves breaking large problems into smaller, more manageable components. Each section of the project may be done more than once—in fact, continually refining a result over multiple iterations is the advantage of an adaptive model. Graphically, this process resembles a spiral (hence the name), containing

the different phases and requirements of the project. There is an element of *risk* that is addressed in the spiral model. Aspects of the project that are deemed risky are closer to the center of the spiral, meaning they are tackled early on it the development process. (Satzinger et al., 2010) This approach can also allow for the construction of prototypes, which are useful for exhibiting progress to stakeholders and receiving important feedback.

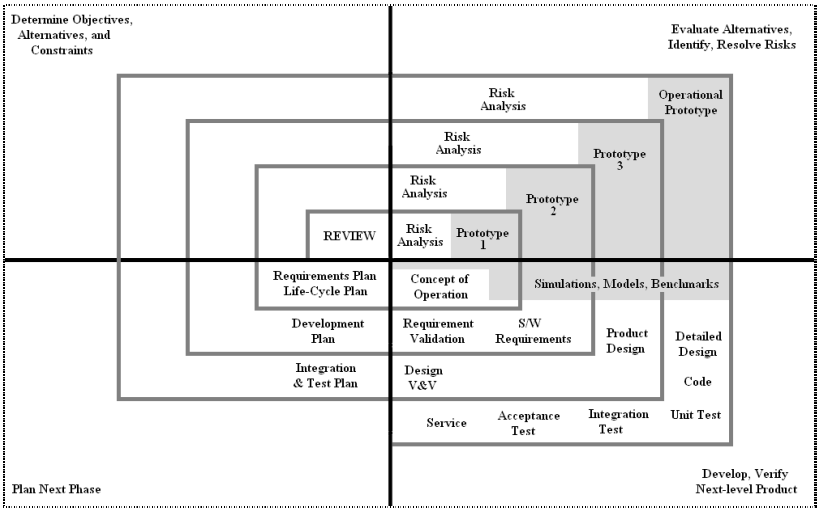


Figure 5 - Boehm's Spiral Model (Sommerville, 2010)

3.4 *Agile Development*

Agile methods lead to an accelerated development and deployment of software. This is because agile methods—using incremental development—place a strong focus on design and implementation, and incorporate other activities into these. Developers using agile methods tend to work in small teams that communicate quickly and informally. Design documentation is minimized or automatically generated, and the system or software goes through multiple versions. In this way, the developers can very quickly push the system or software out. Agile development doesn't work well for large projects, or projects that require extremely detailed design and specifications before

implementation. There are many agile methods. Some of the most well-known include Extreme Programming (XP), Scrum, Rapid Application Development (RAD), the Agile Unified Process (AUP), Adaptive Software Development (ASD), Crystal Clear, and the Dynamic Systems Development Method (DSDM). (Satzinger et al., 2010)

3.4.1 Extreme Programming

The foundation of XP is its four core values: communication, simplicity, feedback, and courage. Communication involves staying in close contact with those involved with the project, including users, stakeholders, and other project members. This is closely tied with feedback, wherein those same key users and stakeholders provide consistent, useful feedback regarding the system or software as they are presented with updated versions. Simplicity is as one would expect: avoiding overcomplicating aspects of the development process. Courage requires facing some of the harsh realities of project development—for instance, it may be better to abandon certain aspects of a project, to throw away bad code, or to admit that there is not enough allotted time. XP is well-known for two practices (among many): pair programming (in which two programmers work together in close proximity, regularly switching roles), and collective ownership (all team members "own" the project as one). (Satzinger et al., 2004)

3.4.2 Scrum

Scrum was developed as a highly responsive methodology, able to cope with many frequently changing priorities and unsure stakeholders that would normally overwhelm a project. This development methodology is very focused on team development, and revolves around a *product backlog*. The product backlog is a list of features, user functions, and technology that is actually owned by the primary client. He

or she is handed control of the requirements that would typically be managed by the project team. A *scrum master* is similar to a project manager, though Scrum-driven projects are self-organizing and loosely scheduled around *sprints*. A sprint is a pre-defined period of time (usually 30 days), during which certain goals or deliverables are outlined, and worked towards. During each sprint, the team meets for daily progress reports, and focuses strongly on their goals; the scrum master deals with other project management issues. (Satzinger et al., 2004)

3.4.3 Rapid Application Development

The foundation of Rapid Application Development (RAD) is the continuous creation of prototypes. RAD focuses less on the planning aspect of software development, and places emphasis on actually developing the software. Early on in the project, requirements for the software are defined, and user feedback is gathered. As the software is built and changes are required, "plans" are established on the fly and quickly integrated into software. Throughout the development, users may continue to provide feedback, and many prototypes may be built and tested before a final version is reached. At this point, a compressed version of the usual implementation phase activities commences; this includes data conversion, the switch to the new system, user training, and so on.

3.5 *Selected Approach*

Each of the previously detailed development methodologies were carefully considered for this project. The waterfall approach is far too rigid; more importantly, it is intended for much larger projects that need a great deal of planning. The advantage of the spiral model is that it is adaptive. However, it is also better applied to larger, more time-

consuming projects. Because this project involves a single developer with a small supporting team, a fast-moving, dynamic methodology is preferred—an agile development methodology. Of the many available methods, Scrum, XP and RAD showed the most promise. While they had constructive elements, Scrum and XP were too team-oriented and better suited to projects with a larger time frame.

For this project, the most appropriate development methodology was RAD, with elements of incremental development. This project had to be finished over a short period of time (only a few months). During this time, frequent user feedback was needed, as well as many prototypes. The prototypes themselves changed regularly as project requirements shifted. The earliest prototypes, for instance, consisted of a single application that only allowed users to log in. Over time, the prototype grew to include more functions: changing passwords, retrieving passwords, and managing user information. By the end of the project, the prototype included a fleshed-out version of the central authentication application, as well as two small-scale Entropy and myCSB test application with which it communicated.

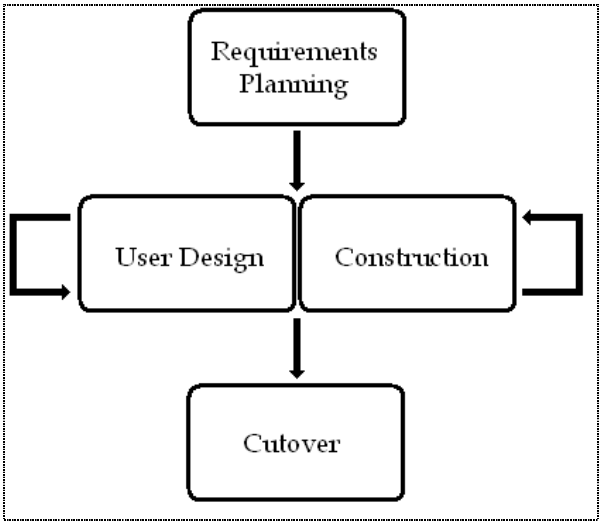


Figure 6 - Rapid Application Development

CHAPTER IV: IMPLEMENTATION METHODS INVESTIGATED

4.1 *Database-centric Options*

4.1.1 Option A: Fully Merged Databases

This project involves two separate databases. Each database—designated CSBLive2007 and Entropy2012—share the same SQL Server instance. A considered solution for authentication centralization is a complete merge of those databases into a single entity that serves the needs of both myCSB and Entropy services.

However, this approach—while not impossible—is highly impracticable due to the sheer sizes of the respective databases. CSBLive2007 contains >200 tables, while Entropy contains around 60 tables. To amalgamate them into a single database would result in a disorganized, unwieldy jumble of tables, stored procedures, and diagrams. Because students work on both systems, it is practical to keep the databases smaller. Furthermore, authentication is only a small part of these systems. Most of these tables deal with other imperative system functions, such as the Adaptive Grading/Learning System (AGLS).

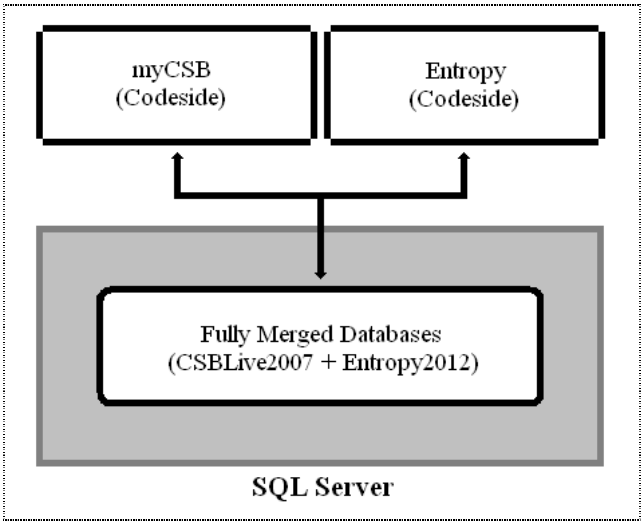


Figure 7 - Fully Merged Databases Architecture

4.1.2 Option B: Consolidated Authentication Database

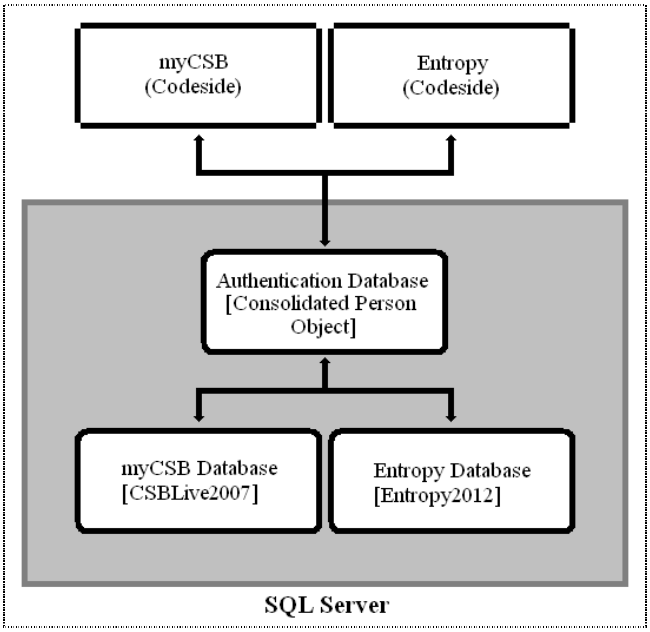


Figure 8 - Consolidated Authentication Database Architecture

The consolidated authentication database takes a different approach than the full database merge. The Entropy2012 and CSBLive2007 databases are left separated, so as not to interfere with their diverse normal functions, such as existing stored procedures and views. To achieve centralized authentication, each database is instead combed for any tables and stored procedures related to authentication. Most notably, anything related to a *Person* object, related roles, and other authentication necessities such as logging changing and retrieving passwords.

With the authentication-related aspects of the databases identified, a new database is created that is dedicated entirely to identity verification. All authentication aspects are placed into this new database, which interacts with myCSB and Entropy services. Structurally speaking, the auth-database is placed "on top" of the original databases, since

users must interact with it first when logging in before using with the others. On the client side, the user will interact with an entirely new application that is dedicated to authentication.

4.2 *Authentication-centric Options*

4.2.1 Option A: ASP.NET Windows-based Authentication Methods

Windows-based Authentication

The main advantage of ASP.NET's Windows-based authentication is that it uses Windows user accounts and directory rights to grant access to restricted pages. This type of authentication is configured via the Internet Information Services (IIS) console; when a user visits a page that requires identity verification, a login dialog box appears. (Boehm, 2008) An advantage of this is that no custom code need be written for authentication.

UNCW users are assigned their own Windows accounts that can be used to access various online university services. As such, it is possible to configure Entropy and myCSB to use Windows-based authentication, allowing users to use minimal credentials. Certain aspects of the current custom authentication method would be removed, configuration files would be modified to support this type of authentication, and the ASP.NET applications would rely on the primary Active Directory on the network. Ultimately, however, Entropy and myCSB are custom applications, and require autonomy from the university network. Some of the users of these applications (outside users, such as volunteers) do not have university privileges, and would have no way to use the sites if the they relied on University-assigned emails and IDs. For these reasons, Windows-based authentication cannot be implemented.

Windows Live ID Authentication

Previously called *Passport authentication*, this technique performs identity verification using a centralized account management service owned by Microsoft, called Windows Live ID. In order to use this type of authentication, the Windows Live ID Web Authentication software development kit (SDK) must be downloaded, and the website in question must be registered with Microsoft in order to obtain an application ID. This method allows a user to own a single user account that can access any website using Windows Live ID authentication. (Boehm, 2008)

Windows Live ID Web authentication is less popular than the other available ASP.NET authentication options. Because of this, it is uncertain how long this alternative will remain available. Additionally, there are other more well-known and customizable single sign-on solutions for the Web, such as OpenID and OAuth.

4.2.2 Option B: OpenID Integration

The Web is site-centric, frequently requiring users to create a username and password; over time, the number of passwords a single user owns multiplies. The average Web user inputs about eight passwords a day. Furthermore, they have approximately twenty-five accounts, each requiring a password. (Florencio & Herley, 2007) Remembering so many passwords can become taxing, and leads to users developing habits and behaviors that defeat the purpose of security rooted in knowledge factors ("something known"). OpenID is designed as a single sign-on authentication technology (giving users the means to sign in to many different services using only a single set of credentials), and attempts to address this problem, sometimes referred to as "password fatigue."

In order to realize single sign-on authentication, a Web SSO like OpenID must define the roles of those entities involved in the identity verification procedure. There are two primary roles to consider: that of the *identity provider* (IdP), and that of the *relying party* (RP). An IdP takes on the responsibility of collecting user information, as well as executing the actual authentication process. The RP is the site or service that wishes to ensure that the credentials passed by the end user are legitimate.

In this scenario, myCSB and Entropy could make use of OpenID's authentication process. Not only this, the custom authentication already in place need not be removed; OpenID can act as an *additional* sign-in alternative for users. Entropy and myCSB would act as relying parties, and send the user to a trusted identity provider, such as Google. The IdP performs the authentication, and returns the result to the RPs; if there is a successful match, it is left to the RPs to make authorization decisions (i.e., determining user roles).

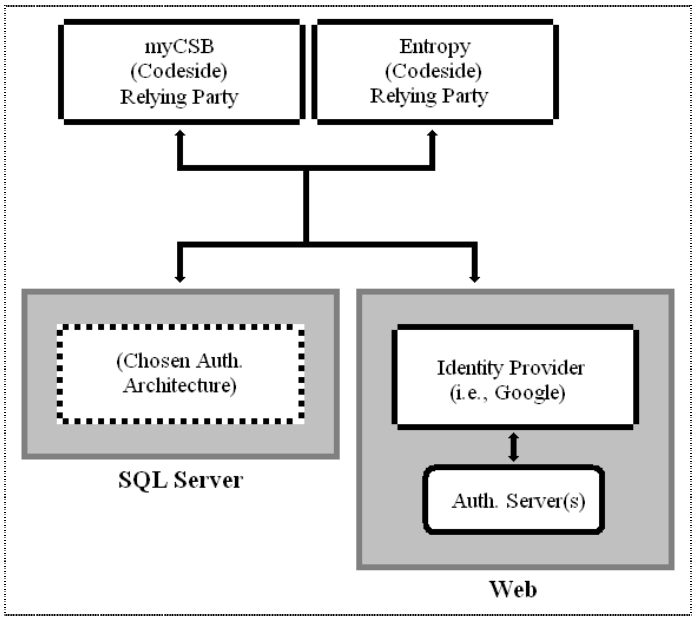


Figure 9 - Potential OpenID Integration Scheme

4.3 *Authentication Best Practices & Standards*

Multi-Factor/Strong Authentication

Most people will inevitably find themselves in a situation that necessitates authentication. For instance, many people have accounts for various online venues, such as e-banking, e-shopping, and social networking sites. Each of these online services requires the user to provide his or her identity consistently and accurately. The service, via diverse techniques and methods currently available, authenticates that identity.

The previous example illustrates a situation in which authentication takes place by means of a user-provided password. This is only one of many authentication avenues, and falls under the category of a *factor*. There are three primary authentication factors.

Ownership factors refer to something the user *has.* (Federal Financial Institutions Examination Council, 2008) In this category, a physical device of some kind is required for authentication. A common example of "something the user has" is an identification card. A bank card, for instance, is required for the user to establish his or her identity and access their account at an Automatic Transfer Machine (ATM). Using a personal, portable physical device such as this makes it simple to supply authentication. However, there is an ever-present risk of the device breaking or experiencing wear due to extended use, and of losing the device or allowing it to fall into the wrong hands. There is also the possibility of the device being duplicated. (Sutcu, Sencar, & Memon, 2005)

Knowledge factors refer to something the user *knows*. (Federal Financial Institutions Examination Council, 2008) Authentication in this category is achieved by means of something unique remembered by the user such as a password, pass-phrase, or Personal Identification Number (PIN) (the "authenticator" (Committee on National Security Systems, 2010). The prior example, in which a user provides a password to the login system of an online service, is a frequently encountered situation where a

knowledge factor is applied. Relying on "something the user knows" is advantageous in that ideally, *only* the appropriate user knows the relevant information required for authentication. In practice, unfortunately, users display a tendency to intentionally degrade the security of this practice in favor of expediency. (Sun, Popisil, Muslukhov, Dindar, Hawkey, & Beznosov, 2011)

Inherence factors refer to something the user *is* (Federal Financial Institutions Examination Council, 2008) or *does*. More specifically, this category falls within the field of *biometrics*, in which authentication is a principal concern. In circumstances where biometric systems are used, authentication may be achieved by supplying a fingerprint or retinal pattern. Other methods of recognition include (but are not limited to) iris, voice, facial, signature, and gait. The field of biometrics is still expanding, and various inherence factors are being investigated as feasible authentication options. As of now, biometric authentication is normally employed in corporate environments (Sutcu et al., 2005), though personal biometric systems are becoming more widespread.

When attempting to produce a highly secure system authentication scheme, it is generally insufficient to employ only a single technique (known as *single-factor authentication*). Each authentication factor provides a measure of security, but also comes with its own set of shortcomings. The reality is that a determined, resourceful attacker attempting to bypass any of these techniques can find a way to do so.

In order to minimize the possibility of an attacker successfully circumventing the aforementioned methods, it has become common practice to use two or more of them in conjunction. This is known as *two-factor authentication* (de Borde), or *multi-factor authentication*. In this case, a user must provide at least two proofs of identity to be

authenticated. A frequently encountered example of this is identity verification at an ATM. When attempting to access his or her account, the user must first provide a physical device—the identification card. Following this, the user must supply a PIN. Alone, these techniques afford a relatively low level of security, particularly the PIN, which is considered the "weakest link" in this particular process. (Sutcu et al., 2005) When used together, a higher level of security is attained; consequently, this is the preferable practice, and adheres to the definition of *strong authentication* as determined by the U.S. Government's National Information Assurance Glossary. Strong authentication is a "layered authentication approach relying on two or more authenticators to establish the identity of an originator or receiver of information." (Committee on National Security Systems, 2010)

Encryption & Hashing

An important aspect of authentication is the format in which passwords are stored. The simplest way to store this kind of personal information is to keep it in a database entirely in plaintext (databases are not the only technique for storage—directory services, configuration files or other methods can also be used). For obvious reasons, however, this is a significant security vulnerability. Plaintext passwords are far more susceptible to infringement, which puts users in danger. This is principally unacceptable for businesses that handle a great deal of personal and private information. Encryption and hashing techniques offer a realistic counter to this weakness.

Encryption may be defined as "the coding or scrambling of data so that humans cannot read them," (Hoffer, Prescott, & Topi, 2009) or "the process of converting readable data into unreadable characters to prevent unauthorized access." (Shelly et al.,

2007) Encryption techniques can successfully translate plaintext into a string of gibberish (*ciphertext*) that is largely useless to an attacker. This technique is not foolproof, particularly if the password being converted is weak or simple to begin with. More importantly, proper encryption algorithms are "two-way," meaning the encrypted password can be *de*crypted using some sort of key. This in itself is a problem; if an attacker acquires the key, the encryption efforts are rendered moot, and all obfuscated information is at their fingertips. Using a one-way method (such as a *cryptographic hashing function*) may therefore be preferred. It normally takes an unreasonable amount of time to "crack" an encrypted password—so long, in fact, that an attacker may as well not bother. Additionally, using a one-way method ensures that the plaintext versions of stored private data are also unreadable to the application owner(s). There is typically *no reason* for passwords to be seen by anybody, even non-malicious users.

There are numerous encryption algorithms available. One of the earliest algorithms is the Data Encryption Standard (DES), originally developed in 1977 by IBM and approved by the National Security Agency (NSA). This specification was eventually outmoded by Advanced Encryption Standard (AES) in 1998; it has since become a Federal government standard. Another early encryption algorithm is RSA, which was published in 1977. This method uses public-key cryptography—two keys, one public (known to all) and one private, are used to encrypt and decrypt data.

While encryption algorithms are commonly employed to keep data secure, cryptographic hashing functions are preferred by many due to their one-way utility. Perhaps the most well-known hashing function in use today is MD5, which was first published in 1992. Unfortunately, MD5 and its predecessors have been thoroughly

deconstructed and numerous flaws have been revealed. In response, the United States Computer Emergency Readiness Team (US-CERT) has recommended adopting the NSA-designed SHA-2 hashing function. Hashing function effectiveness can be augmented by applying a *salt*, which consists of "random" bits added to an obfuscated password or passphrase. Salting private data is an effective way to counteract dictionary attacks and rainbow tables.

UNCW's Entropy and myCSB applications use RC4 encryption when performing authentication. Produced in 1987 by Ron Rivest, RC4 makes use of a *stream cipher*. A pseudo-random string of bits—normally between 40 and 128 (a key length of >80 bit is considered "strong")—is generated using a key-scheduling algorithm. Via bitwise exclusive-or function, the pseudo-random bits are combined with the plaintext (such as an inputted password) to produce a *keystream.* RC4 is quite fast and uncomplicated, and can be used with Secure Sockets Layer (SSL). However, this technique has several vulnerabilities that must be understood and addressed prior to its appliance. The major vulnerabilities involve observing biases in the keystream; there are ways to mitigate these problems, such as using hashed session keys. (Wash, 2001) Also, in response to its weaknesses, RC4 variants (such as RC4+, RC4A, and VMPC) have been developed. For more information of RC4's role in the authentication of myCSB and Entropy systems, please refer to chapter 2.3, labeled *Systems Specifications*.

Cookies

There are many techniques to perform Web-based identity verification, and not all of them necessitate a database. Nevertheless, the most prevalent authentication techniques make use of small files called *cookies* that interact with browsers, hard disks

and databases. Nowadays, cookies have entered the public consciousness. Even non-technical Web users understand several things about authentication: that they need a username and a password, and that cookies are somehow involved.

Cookies follow a general process. When a user visits the log in page, they supply a username and password. Authentication takes place, and a match is found. A cookie is now created that contains several pieces of information, including confirmation that authentication was successful. The browser sends the newly created cookie back to the server, which determines that the user has been authenticated by comparing a unique number in the cookie ("token") with a stored value associated with the user's credentials, and forwards the restricted page to the browser. (Boehm, 2008) From here, as long as the session token persists, the user can continue browsing restricted pages.

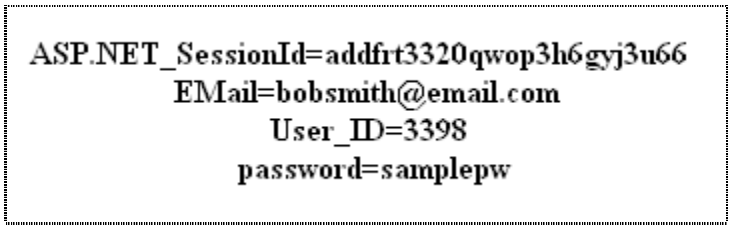


Figure 10 - Sample ASP.NET Session Cookie (Boehm, 2008)

A cookie can be designed for different tasks, such as session management (the user stays logged in as long as they stay on the website) and persistent login (the user stays logged in between visits). This means the attributes and values included in the cookie—while similar—won't always be the same.

Despite their widespread use in the authentication process, cookies are not completely secure. Malicious users can intercept cookies and use them to their advantage via network eavesdropping, DNS cache poisoning, and various cross-site scripting (XSS) techniques, to name a few. The best lines of defense are to use secure lines of

communication (HTTPS), and to make certain that any sensitive data in the cookie is unreadable by way of hashing (with salt) or encryption.

Stored Procedures

When an application makes use of a database, it recurrently performs *queries* that affect the stored data in some way. The term *CRUD* (*Create*, *Read*, *Update*, *Delete*) encapsulates several common queries. A *stored procedure* (or *sproc* for short) is a module of code in the database that implements application logic. (Hoffer et al., 2009) Authentication can be achieved by building stored procedures specifically for that purpose; other authentication-related functions, such as password changes and retrieval, can also be built into sprocs.

What are the benefits of sprocs? For one, sprocs reduce the amount of client-side code, and move application logic to the database server. This also has a positive affect on network traffic, since the server bears more processing than the client. Most importantly, perhaps, is the security aspect of sprocs: the procedure itself is accessed, instead of potentially-sensitive stored data (such as passwords). Furthermore, sprocs can be called repeatedly by multiple applications; this reduces application code redundancy, and improves data integrity. (Hoffer et al., 2009)

Stored procedures come with drawbacks, as well. For one, they are tightly integrated with the database in which they a written. Any changes made to tables in the database can require immediate modification of the procedures. In addition, the client must be preloaded with the proper calls to the sprocs, and modification of the procedure can require application code changes, as well. As the number of users online increases, sproc performance may begin to diminish.

There are other methodologies for dealing with application logic that interacts with a database. *Ad hoc SQL* is an alternative to stored procedures, and involves placing SQL statements directly into code. However, this is widely viewed as extremely insecure and bad practice in general, as the application is left vulnerable to SQL injection attacks. It is possible to safely place database statements into code by using *parameterized SQL*. By virtue of its design, parameterized SQL is protected from injection attacks. Nonetheless, placing any database-related statements codeside is disputable. If used, parameterized SQL would likely work best in smaller systems, while larger systems are better off employing stored procedures.

Ultimately, no matter the methodology used to encapsulate and perform sections of application logic, there is one rule that stands: the database itself must be thoroughly contemplated and well-designed from the very beginning. In a relational database, this includes implementing integrity constraints and observance to normalization rules. If the database is poorly conceived, then any queries—regardless of style—will perform with substandard results, as well.

Additional Security Aspects

Many security-related characteristics of the authentication process are never actually seen by the user, as they occur behind-the-scenes. In a properly-designed authentication system, there are additional security aspects that should be considered; these are features with which the users may interact, and are intended to mitigate risky user behavior. Naturally, these features will clash with user convenience, and a careful balance must be struck.

A digital identity has its own "life cycle"—it is created, exists for a period of time, and is ultimately terminated. For security purposes, it is preferable to deliberately appoint an identity's password with a limited duration. When the *maximum duration* is impending, the user should be notified and prompted to safely change their password. Related to maximum duration is *minimum duration*—a password cannot be changed until after a certain period of time has passed. This stops users from changing their password repeatedly; this often happens when users want to revert to their original password, and there is a *password history* policy in place that disallows the same password to be set within *n* password-change iterations. (Microsoft, 2005) In a "high security" system, the maximum duration is around 42 days, the minimum duration is 1 day, and the password history stores 24 previous passwords. Naturally, these numbers depend on the environment in question. (Chinta, Danseglio, & Resnick, 2004)

Another password-related policy is that of the *account lockout*. This is applied to stop malicious users from illicitly accessing accounts. When the attacker unsuccessfully attempts to login, an *observation window* is opened that counts subsequent failed login attempts. After *n* failed attempts (the *account lockout threshold*), the account is locked and cannot be accessed at all for a set amount of time (the *lockout duration*). For a "high security" system, a standard account lockout threshold is 10 tries, with an observation window of 30 minutes, and infinite lockout duration (to be unlocked at the behest of an administrator). (Chinta et al., 2004) An account lockout security policy must be employed with care; it is possible that legitimate users may be locked out accidentally.

In most cases, during the authentication process, a *session* is created for a legitimate user. The session is fundamentally a unique cookie that contains information

about the user. Typically, the session will have a *timeout* property, equal to *n* minutes. When a user allows their session to run idle (no activity or refreshing), the timeout value eventually drops to zero, and the user receives a session timeout that prompts them to resubmit their credentials. This is done to ensure that—in the event that the legitimate user simply walked away or forgot about their session—no malicious users can hijack the session. The ASP.NET timeout value is set to 20 minutes by default; as with other policies, the ideal timeout value depends on the environment. It is important to consider that an active user session utilizes memory, and shouldn't remain idle for too long; on the other hand, in order to maintain an acceptable degree of user convenience, it shouldn't be too short, either.

There are authentication-related functions that require the user to prove their identity again, such as changing a password. *Security questions* can be implemented in lieu of passwords as proof of identity. These questions are typically set up with the account—the user may be allowed to choose them, or they may be predefined. It is vital that the chosen security questions encourage answers that are highly unique. Otherwise, the security questions may be far easier for a malicious user to take advantage of than a password. If security questions are used, it is best to not allow users to pick their own questions, and to couple them with another form of identity verification (such as retyping the regular password).

A recent security challenge authentication systems face is that of *internet bots*, which perform automated tasks at a rate far beyond that of any person. Bots may attempt to illegitimately access a system by means of automation in order to perform a repetitive task (such as sending spam email). (Kim, Jeong, Kim, & So, 2010) Consequently, it is

important to consider implementing bot detection techniques into an authentication system. Once such technique is the employment of CAPTCHAs (*Completely Automated Public Turing tests to tell Computers and Humans Apart*), which—while not foolproof—can effectively impede bots. Because it incorporates the parsing of idiosyncratic imagery, CAPTCHA design necessitates human intervention. (Bursztein, Martin, & Mitchell, 2011)

User Convenience Aspects

When designing an authentication system, a great deal of focus is often placed on making it as secure as possible. While effectual security is vital, considerations must be made regarding the users of the system. It has been shown time and time again that, when interacting with highly secure systems that require dedication and effort, users will develop security-degrading habits in favor of convenience. (Sun, et al., 2011) User convenience, then, is a metric that must be taken into account.

It is ordinary for users to forget their credentials. Because of this, users may develop security-degrading habits, such as writing down or sharing their credentials. It is essential that applications include a way to safely retrieve forgotten passwords. Typically, this is done by adding a "forgot password" link at the login page. The user is prompted to enter a valid email address associated with the account, and the forgotten password can then be sent directly to the email address. This arrangement makes it reasonably simple for users to retrieve their lost password. However, this system does *not* adhere to security best practices. Allowing a password to be sent in plaintext via email suggests that either the passwords are encrypted and then decrypted, or that they are stored in plaintext from the start. It is preferable to instead use a *password reset* system, in which the user is sent

a single-use tokenized URL. When clicked, the URL allows the user to specify an entirely new password. (Hunt, 2011)

When visiting a website, users are often given the choice to allow their login information to be remembered. The login form normally includes a checkbox labeled "remember me." When selected, the user is able to leave the site entirely, and remain logged in upon their return. This is known as *persistent login*, and has become a common feature of websites with accounts. Persistent login works by issuing a login cookie, in addition to the standard session management cookie (or by specifying the session cookie as persistent itself). The unique login cookie is stored away; when the user returns to the site, the stored information is compared to the login cookie stored in their browser. If there is a match, the user is issued another login cookie, in order to maintain persistent login. Because it relies wholly on cookies, a persistent login mechanism must be cautiously designed. Cookies can be intercepted and used maliciously, and therefore are not entirely safe. (There are proposed designs to mitigate this problem: see (Jaspan, 2007))

A fundamental security policy is the required changing of passwords when a maximum duration is arrived at. However, users may wish to change their password voluntarily for a variety of reasons. For instance, if a password is compromised, the valid user has reason to change it without delay. After logging in successfully and selecting the "change password" option, the user will be required to 1) answer the security questions they configured when first creating the account, and 2) retype their previous password. The need to voluntarily change a password may clash with the minimum duration policy for passwords, requiring administrator involvement.

ASP.NET Authentication Template

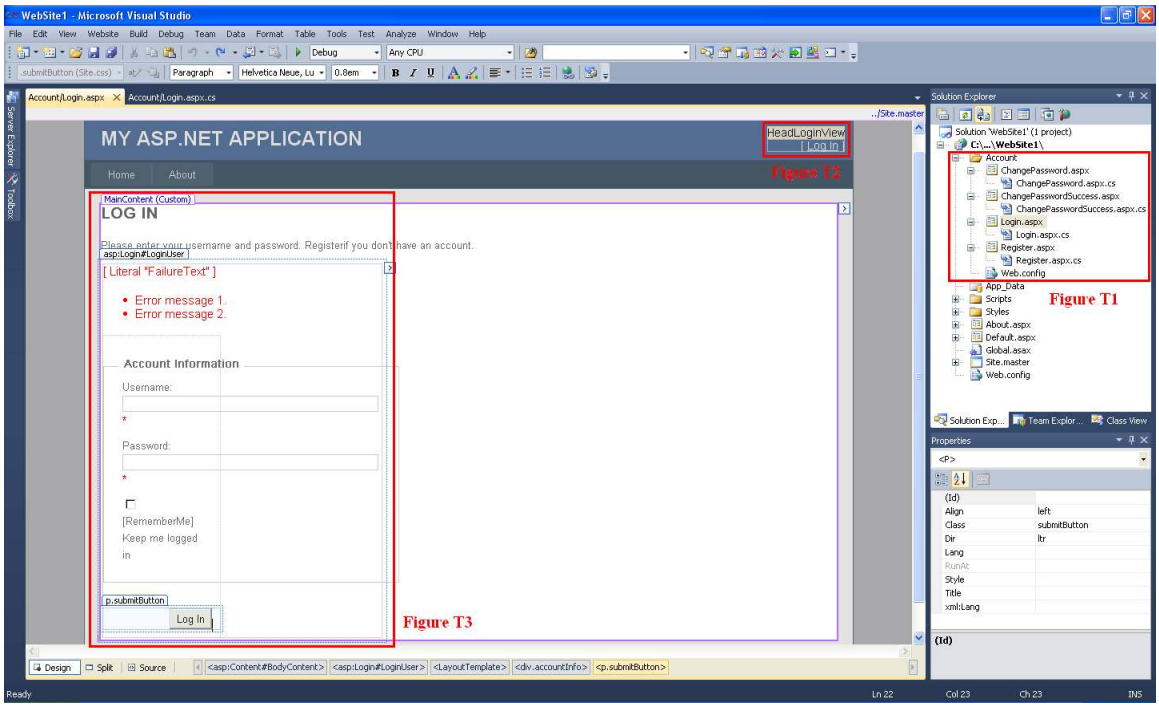


Figure 11 - ASP.NET Authentication Template in VS 2010

Figure 11 - T1

When a standard website template is picked in Visual Studio 2010, the application is populated with several aspx pages. Pages related to authentication are automatically placed in the *Account* folder. Among these are: *Login.aspx*, *Register.aspx*, *ChangePassword.aspx*, and *ChangePasswordSuccess.aspx*. Each of these pages has underlying code (in this example, the code is C#; Visual Basic is also available). The template only fills in the barebones of what's needed for fully functional authentication; it is up to the developer to further modify what's given to suit his or her needs.

Figure 11 - T2

Figure T2 contains a link to the log in page. This link is embedded into the master template of the website (Site.master).

Figure 11 - T3

Figure T3 shows the automatically-generated log in form. The entire form is structured as a single entity, designated *LoginUser*, and contains a username box, password box, “remember me” checkbox, and a log in button. In addition, error-checking is included in the template.

CHAPTER V: IMPLEMENTATION METHOD SELECTED

The following constraints were established to select a method:

- Must allow the databases to remain small
- Must avoid tying into the UNCW domain
- May be outsourced, but only if implemented as a supplement to the existing Forms authentication

After consideration of the various approaches that could be used to centralize the authentication of the systems, the method selected is the consolidated database approach. With the exception of OpenID, the other methods contain aspects that ultimately disqualify their employment. The consolidated database approach has no attachment to the UNCW domain, and allows the databases to remain relatively small, while installing a third auth-centric database.

With the consolidated database in mind, centralized authentication was approached by creating an application dedicated to authentication functions and user profile management. This application stands apart from the primary myCSB and Entropy applications; however, it communicates with them via redirects (outbound links) and mutual cookies. In a sense, there are limitations to the project that are directly related to the environment in which the systems exist. All of the applications must be present on the same server, so that a shared cookie may be employed. Further, the applications use the same security certificate, which provides a level of security.

In addition, OpenID was considered as a possibility. This technique adheres to the third metric: it may be applied as an additional authentication option. However, based on several survey results, it was decided that OpenID would not actually be implemented,

nor would it be pilot tested. However, a goal of this project is to examine it in order to understand how it works. For more information on the authentication surveys and implementation strategies, please refer to chapter 7, labeled *Alternative Authentication Analysis*.

CHAPTER VI: CENTRALIZED AUTHENTICATION DEVELOPMENT

6.1 *Design Materials*

After concluding that the centralized authentication database was the best option, the first step in the project process was to begin the design phase. This involved creating a preliminary plan for the design of the database and changes to the code base, as well as the initiation of several common design techniques: use case diagrams, CRUD diagrams, and use case descriptions.

At the start, it was important to acquire access to the relevant systems: the databases behind Entropy and myCSB, as well as the current code. Using these as references, the objective was to find all instances of the database that related to the Person object and the authentication procedure. These pieces included tables and stored procedures that are very closely tied with the code. This allowed for more accurate diagrams to be designed. For example, **Figure 12** shows an early plan for the tables and fields in the authentication database.

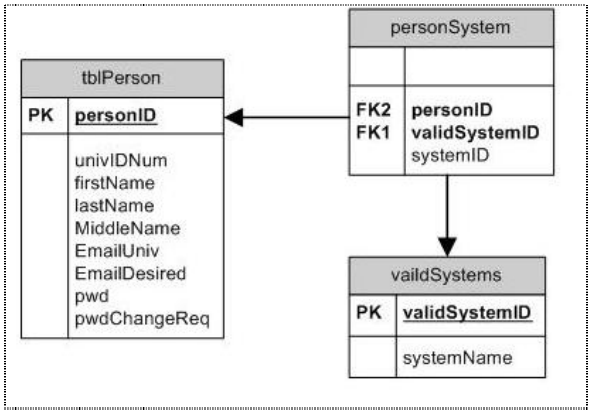


Figure 12 - Consolidated Auth. Database Tables

The simplest diagrams to be created were the CRUD diagram (which required picking out relevant system functions, and matching them up with database entities), and the use case diagram (which showed the users of the system, and their allowed functions). The more time-consuming materials were the use case descriptions, which went into a great deal of detail about essential system functions. (A sample use case description for the login function is shown in Table 1; the other use cases related to this project are located in Appendix B—Use Case Descriptions)

Table 1 - Sample Use Case Description for Logging In

<table>
<tr><td>Use Case Name:</td><td>Login</td></tr>
<tr><td>Scenario:</td><td>Users login to their account</td></tr>
<tr><td>Triggering Event:</td><td>A user clicks the login button.</td></tr>
<tr><td>Brief Description:</td><td>A user visits the landing page for Entropy/CSB Portal. They supply a valid username/password combination and click the login button to access their account. Usernames can include their University ID#, their University email address, or their desired email address.</td></tr>
<tr><td>Actors:</td><td>All valid users</td></tr>
<tr><td>Related Use Cases:</td><td>readPerson, readPersonRole</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>User must have an existing account with username/password combination.
The user’s password must not be expired.</td></tr>
<tr><td>Postconditions:</td><td>The user must be redirected to their account main page.
A temporary session must be created for the user.</td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table>
<tr><td>1. User visits the landing page of the website.
2. User enters their username (univ ID, univ email, or desired email) and password and clicks the login button.

3. The user is redirected to their homepage.</td><td>2.1 The system receives the username / password combination and attempts authentication.
2.2 The SP checks that the username exists and returns the personID.
2.3 The system compares the encrypted values.
2.4 The user’s role is determined in order to properly display their options.</td></tr>
</table></td></tr>
</table>

Exception Conditions:	1. If the user was previously logged in, they may still be logged in due to persistent login (this may apply only to CSB, not Entropy). 2. If the username/password combination is incorrect, the user will be denied access and returned to the landing page.

6.2 *Centralized Authentication Database: Development*

Testing

Using the design materials as reference, the next step was to finally delve into the code and the databases and begin working out implementation strategies. This required building a functional test environment that mimicked or mirrored the current functions of Entropy and myCSB. A personal database and workspace was made available for testing purposes. The test environment could be approached from 1 of 3 tacks: 1) creating a small-scale replica of the systems with authentication-limited code and databases, 2) using the full code-base with limited databases, or 3) setting up the entire site with fully transferred databases. The first option was deemed the most appropriate for this project.

After reviewing the code for database calls, three small-scale databases were built: authTestDB, entropyTestDB, and csbTestDB. Each of these databases contained a Person table with similar columns. At the start, user roles were disabled, since they were not needed for authentication. Similar stored procedures were also created in each database, including procedures for updating users and changing passwords. Sample users were added into the databases; however, the password field was designated as *varbinary* data type, and data could not be manually inputted. This problem was tackled using a change password workaround (more on this below).

Early on, it was decided that the personID attribute of all users matching across all databases was advantageous. To make this happen, the plan was to reset the *Identity*

Seed property of the personID in each Person table to a value greater than the current number of users (such as 25000). After doing this, repopulating the databases should match up the identifiers and simplify the manipulation of user information in the databases. For the csbTestDB and entropyTestDB databases, the personID was deliberately not specified as an identifier. This way, users could be manually added with IDs that matched those in the auth-database. A means to migrate already-existing personIDs to new values also needed to be developed.

At this stage, it was necessary to begin building the code-side of the test environment in Visual Studio using ASP with underlying Visual Basic. The original code was reviewed, and a small-scale replica of the Entropy system was put together, piece by piece. The replica served as the first of many prototypes that would evolve into a functioning centralized authentication system. This portion of the development required finding implementation strategies "on the fly"—looking at the code, and working out modifications to centralize authentication functions.

The opening goal was to get simple authentication working. This was completed by creating a new connection string to the authentication database. The easiest way to do this was to add an object on the design side—such as a listbox—and create the connection via the wizard. The connection was then instantiated in the *sharedItems.vb* class, with a protection level of "shared" to allow any class to access it. Because users did not have passwords, the RC4 encryption portion of the code was disabled, permitting authentication using only a valid username. From here, each user could be properly assigned an encrypted password using the change password function, which inserts encrypted passwords into the auth-database—the other databases no longer need to store

passwords. Each user had valid credentials, and encryption could now be re-added into the code.

With logging in and password changing finished, password retrieval needed to be added. To test retrieval, the page needed to be pointed to the authentication database, and the admin email needed to be changed to an accessible address. Stored procedures were used to retrieve the password, and the email was sent using the SmtpClient class (a subset of System.Net.Mail).

The next objective was ensuring that the system could retrieve and manipulate each user's ID(s), for both Entropy and myCSB. More importantly, the design needed to allow for the addition of future systems, such as Advising and Events management. The database table *personSystem* contained the needed values: the systemID, and the personID for that system. A stored procedure designated *validSystemsGet* was added to the auth-database, and two Integer fields (*csbSys* and *entSys*) were added to the *person.vb* class in code. Each of these field values could be assigned by using the stored procedure to read the data into a temporary hashtable, and setting the variables equal to the hashtable values. To find out if the user had an existing primary key in one of the systems, the *entSys* and *csbSys* values could be retrieved from their cookie (designated *u*) and checked for a value greater than 0.

```
Dim u As New person
If HttpContext.Current.Request.Cookies("csbInfo") Is Nothing Then
        'timeout
Else
        retrieveCookie(u)
        If u.csbSys > 0 Then
                'The user has an existing csb primary key
        End If
End If
```

Figure 13 - Checking the csbSys Value

With the authentication functioning and the user system values handled, it was necessary to begin undertaking several important system functions: the creation of new users, the updating of existing users, and the deletion of users. These critical operations needed to be modified to duplicate across multiple databases. For example, an Entropy user may update their profile; parts of their profile—such as names and desired emails—must also be updated in the auth-database.

The first of these functions to be tackled was the insertion of new users. It was found that the original stored procedure—*uspPersonUpdate*—that performed inserts also updated existing users by first checking for the user's personID in the database. If the ID exists, the user's inputted information is changed as specified. This simplified the design process. In order to allow new users to sign up and insert their information into the proper database, a query-string designated *sys* was added to the default page. The *sys* string was added to the URL of the sign-up page. The possible values of the string (such as 1 or 2) were used as Select Case Statement choices, and through them, the user's data could be correctly added to the databases. A new database connection string was added to the *sharedItems.vb* class in order to test adding the same data across multiple locations.

```
'Designating the sys string, which runs when the New User button is clicked.
Response.Redirect("~/user/profile.aspx?sys=" u.source)
```

```
'In the profile.aspx page—instantiating the sys string.
If Not Request.QueryString("sys") Is Nothing Then 'If querystring "sys" exists…
        ViewState("sys") = Request.QueryString("sys")
End If
```

```
'The user submits changes, and the update goes to the appropriate databases.
Select Case ViewState("sys")
        Case 1
                'Code to insert into Entropy
        Case 2
                'Code to insert into CSB
End Select
```

Figure 14 - Using *sys* Querystring to Make Insert Decisions

Inserting new users also raised several scenarios that needed to be addressed. These scenarios are as follows: 1) the user is completely new, and has no existing accounts with either system, 2) the user has an account with both systems, or 3) the user has an account with only one of the systems. To address these situations, the sign-up page was modified to first ask for the user's university ID number, and their university email address. The system first checks the auth-database for a match. If no match is found, they are permitted to sign up for a new account with the site, including a password. However, if a match is found, the system asks the user to confirm the retrieved identity (see **Figure 15**). When they do, the system first makes sure they don't have an account already with the site for which they are signing up. If so, they are redirected to log in; if not, process of elimination dictates that they have an account with the other system. Their account information is retrieved from that location and duplicated, thereby creating their account without the need to fill out a form and specify a new password (see **Figure 16**).

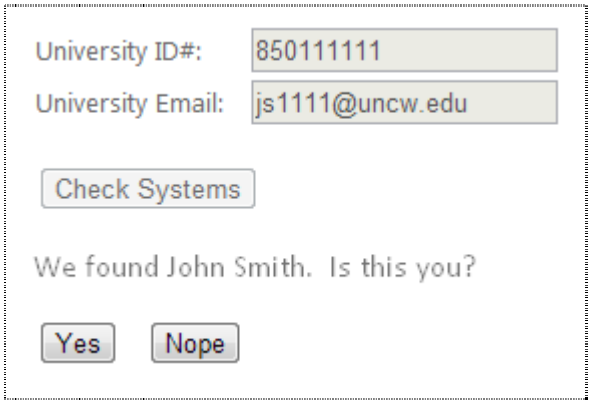


Figure 15 – Checking for a User in the Systems

Additionally, inserting a new user required adding them to the *personSystem* table in the auth-database. A stored procedure called *validSystemsInsert* was created. When the user registers for a particular site, several key identifiers are processed: their unique

personID in the auth-database, the systemID of the site for which they registered, and their personID in the primary database of that site.

```
'Routine that runs when user clicks button to check Univ ID & Univ Email
ViewState("id_check") = Me.uiUnivIDCheck.Text
ViewState("email_check") = Me.uiUnivEmailCheck.Text
If authDBCheck(ViewState("id_check"), ViewState("email_check")) Then
        'Code to give user identity confirmation option
Else
        pnlInsertUpdateForm.Visible = True 'New user sign-up
End If
```

```
'Routine that checks in which databases the user exists (for Entropy sign-up)
Dim entExists = False
If entropyDBCheck (ViewState("id_check"), ViewState("email_check")) Then
        'You're already registered in Entropy – please sign in
        entExists = True
End If
If Not entExists Then 'If we didn't find them in Entropy…
        If csbDBCheck(ViewState("id_check"), ViewState("email_check")) Then
                'Duplicate user's CSB record into Entropy
        Else
                Exit Sub
        End If
End If
```

Figure 16 - Checking if User Exists & Inserting

When inserting a new user, it became necessary to figure out how to handle their role(s) in the system—they could be a student, instructor, administrator, web intern, CSB volunteer, or any combination thereof. The login procedure under *Default.aspx.vb* was modified to call another subroutine that retrieves roles. The system design dictates that users are logged in using the authentication database, but their roles remain in the primary databases of the applications. The subroutine receives the matching personID from the authentication procedure, as well as the user's source value. The source value determines which primary database is checked for roles. Like the system values, the roles are then stored in their Person object and kept in session.

Next, user updates needed to be managed. The *profile.aspx.vb* page was changed to pull data from the proper primary database based on the source value. A bigger challenge was replicating updates where necessary. In the insert/update routine of the code, a Boolean object called *isNew* was created. If the personID of the user (stored in a ViewState object) is greater than 0, *isNew* is false, indicating that they are a returning user. After the initial update in the auth-database, a routine runs that updates their information for the site they are currently accessing. The user personID for the opposite system (stored in the cookie) is called from the person session and used to check if a second routine needs to run and update in that location as well. **Figure 17** exhibits this procedure.

```
'Updating a user in multiple places (Entropy example)
If Not isNew Then 'If user already exists…
        authConn.Open() 'Open connection string to auth-database
        cmd.ExecuteNonQuery() 'Update auth-database
        If u.source = 1 Then 'They are in CSB right now…
                csbUpdate(u.csbSys) 'perform CSB update
                If u.entSys > 0 Then
                        entropyUpdate(u.entSys)
                End If
                isOk = True
        End If
        'More code to do the reverse of the above: update an Entropy user,
        'and check to see if a CSB update is needed.
Else
        'They are a new user – perform an insert here
End If
```

Figure 17 - Updating User in Multiple Locations

In order to create a seamless user experience, each page in the authentication application needed to be capable of changing its appearance. Two additional master pages were created for the auth-application, *SiteCSB.master* and *SiteEntropy.master*. Master pages cannot be set in any location outside of the *Page_PreInit* method of a page; this pre-initialization normally runs automatically prior to the *Page_Load* method, and is

not seen by the developer. The pre-initialization method was defined in each page, and checks either the *sys* querystring or the cookie *source* value to determine which master page to adopt.

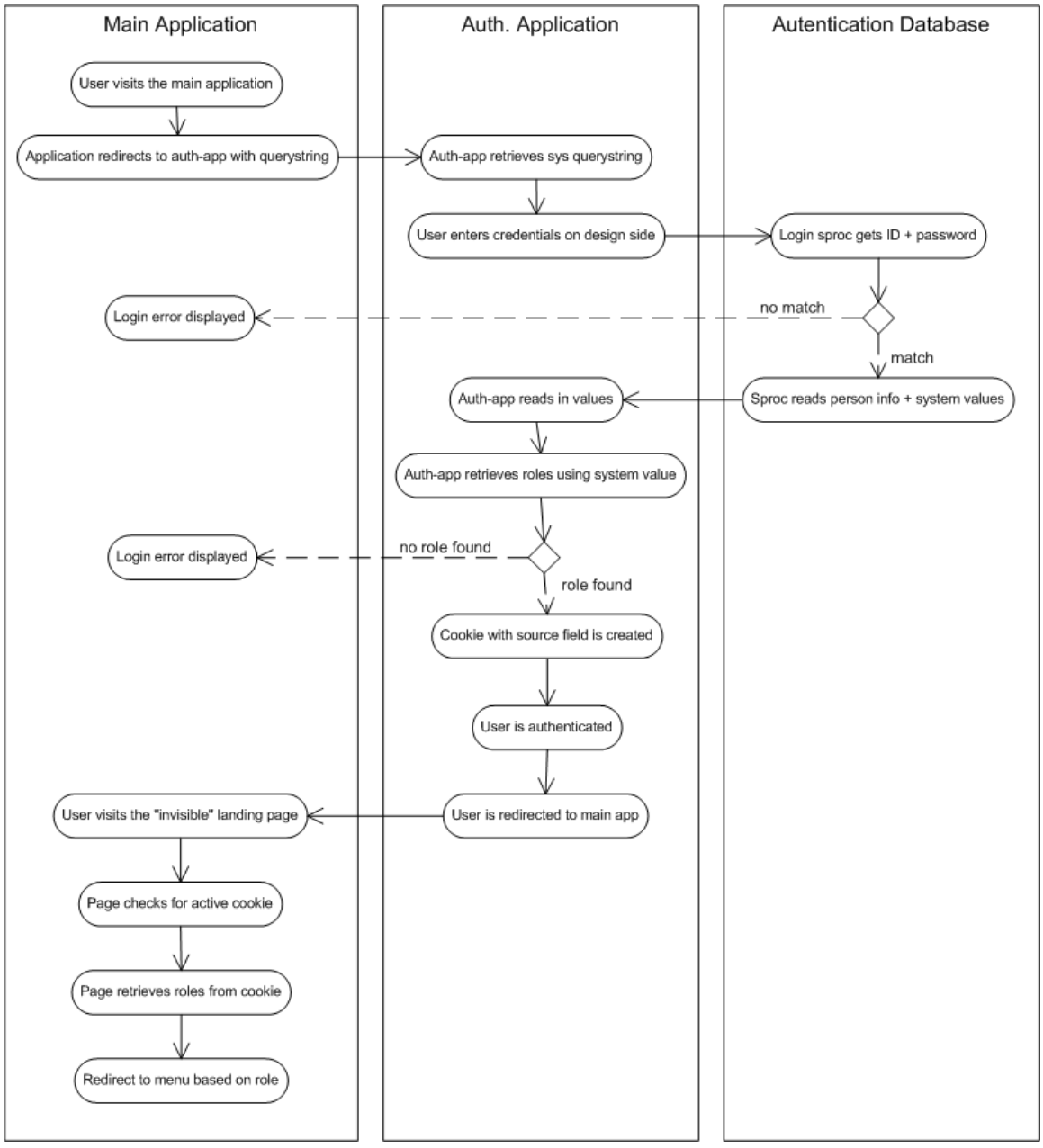


Figure 18 - CAS Login Process

The biggest issue left was one of cross-application communication. Two other small-scale applications were created to represent Entropy and myCSB. These

applications had the bare minimum of functions: a default page that redirected to the auth-application, several types of user menus, and an "invisible" landing page that retrieved the *csbInfo* cookie, grabbed its roles, and redirected the user to the proper menu based on those roles. **Figure 18** shows the process of logging in on the centralized authentication system—from visiting the main application, to ending up at the appropriate menu.

With the primary functions in place, the prototype was near completion. All three applications were moved into a safe workspace on a University server. Backups of the three local test databases were made, and they were migrated onto a SQL Server instance dedicated to testing. Several changes needed to be made to the prototype applications in order to actually test them in this environment. The data connections were changed from the local testing database to those on the testing server, and outbound links were added to properly redirect users between applications.

After the applications and databases were placed into a safe environment, testing commenced. From the start, the applications were able to communicate through the shared cookie, and the redirects worked properly. Through experimenting, several bugs were fixed, until the important authentication functions were operational. This documented, functional prototype provides a strong foundation to implement a centralized authentication scheme in the near future.

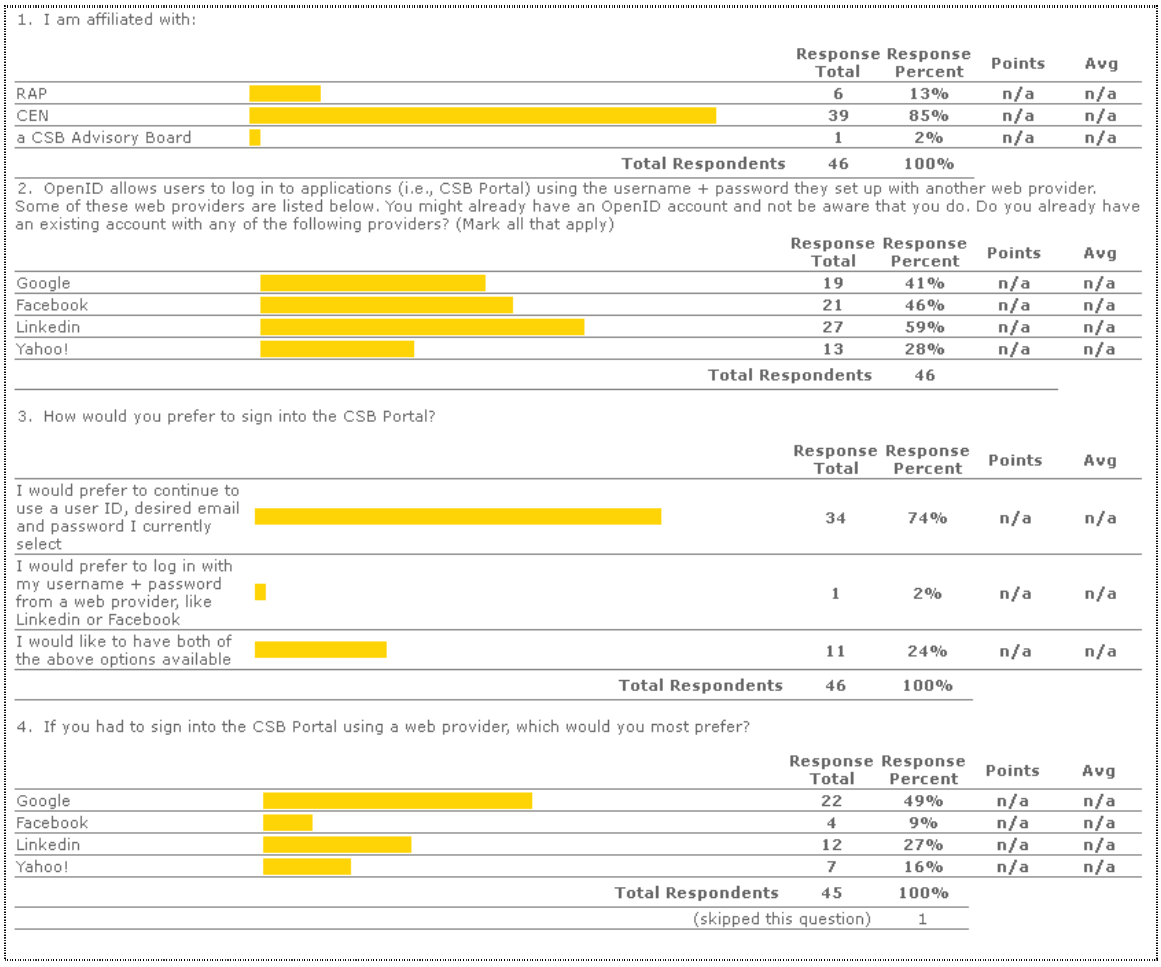
CHAPTER VII: ALTERNATIVE AUTHENTICATION ANALYSIS

7.1 *Implications of Survey Results*

In the early stages of the project, OpenID was considered as a possible avenue for authentication. OpenID allows users to sign in to applications using credentials set up with a peripheral service, such as Google; for more, see chapter 4.2.2. However, such a change should be carefully approached. It was decided that prior to fully committing to OpenID, user response needed to be gauged. A concise, 6-question survey was written up (the survey may be found in Appendix D—Survey Materials). While the focus of the survey is OpenID, this was viewed as an opportunity to acquire further information regarding the general authentication habits of myCSB and Entropy users. The survey was distributed via email to several hundred potential respondents, in two separate mailings: one in late July, and another in early September.

The first survey request received 46 responses total. Of these, 38 were students, while 8 were faculty members. Based on this small sample size, Entropy is visited more often than myCSB, which the majority of respondents visit "only a few times a semester." Unsurprisingly, the majority of respondents (35) have a Facebook account, followed closely by a Google account (31). Surprisingly, Google was the preferred authentication identity provider by a wide margin (27 total, to Facebook's 14). However, there was a perceptible trend against implementation of third-party authentication. Twenty-nine respondents wanted to continue using their current credentials with no changes to the login process, while 17 were interested in decentralized authentication, mostly in addition to the current process.

Via SelectSurvey.NET, it was possible to take a closer look at the answers given by survey participants. It was found that out of the 8 faculty responses, 4 of them did not support OpenID integration; the rest were students. This was surprising. It was speculated that most students would embrace an alternate sign in option, particularly one based on Facebook or Google. In addition, the reason Entropy receives more recurring visits than myCSB is that both students and teachers need to visit it in order to work with grades, assignments, and quizzes.



1. I am affiliated with:

	Response Total	Response Percent	Points	Avg
RAP	6	13%	n/a	n/a
CEN	39	85%	n/a	n/a
a CSB Advisory Board	1	2%	n/a	n/a
Total Respondents	46	100%		

2. OpenID allows users to log in to applications (i.e., CSB Portal) using the username + password they set up with another web provider. Some of these web providers are listed below. You might already have an OpenID account and not be aware that you do. Do you already have an existing account with any of the following providers? (Mark all that apply)

	Response Total	Response Percent	Points	Avg
Google	19	41%	n/a	n/a
Facebook	21	46%	n/a	n/a
Linkedin	27	59%	n/a	n/a
Yahoo!	13	28%	n/a	n/a
Total Respondents	46			

3. How would you prefer to sign into the CSB Portal?

	Response Total	Response Percent	Points	Avg
I would prefer to continue to use a user ID, desired email and password I currently select	34	74%	n/a	n/a
I would prefer to log in with my username + password from a web provider, like Linkedin or Facebook	1	2%	n/a	n/a
I would like to have both of the above options available	11	24%	n/a	n/a
Total Respondents	46	100%		

4. If you had to sign into the CSB Portal using a web provider, which would you most prefer?

	Response Total	Response Percent	Points	Avg
Google	22	49%	n/a	n/a
Facebook	4	9%	n/a	n/a
Linkedin	12	27%	n/a	n/a
Yahoo!	7	16%	n/a	n/a
Total Respondents	45	100%		
(skipped this question)		1		

Figure 19 - 2nd Survey (Volunteers)

A very similar survey was distributed in late September 2012 to UNCW volunteers—namely, CEN, RAP, and CSB Advisory Board members. Volunteers are

important users, and allowing them the option to sign in with one of their social networking accounts (such as Linkedin) could prove to be very convenient. However, as in the previous survey, the results trended strongly against implementation: out of 46 total respondents (39 of them CEN affiliates), 34 were against, and only 12 were in favor. As anticipated, Linkedin was the highest online social service used (27)—but again, Google was the most trusted provider (22). Ultimately, the compiled results of these two surveys display a clear user preference to keep things as they are. User preference is critical; consequently, an OpenID authentication solution will not be implemented at this time. **Figure 19** illustrates the results of the volunteer survey in detail.

As this sign-in technique grows in popularity, it is still worth investigating what is required for implementation. Based on the survey results, the most pertinent authentication techniques to examine are Google authenticator, and Facebook Connect. The following sections discuss the potential requirements for implementing these techniques.

7.2 *Implementation Strategies*

Facebook Authentication

Facebook has become very fashionable in recent years. It is difficult to find an internet user who does not own a Facebook account. Because of this, allowing users to sign into websites using their Facebook account is becoming a common occurrence. Any website or application can do this, including myCSB and Entropy. What follows is the general process of adding Facebook Connect to an ASP.NET website.

First, a distinction must be made between the Facebook API (hereafter called FB-API) and the Facebook Connect API (hereafter called Connect-API). The FB-API works

natively within the Facebook framework. Any applications using this API may only be accessed via the Facebook website. The Connect-API, on the other hand, allows third-party sites/applications (such as Entropy and myCSB) to connect to Facebook across domains, and securely handle user information. As such, the Connect-API is what CSB would use for this type of integration.

To get started, the website or application must be registered with Facebook. This can be done by visiting the FB developers' website (www.facebook.com/developers). The developer can choose whether they are registering an application or a website. During the registration process, an API key will be provided, which is critical for FB authentication to function. They must also specify the callback URL of their project—for example, test-site.com. If necessary, localhost can also be specified for testing locally on a machine.

With registration finished, the developer can now begin coding. In order for the third-party site to interact with Facebook, the Javascript Client Library must be employed. This enables cross-domain communication. To do this, a simple html file called *xd_receiver.htm* should be created at the root level of the solution.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
        <script
        src="http://static.ak.connect.facebook.com/js/api_lib/v0.4/XdCommReceiver
        .js" type="text/javascript"></script>
</body>
</html>
```

Figure 20 - Enabling Cross-Domain Communication

At this stage, the login page—most likely *Default.aspx*, in this case—can be modified to display and run Facebook login. Several pieces must be put into place. First,

the application must be configured to use Facebook Markup Language (FBML), which is Facebook's propriety markup language, and then given access to the JavaScript API.

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:fb="http://www.facebook.com/
2008/fbml">
```

```
<script src=
"http://static.ak.connect.facebook.com/js/api_lib/v0.4/FeatureLoader.js.php"
type="text/javascript"></script>
```

Figure 21 - Configuring FBML & JS-API

A short script must be added to the HTML markup that initializes Facebook Connect for the application. This piece of code includes a call to the *xd_reciever.htm* class mentioned earlier, as well as the API key that is acquired during the website registration process.

```
<script type="text/javascript">
 FB.init("YOUR_API_KEY_HERE", "xd_receiver.htm");
</script>
```

Figure 22 - Initializing FB-Connect & API Key

With these connections in place, a familiar item can now be generated: the Facebook login button. This is also done with a snippet of code.

```
<fb:login-button onlogin="alert('Authenticated!');"></fb:login-button>
```

Figure 23 - Generating FB Login Button

When this button is clicked on the login page, a FB Connect screen appears. (Konrad, 2009) It gives the user the option to 1) log into the application in question, and 2) allow Connect to use their personal information. When the user gives their permission, an access token is returned that provides access to Facebook APIs. This token is a temporary random string containing information about the user and their session.

(Facebook Developers, 2012) The aforementioned is not the only technique for authenticating via Facebook. However, the JavaScript API and FB login button are frequently used, and developer registration is always requisite.

It should be noted that Facebook does *not* permit third-party websites and applications to store Facebook user credentials. This is for security purposes, and is necessary. This encourages developers to integrate Facebook Connect as their primary authenticator, instead of an additional login method. There is minimal information available about adding Facebook Connect as a supplementary authenticator to a primary method, such as ASP.NET Forms. Achieving this would likely entail storing the user's Facebook ID alongside a dedicated set of credentials, and performing a match in order to load their profile information.

Google Authenticator

Like Facebook, Google has wide usage. Google is known primarily for its search engine; nonetheless, the company offers many other services to users and developers. Authentication is one of these services. Google functions as an identity provider, allowing developers to incorporate Google authentication into their projects.

There are different ways to integrate Google authentication into a website or application. However, the easiest involves using the DotNetOpenAuth (DNOA) open source library. DNOA was designed specifically for the Microsoft .NET Framework, and enables developers to employ OpenID and OAuth authentication. DNOA may be downloaded from http://www.dotnetopenauth.net/, and includes AJAX-style controls that can easily be added to pages. (DNOA Contributors, 2012)

The code to implement Google authentication is relatively concise. To start, two namespaces must be imported:

```
USING DotNetOpenAuth.OpenID;

USING DotNetOpenAuth.OpenID.RelyingParty;
```

Figure 24 - Importing DNOA Namespaces

A button should be added to the login page—in this case, the login page is likely *Default.aspx*. The button CommandArgument attribute should point to the following URL:

```
CommandArgument="https://www.google.com/accounts/o8/id"
```

Figure 25 - Google Auth. Link

The underlying code for the button should use the OpenIdRelyingParty and UriBuilder objects to redirect the user to the Google sign-in page. From here, users may login with their Google credentials. After inputting their credentials, the user should then be transferred to their profile on the relying party website. This can be done with a Select Case Statement: in the case of AuthenticationStatus.Authenticated, the user is successfully redirected. Else, they are not, and an error is displayed. (Khandelwal, 2012)

CHAPTER VIII: ASSESSMENT OF CURRENT SYSTEMS & RECOMMENDATIONS OF BEST PRACTICES

While the metrics listed in chapter 4.3 are not all-inclusive, they provide a foundation for assessing the best practices observance of an authentication system. In the following sections, the authentication aspects of Entropy and the CSB Portal will be subjected to a general evaluation, anchored in the metrics explored in this paper. Based on the results, recommendations will be made where applicable, and design decisions will be clarified.

8.1 *Assessment of Metrics*

Multi-Factor/Strong Authentication

Like most websites, myCSB and Entropy do not make user of multi-factor authentication. In order to do this, the systems would need to add another layer of authentication, in addition to the username/password structure already in place. Effectively achieving this is challenging for websites. Often, it requires bringing in external hardware, though there are ways to enforce multiple authentication methods without doing this.

A common method of proving user identity alongside passwords is security questions. When the user logs in with their credentials, they may first be redirected to a security question page prior to the landing page. As discussed in chapter 4.3, security questions are often viewed as an inadequate security measure; nevertheless, this technique could be effective, as long as it is not employed as a primary authentication technique (for instance, it may be used as a prerequisite for successfully submitting a "change password" request). Other methods of identity verification may be easily

spoofed, such as MAC address identification. Ultimately, the systems in question are not high-security, and make up for the lack of multi-factor authentication in other areas (encryption, HTTPS, and judicious design).

Encryption & Hashing

It is crucial for the systems in question to obfuscate the sensitive data of its users. As documented in chapter 2.3, myCSB and Entropy both use RC4 stream cipher to encrypt passwords and sensitive data. Furthermore, both systems make use of a salt, which is strongly encouraged. While RC4 gets the job done, it is somewhat outdated; there are newer, more robust encryption/hashing algorithms available for use.

Chapter 4.3 explains the significant distinction between encryption algorithms and hashing functions. Because Entropy and myCSB use an encryption algorithm, it is possible to decrypt the passwords and view them as plaintext. It is preferable to use a one-way hashing function, such as Bcrypt (the Bcrypt.NET Library exists for precisely this situation). A password should be viewed as the property of the user that chose it—nobody but them has any reason to see it, or even have a means to do so. A hashing function like Bcrypt with a unique salt ensures that—as far as the system is concerned—*only* the user can know their password.

The above technique would also require a restructuring of the password retrieval system, since passwords could no longer be decrypted. Passwords would no longer be retrieved via email; as an alternative, the user would be provided with a link to reset their password completely. For more, see the *User Convenience Aspects* subdivision of this section.

Cookies

Cookies are currently used in both myCSB and Entropy. They are an integral part of the authentication process. Without a session cookie, users would be unable to stay logged in for extended periods of time. ASP.NET does natively provide cookieless sessions that work by embedding session information directly into the website URL. This is dangerous and could lead to account hijacking—cookies, despite their vulnerabilities, are therefore preferred. (Hunt, 2011)

In both systems, cookies are created by calling ASP.NET's type-safe HttpCookie class. Neither system allows session IDs to be reused (this is possible using SessionStateSection.RegenerateExpiredSessionId). It is important to ensure that cookies are always sent over HTTPS; both Entropy and myCSB use HTTPS when logging users in, and when changing passwords. The systems also abide by best practices by encrypting the data stored in the cookies (see **Figure 26**).

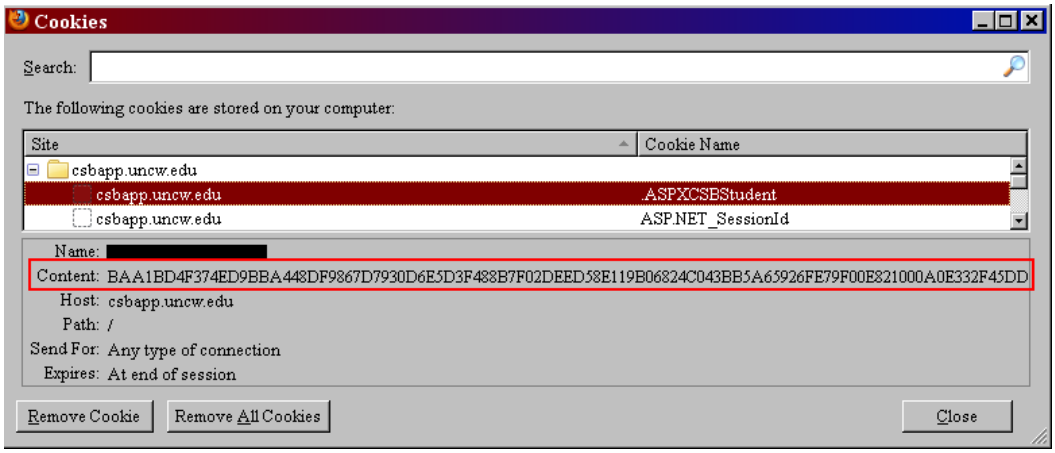


Figure 26 - Encrypted CSB Cookie Data

Stored Procedures

Stored procedures are used to great effect in both systems. Essentially all important operations are relegated to sprocs. This includes authentication-related functions, such as logging in, changing and retrieving passwords, and modifying users in

the systems. The sprocs are tightly integrated with code; when changes are made on one side, the other must be modified as well to avoid breaking things. Due to the nature of stored procedures, this is unavoidable. In this case, sprocs are definitely preferable to parameterized SQL; both systems are too large, and employ too many different functions, to effectively implement the latter.

More importantly, the databases themselves adhere to appropriate normalization rules. The main tables use primary key identifiers, typically unique integers. Tables that could encapsulate many fields are broken into several smaller tables—for example, instead of placing the Address field directly into the Person table, the myCSB database uses a validAddressTypes table to determine all *kinds* of addresses, and links them up with People in tblAddress (with foreign key identifiers). By following normalization rules, myCSB and Entropy minimize data redundancy and dependency.

It should be noted that some of the tables and procedures are relics of previous SQL versions, and are no longer in SQL 2008 or 2012. For instance, several tables related to the Person object in the csbLive2007 database—tblPersonDesiredEmail, tblPersonUnivEmail, and tblPerson85—can be safely dropped, once the relevant stored procedures are modifed.

Additional Security Aspects

Additional security aspects as they relate to passwords are minimal in Entropy and myCSB. Users are not required to change their passwords after a set amount of time—they may be changed when the user chooses, or when requested by the system admin. There is no minimum duration for password changes, and a password history is

not enforced. Account lockouts are not enforced. Session timeouts are implemented, which is obligatory for Entropy, and standard in the live version of myCSB.

This above arrangement is sufficient since users typically own these accounts for only a few years before leaving the university. However, it is recommended that a per-semester password change is imposed; the *Person* table in the authentication database already contains a bit field that specifies whether or not a user needs to change their password. This provides a starting point for timed password changes. Additionally, there is no need to add security questions to the system, but they could be strategically useful as an additional security measure.

User Convenience Aspects

Entropy and myCSB make use of the user convenience aspects covered in this paper, with minor implementation differences. The voluntary password change and forgotten password retrieval functions are necessary in these systems.

The voluntary change password operation is a separate function in each system, with its own page and associated stored procedure. The password retrieval function is also a separate function in each system. The system pulls the inputted email from the text box, and finds the associated encrypted password (binary) via stored procedure. The password is then decrypted, and a method called *SendEmail*—located in the *App_Code* folder—sends the message using the SmtpClient class (a subset of System.Net.Mail).

While the previous arrangement works, it may be preferable to instead implement a password reset mechanism. Instead of emailing the user the password itself, the user receives a link with a non-persistent unique token that is tied to their account. When the link is clicked, they are prompted to reset their password entirely. This requires no

decryption of sensitive data, though it may have a negative effect on user convenience, since it forces users to specify a new password instead of recovering the forgotten one.

As noted in chapter 2.3, Entropy does not allow persistent login, since it handles grades. This is a necessary decision. Conversely, the CSB portal—which handles less sensitive data than its counterpart—does allow persistent login. However, it is not actually available on the production site; it is currently enabled only on the intern test site that is now under construction. This is prudent, as persistent login is simply not necessary; it exists primarily to increase the "stickiness" of a website, and encourage return visits. Entropy and myCSB are university-affiliated and have no need of this brand of marketing—users are obligated to return to these sites for non-entertainment purposes. (Hunt, 2011)

CHAPTER IX: TIME LINE & PROJECT SUMMARY

9.1 *Timelines*

Original Timeline

The table below displays the major activities that occurred during the project process. The estimated timeline and the actual timeline are shown side-by-side for comparison.

Table 2 - Early Timeline

ACTIVITY	ESTIMATED	ACTUAL
Decide on research project	Spring 2012	April 2012
Assemble committee	by July	Mid-August
Decide on committee chair	by July	May 2012
Study authentication best practices	May 1 – June 30	June 1 – July 31
Study myCSB & Entropy systems	June 1 – June 30	June 15 – ~July 3
Write up authentication best practices	by June 30	June 25 – August 1
Document existing systems info	by June 30	July 5
Distribute authentication survey	by June 30	July 12
Compile survey results	July 15	July 27
Study project methodologies	July 1—July 14	July 1 – July 24
Pick a methodology	by July 15	August 9
Work out recommendations	July 1 – July 15	August 9
Proposal paper rough draft	July 15 – August 15	July 20
Go over proposal draft with chair	~August 15	August 9
Proposal completed	by ~September 3	September 7
Proposal date	~September 17	September 15
Development phase begins	September 18	September 16
Work on final research paper	September 18 – November 31	October 28—November 25
Development completed	by late November	November 30
Finish final research paper	by late November	November 25
Final defense	December 2012	November 19

Revised Timeline

	Task Name	Duration	Start	Finish	Predecessors
1	**Analysis & Early Documentation**	**18 days?**	**Thu 9/13/12**	**Sun 9/30/12**	
2	Begin Best Practices Documentation	18 days?	Thu 9/13/12	Sun 9/30/12	
3	Retrieve OpenID Survey 2 Results	1 day?	Tue 9/18/12	Tue 9/18/12	
4	Compile Results & Write-up	3 days?	Wed 9/19/12	Fri 9/21/12	
5	Create Use Cases	5 days?	Mon 9/24/12	Fri 9/28/12	
6	Create CRUD	5 days?	Mon 9/24/12	Fri 9/28/12	
7	**Further Documentation & Implementation Initiation**	**44 days?**	**Thu 10/4/12**	**Fri 11/16/12**	
8	Begin OpenID Requriements Documentation	16 days?	Thu 10/4/12	Fri 10/19/12	
9	Prepare safe testing space	2 days?	Mon 10/8/12	Tue 10/9/12	
10	Create authentication database	17 days?	Mon 10/15/12	Wed 10/31/12	
11	Modify client-side code as needed	14 days?	Mon 10/15/12	Sun 10/28/12	
12	Begin best practices integration	20 days?	Sun 10/28/12	Fri 11/16/12	
13	**Final Documentation & OpenID Implementation Phase**	**27 days?**	**Fri 11/2/12**	**Wed 11/28/12**	
14	Begin OpenID pilot test or Implementation	15 days?	Fri 11/2/12	Fri 11/16/12	
15	Continue fine-tuning centralized authentication	7 days?	Sat 11/3/12	Fri 11/9/12	
16	Continue fine-tuning best practices	8 days?	Wed 11/21/12	Wed 11/28/12	
17	OpenID fine tuning	8 days?	Wed 11/21/12	Wed 11/28/12	
18	Complete all documentation	18 days?	Sun 11/11/12	Wed 11/28/12	
19	**Final Paper & Defense**	**18 days?**	**Fri 11/23/12**	**Mon 12/10/12**	
20	Compile notes for final defense paper	6 days?	Fri 11/23/12	Wed 11/28/12	
21	Put together final defense paper	6 days?	Fri 11/30/12	Wed 12/5/12	
22	Create final defense presentation	5 days?	Mon 12/3/12	Fri 12/7/12	
23	Final defense	1 day?	Mon 12/10/12	Mon 12/10/12	

Figure 27 - Fall Timeline

9.2 *Timeline Analysis & Lessons Learned*

The first timeline was constructed in late July of 2012. The second timeline was created for the project proposal, and was meant to display to the committee the feasibility of completing the project goals. This timeline was designed to include weekends, since I had no intentions of taking those days off. Furthermore, no predecessors were specified, since most of the project activities could be undertaken simultaneously. I managed to stay largely on-schedule for most of the activities; however, because the centralized authentication took so much time, the other deliverables were cut back. Also, I was unrealistic in my assumption that I would be able to continue working on the project well

into December; in reality, the paper was due on November 26, and the development needed to be completed before December.

I learned a great deal during the project process. First of all, it is important not to over-extend yourself. Early on in a project, it is easy to look at what needs to be done, and start adding objectives. Once the project actually begins and demands attention to detail, things can quickly get overwhelming; the "actual" dates on my timeline illustrate this. The centralized authentication database seemed easy to achieve from a distance, but once I began working on it, the additional objectives I added—the OpenID pilot test and best practices analysis and execution—appeared to be more time-consuming and difficult. Luckily, I was able to avoid scope creep; on the contrary, I ended up lessening my workload as needed.

During the project process, I learned that things will happen that can hinder progress. Some things aren't foreseeable, and other obligations can take up a lot of time. For me, this included work, family, and pet care. Between all of these activities, I found it challenging to stay motivated and avoid procrastinating.

I also realized the enormity of planning well in advance when dealing with other people. They have their own schedules, so meetings and events should be nailed down early. This became very clear as I was preparing meetings in the spring semester while searching for a committee, and again in the fall during advising season. On a related note, there is nothing wrong with asking others for help. Sometimes, their advice and views on the matter are exactly what is needed. This is something I already knew, but the idea was solidified during this project.

During my time at UNCW, I was taught the importance of design. A project is far more likely to succeed if there is significant time spent on the design phase, learning about the objectives and putting together practical diagrams. This project was not an exception to this rule. I was also glad that I tend to save all of my past work from previous classes. At certain points in the project, I would hit a roadblock that could be overcome by studying past schoolwork. This is another idea of which I already knew the value, which was further demonstrated during this project.

During the coding portion of the project, it was easy to become frustrated when facing a problem. I found that it was helpful to only code a little at a time, and comment heavily (the latter is a best practice). Sometimes, when trying to debug a piece of code—such as a faulty stored procedure—it was helpful to simply erase what was there, and start again from scratch. Overall, working on pre-existing code is a delicate procedure. It is challenging to add or modify features without breaking something.

On the whole, I learned a great deal during this undertaking. I learned about working with others, keeping motivated, and staying organized as the work piles up. This was a good experience that will aid me in future endeavors.

9.3 *Future Work*

This project could provide a jumping-off point for future projects. The three main objectives—the consolidated authentication database, the decentralized authentication scheme, and the best practices assessment and implementation—offer a basis for more work.

The centralized authentication was designed to accommodate additional systems. As of now, only Entropy and myCSB are supported, but future projects could focus on

adding the University college Advising and Events systems. As it stands, the centralized authentication client-side is a separate application; future projects could focus on changing it to a web service.

The OpenID portion of this project did not go as far as anticipated. Due to negative user feedback, the initial implementation or pilot test plans were scrapped. However, user preferences toward third-party authentication may change in the future. If that happens, this paper provides a short description of what is needed for two of the most popular authenticators, as well as further resources in the references. It could be quite a challenge to implement decentralized authentication in addition to the existing procedure.

The best practices reviewed in this paper could also offer a foundation for potential projects. For instance, the encryption scheme could be analyzed and updated to a make use of a more robust algorithm. New password policies could be put into place (such as a password reset, instead of a password retrieval), or even the addition of security questions or CAPTCHA.

ACKNOWLEDGEMENTS

I would like to thank Dr. Tom Janicki, for acting as my project chair, and for helping me form a project topic, begin my research, and make significant progress. Thanks to Dr. Doug Kline and Dr. Laurie Patterson, for acting as committee members for this project, and for offering ideas and support during the research process. Thanks also to Kevin Matthews, for help in administering the associated authentication survey.

REFERENCES

Sommerville, I. (2010). *Software engineering*. (9 ed.). Boston, MA: Addison-Wesley.

Hoffer, J. A., Prescott, M. B., & Topi, H. (2009). *Modern database management*. (9 ed.). Upper Saddle, NY: Prentice Hall.

Boehm, A. (2008). *Murach's asp.net 3.5 web programming with vb 2008*. Mike Murach & Associates Inc.

Shelly, G.B., Cashman, T.J., & Vermaat, M.E. (2007) *Discovering Computers 2007 A Gateway to Information.* Thomson Course Technology.

Satzinger, J. W., Jackson, R. B., & Burd, S. D. (2010). *Systems analysis and design in a changing world*. (5th ed ed.). Boston: Course Technology Ptr.

Satzinger, J. W., Jackson, R. B., & Burd, S. D. (2004). *Object-oriented analysis and design, with the unified process*. Course Technology Ptr.

Committee on National Security Systems. (2010). *National Information Assurance Glossary. CNSS Instruction No. 4009.*

Mallow, C. Authentication Methods & Techniques. *SANS® +S™ Training Program for the CISSP® Certification Exam.*

Federal Financial Institutions Examination Council (2008). Authentication in an Internet Banking Environment.

de Borde, D. Two-factor authentication. Considerations for selection of a two-factor authentication system. *Siemens Insight Consulting—White Paper.*

Sun, S., Popisil, E., Muslukhov, I., Dindar, N., Hawkey, K., & Beznosov, K. (2011). What Makes Users Refuse Web Single Sign-On? An Empirical Investigation of OpenID. *Symposium on Usable Privacy and Security (SOUPS) 2011, July 20-22, 2011, Pittsburgh, PA USA.*

Sutcu, Y., Sencar, H., & Memon, N. (2005). A Secure Biometric Authentication Scheme Based on Robust Hashing. *MM-SEC'05, August 1–2, 2005, New York, New York, USA.*

Jaspan, Barry. (2007). Improved Persistent Login Cookie Best Practice. Retrieved August 7, 2012 from *http://jaspan.com/improved_persistent_login_cookie_best_practice*

Florencio, D., & Herley, C. (2007) A Large-scale Study of Web Password Habits. *In Proceedings of the 16th International Conference on World Wide Web, pages 657-666, New York, NY, USA, 2007. ACM.*

Kim, W., Jeong, O., Kim, C., & So, J. (2010) On Botnets. *iiWAS-2010, 8-10, November, 2010, Paris, France.*

Bursztein, E., Martin, M., & Mitchell, J.C. (2011). Text-based CAPTCHA Strengths and Weaknesses. *CCS'11, October 17-21, 2011, Chicago, Illinois, USA.*

Microsoft. (2005). Password Best Practices: Logon and Authentication. Retrieved August 2, 2012 from *http://technet.microsoft.com/enus/library/cc784090%28v=ws.10%29.aspx*

Chinta, R., Danseglio, M., & Resnick, M. (2004). Account Lockout Best Practices. *Microsoft White Paper.*

Wash, Rick. (2001). Lecture Notes on Stream Ciphers and RC4. *Working paper—Case Western Reserve University*.

Hunt, Troy. (2011). OWASP Top 10 for .NET developers. *Blog series.*

Facebook Developers. (2012). Access Tokens and Types. Retrieved November 20, 2012 from *https://developers.facebook.com/docs/concepts/login/access-tokens-and-types/*

Konrad, Bill. (2009). How to Integrate with Facebook Connect. Retrieved October 27, 2012 from *http://devtacular.com/articles/bkonrad/how-to-integrate-with-facebook-connect/*

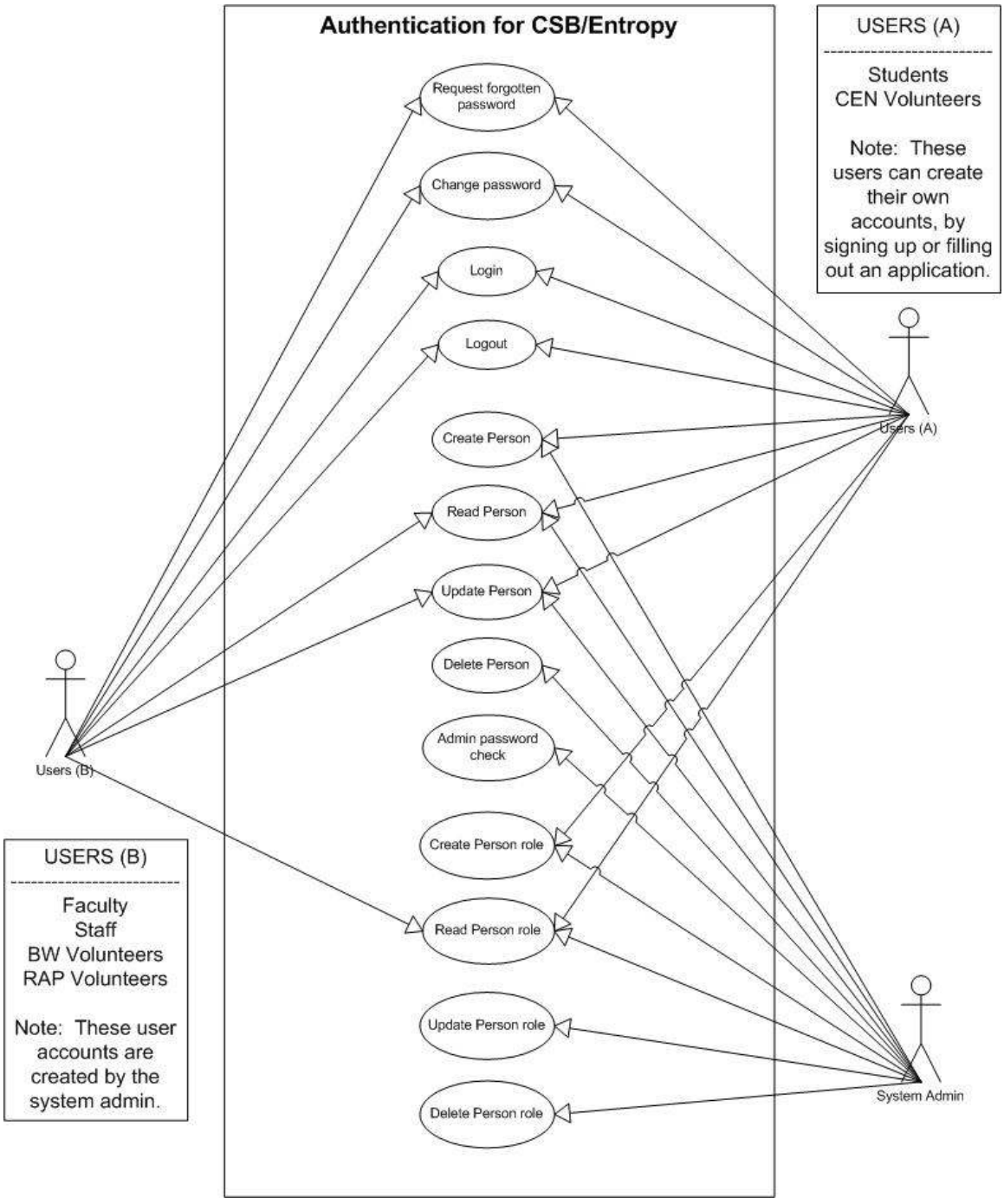
Khandelwal, Prashant. (2012). Implementing Google Account Authentication in an ASP.NET application. Retrieved October 28, 2012 from *http://dotnet.dzone.com/news/implementing-google-account*

DNOA Contributors. 2012. DotNetOpenAuth – OpenID, OAuth, and InfoCard for .NET. Retrieved October 28, 2012 from *http://www.dotnetopenauth.net/*

APPENDICES

APPENDIX A – CAS Diagrams

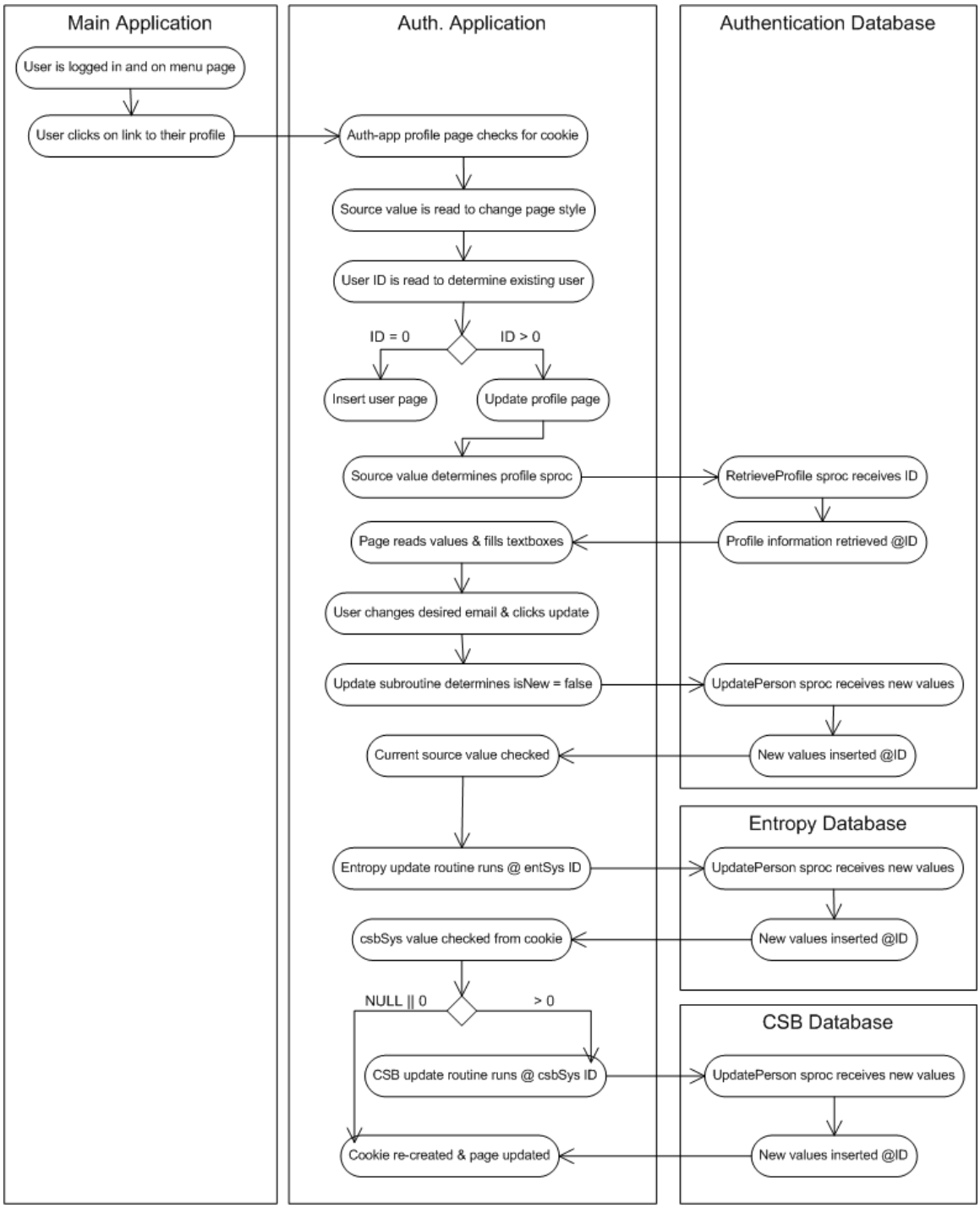
Use Case Diagram



CRUD Diagram

	A	B	C
1	CSB/Entropy Auth. CRUD		
2		**Person**	**PersonRoles**
3	Login	R	R
4	Logout	-	-
5	changePassword	U	-
6	requestPassword	R	-
7	adminPasswordCheck	R	-
8	createPerson	C	-
9	readPerson	R	-
10	updatePerson	U	-
11	deletePerson	D	-
12	createPersonRole	-	C
13	readPersonRole	-	R
14	updatePersonRole	-	U
15	deletePersonRole	-	D

Swimlane Diagram – Example of Changing Desired Email on Entropy



* The purpose of the above diagram is to illustrate what happens in the centralized authentication system when an Entropy user changes their desired email.

APPENDIX B – Use Case Descriptions

<table>
<tr><td>Use Case Name:</td><td>Logout</td></tr>
<tr><td>Scenario:</td><td>A logged in user decides to end their session</td></tr>
<tr><td>Triggering Event:</td><td>A user clicks the logout button.</td></tr>
<tr><td>Brief Description:</td><td>A user clicks the logout button and is redirected to the landing page of the website. Their session is terminated.</td></tr>
<tr><td>Actors:</td><td>All valid users</td></tr>
<tr><td>Related Use Cases:</td><td></td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>User must have an active session.</td></tr>
<tr><td>Postconditions:</td><td>The user session cookie must be terminated.
Any persistent login cookies must also be terminated (CSB only).
The user must be redirected to the main page of the website.</td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table><tr><td>1. A logged-in user clicks the “logout” button.

2. The user is redirected to the landing page.</td><td>1.1 The system terminates the user session.
1.2 The system attempts a redirect to the landing page.</td></tr></table></td></tr>
<tr><td>Exception Conditions:</td><td>1. If the user’s sessions has already timed out, the logout function will be bypassed for the timeout redirect.</td></tr>
</table>

<table>
<tr><td>Use Case Name:</td><td>changePassword</td></tr>
<tr><td>Scenario:</td><td>A logged in user decides to change their current password</td></tr>
<tr><td>Triggering Event:</td><td>User clicks the “change password” button</td></tr>
<tr><td>Brief Description:</td><td>The user visits the “change password” link while logged in. The user fills in the required information, including the old password and the new, and clicks the “change password” button.</td></tr>
<tr><td>Actors:</td><td>All valid users</td></tr>
<tr><td>Related Use Cases:</td><td>updatePerson</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>User must have an active session.
User’s new password must meet minimum security requirements.</td></tr>
<tr><td>Postconditions:</td><td>The password must be changed in the database.</td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table>
<tr><td>1 A logged in user views their profile and clicks the “update password” link.</td><td></td></tr>
<tr><td>2 The user enters their existing password and the new password.</td><td></td></tr>
<tr><td>3 The user clicks the “change password” button.</td><td>3.1 The system checks that the old and new passwords are valid.
3.2 The system processes the password change.</td></tr>
<tr><td>4 The message of success is displayed.</td><td></td></tr>
</table></td></tr>
<tr><td>Exception Conditions:</td><td>3. If the old password is not correct, an error is displayed.
If the new password equals the old password, an error is displayed.</td></tr>
</table>

<table>
<tr><td>Use Case Name:</td><td>requestPassword</td></tr>
<tr><td>Scenario:</td><td>A user who cannot remember his or her password requests it</td></tr>
<tr><td>Triggering Event:</td><td>The user clicks the “request password” button</td></tr>
<tr><td>Brief Description:</td><td>The user visits the “recover password” link. They supply a valid email address, and click the “recover password” button. Their current password is emailed to their supplied address</td></tr>
<tr><td>Actors:</td><td>All valid users</td></tr>
<tr><td>Related Use Cases:</td><td>readPerson</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>The user must have an existing account.
The user’s current password must not be expired.
The user’s supplied email must be valid.</td></tr>
<tr><td>Postconditions:</td><td>An email must be sent to the user containing their password.</td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table><tr><td>1 A user (not logged in) visits the “password help” link on the landing page.
2 The user enters a valid email.
3 The user clicks the “retrieve password” button.

4 The user can now retrieve the password from their email.</td><td>

3.1 The system processes the inputted email address, and ensures that it exists.
3.2 The system sends a message containing the related password to the user.</td></tr></table></td></tr>
<tr><td>Exception Conditions:</td><td>3 If the email address entered is invalid, the user receives no message, and an error is displayed that the email is not on record.
3 If a volunteer user deletes their desired email but cannot remember their password, they must contact the admin, or create a new account entirely.</td></tr>
</table>

<table>
<tr><td>Use Case Name:</td><td>adminPasswordCheck</td></tr>
<tr><td>Scenario:</td><td>The sysadmin needs to find out a password</td></tr>
<tr><td>Triggering Event:</td><td>The sysadmin clicks the button to return the password</td></tr>
<tr><td>Brief Description:</td><td>Via an HTTPS connection, the sysadmin enters the user email; the password is decrypted and returned.</td></tr>
<tr><td>Actors:</td><td>System administrator</td></tr>
<tr><td>Related Use Cases:</td><td>readPerson</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td></td></tr>
<tr><td>Postconditions:</td><td></td></tr>
<tr><td>Flow of Activities:</td><td>Actor System<table><tr><td>1. The sysadmin accesses the user management GUI.
2. The sysadmin enters a valid email or 85# for the user in question.

3. The password attached to the email is displayed to the sysadmin.</td><td>

3.1 The system processes the email or 85# and finds the matching password.
3.2 The password is decrypted.
3.3 The password is sent to the application.</td></tr></table></td></tr>
<tr><td>Exception Conditions:</td><td>2. If the email or 85# entered is invalid, an error is thrown.</td></tr>
</table>

<table>
<tr><td>Use Case Name:</td><td>createPerson</td></tr>
<tr><td>Scenario:</td><td>A new user is created in the database</td></tr>
<tr><td>Triggering Event:</td><td>The new user clicks the “Update Profile” or “Update Information” button</td></tr>
<tr><td>Brief Description:</td><td>The user visits the “new users” link. The user fills out the required information, including a valid UNCW email address and ID number. The user clicks the “Update Profile” or “Update Information” button, which builds their account and creates a new Person in the database.</td></tr>
<tr><td>Actors:</td><td>Students, CEN Volunteers, Sysadmin, Dept. admin assistant, Dept. chair</td></tr>
<tr><td>Related Use Cases:</td><td>createPersonRole</td></tr>
<tr><td>Stakeholders:</td><td>Students, CEN Volunteers</td></tr>
<tr><td>Preconditions:</td><td>The user must have valid UNCW affiliation (ie, student, faculty, etc.).
The user must have a valid UNCW email and ID number.
The user must not already have an existing account.</td></tr>
<tr><td>Postconditions:</td><td>The information inputted by the user must be reflected in the database.
The user must be sent to the appropriate landing page based on their role.</td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table><tr><td>1. A new user clicks the “new users click here” link on the landing page.
2. The user fills in the required fields.
3. The user clicks the “update information” button.

4. A message of success is displayed.</td><td>

3.1 The system processes the information in the fields and creates a new Person in the database.
3.2 A role is created for that user.</td></tr></table></td></tr>
<tr><td>Exception Conditions:</td><td>3. If the user already has an account, they cannot make another.
If any of the required fields are empty or invalid, an error is displayed.</td></tr>
</table>

NOTE: This use case follows the creation of a user by signing up for a new account. Users may also be created directly by a system administrator. Users that MUST be created by a sysadmin include: faculty, staff, and all volunteers (with the exception of CEN volunteers, whose records are generated when they create an application to be a CEN member). New faculty may also be added by the department admin assistant or department chair.

<table>
<tr><td>Use Case Name:</td><td>readPerson</td></tr>
<tr><td>Scenario:</td><td>Information about a user that is stored in the database is viewed</td></tr>
<tr><td>Triggering Event:</td><td>The “profile” link is visited</td></tr>
<tr><td>Brief Description:</td><td>A logged in user is on their homepage. The user clicks on the “profile” link, and is taken to a page listing information about their profile (their Person information).</td></tr>
<tr><td>Actors:</td><td>All valid users</td></tr>
<tr><td>Related Use Cases:</td><td>readPersonRole</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>The user must have an active session.</td></tr>
<tr><td>Postconditions:</td><td></td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table>
<tr><td>1 The logged-in user clicks on the “Profile” link on their homepage.</td><td>1.1 The system retrieves the profile information from the Person table in the database.</td></tr>
<tr><td>2 The user views their profile information.</td><td>1.2 The system displays the relevant information on the profile page.</td></tr>
</table></td></tr>
<tr><td>Exception Conditions:</td><td>1. If the user’s sessions has already timed out, they will be redirected to a timeout page.</td></tr>
</table>

NOTE: This use case follows the viewing of a Person by editing a profile. A user’s Person information may also be viewed by other users through the search function, or by a sysadmin looking them up in the user management GUI.

<table>
<tr><td>Use Case Name:</td><td>updatePerson</td></tr>
<tr><td>Scenario:</td><td>Information about a user that is stored in the database is changed</td></tr>
<tr><td>Triggering Event:</td><td>An “update” button is clicked by the user</td></tr>
<tr><td>Brief Description:</td><td>The user is already logged in and has viewed their profile information. To update their profile information, the user clicks an “update” link (ie, update education, update biography, update above, etc.). The user fills in the relevant information and presses a button to update their profile.</td></tr>
<tr><td>Actors:</td><td>All valid users</td></tr>
<tr><td>Related Use Cases:</td><td>readPerson, updatePersonRole</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>The user has an active session.
The information entered by the user meets formatting requirements.</td></tr>
<tr><td>Postconditions:</td><td>The changes the user made must be reflected in the database.
The changes the user made must be reflected on their profile page.</td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table><tr><td>1. The logged-in user clicks the “Profile” link on their homepage.</td><td></td></tr><tr><td>2. The user clicks the “update” button for one of the listed categories.</td><td>2.1 The system creates redirects the user to the appropriate update page, with editable fields.</td></tr><tr><td>3. The user fills in the relevant information for that category.</td><td></td></tr><tr><td>4. The user clicks the finish button to save changes.</td><td>4.1 The system processes the inputted information, checks for errors, and makes the changes in the database.</td></tr><tr><td>5. A success message is displayed to the user.</td><td>4.2 The user is redirected to a success page.</td></tr></table></td></tr>
<tr><td>Exception Conditions:</td><td>4. If any of the information the user inputted is invalid, an error is thrown and the user may not proceed with the update.
If the user attempts to change their email/desired email, the system must check the ensure that the email does not already exist for another user.</td></tr>
</table>

NOTE: This use case details the updating of a Person by editing a profile. A Person’s information may also be updated by a sysadmin, via the user management GUI.

<table>
<tr><td>Use Case Name:</td><td>deletePerson</td></tr>
<tr><td>Scenario:</td><td>An existing user is removed from the database</td></tr>
<tr><td>Triggering Event:</td><td>An administrator removes a user</td></tr>
<tr><td>Brief Description:</td><td>A system administrator must remove a user. The administrator has access to a GUI that asks for the user’s UNCW email, and brings up the standard profile screen. The user may be removed from here without directly accessing the database.</td></tr>
<tr><td>Actors:</td><td>System administrator, Dept. admin assistant, Dept. chair</td></tr>
<tr><td>Related Use Cases:</td><td>deletePersonRole</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>Admin has privileges to CRUD users.</td></tr>
<tr><td>Postconditions:</td><td>The user in question can no longer log in.</td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table><tr><td>1. The sysadmin accesses the user management GUI.
2. The sysadmin enters a valid email for the user in question.

3. The user’s profile information is displayed.
4. The sysadmin chooses to “delete” the user.

5. The user is removed.</td><td>

2.1 The system processes the email and looks up the matching Person information for that email.

4.1 The system deletes the user by removing their Person information, roles and additional information from the DB.</td></tr></table></td></tr>
<tr><td>Exception Conditions:</td><td>2. If the email is invalid, an error is thrown.</td></tr>
</table>

<table>
<tr><td>Use Case Name:</td><td>createPersonRole</td></tr>
<tr><td>Scenario:</td><td>A role is created for a new user</td></tr>
<tr><td>Triggering Event:</td><td>A new Person is created in the database</td></tr>
<tr><td>Brief Description:</td><td>When a new user is created in the database, an associated role must be assigned. This role determines what they can and cannot do (and what they see) when logged in.</td></tr>
<tr><td>Actors:</td><td>Students, CEN Volunteers, Sysadmin, Committee Chairs</td></tr>
<tr><td>Related Use Cases:</td><td>createPerson</td></tr>
<tr><td>Stakeholders:</td><td>Students, CEN Volunteers</td></tr>
<tr><td>Preconditions:</td><td>The createPerson action must be successful.</td></tr>
<tr><td>Postconditions:</td><td></td></tr>
<tr><td>Flow of Activities:</td><td>Actor System<table><tr><td>1. (see createPerson use case) The user signs up for an account and a new Person is created.</td><td>1.1 When a new Person is created in the database (Person table), an associated role is also created (in the PersonRoles table).</td></tr></table></td></tr>
<tr><td>Exception Conditions:</td><td></td></tr>
</table>

NOTE: This use case details the creation of a Person role when the user creates an account. As noted in the createPerson use case, some users must be created by a sysadmin—roles are created in those instances, as well.

<table>
<tr><td>Use Case Name:</td><td>readPersonRole</td></tr>
<tr><td>Scenario:</td><td>A user’s stored role in the system is viewed</td></tr>
<tr><td>Triggering Event:</td><td>The “profile” link is visited</td></tr>
<tr><td>Brief Description:</td><td>A logged in user is on their homepage. The user clicks on the “profile” link, and is taken to a page listing information about their profile (their Person information)—this includes their role.</td></tr>
<tr><td>Actors:</td><td>All valid users</td></tr>
<tr><td>Related Use Cases:</td><td>readPerson</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td></td></tr>
<tr><td>Postconditions:</td><td></td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table>
<tr><td>3 The logged-in user clicks on the “Profile” link on their homepage.</td><td>3.1 The system retrieves the profile information from the Person table in the database, including their role in the system.
3.2 The system displays the relevant information on the profile page.</td></tr>
<tr><td>4 The user views their role in the profile information.</td><td></td></tr>
</table></td></tr>
<tr><td>Exception Conditions:</td><td></td></tr>
</table>

NOTE: This use case follows the viewing of a role by accessing a profile. A user’s role may also be viewed by other users through the search function, or by a sysadmin looking them up in the user management GUI.

<table>
<tr><td>Use Case Name:</td><td>updatePersonRole</td></tr>
<tr><td>Scenario:</td><td>A user’s stored role is changed</td></tr>
<tr><td>Triggering Event:</td><td>The sysadmin chooses to update the user’s role</td></tr>
<tr><td>Brief Description:</td><td>A system administrator must update a user’s role. The administrator has access to a GUI that asks for the user’s UNCW email, and brings up the standard profile screen. The user role may be changed from here without directly accessing the database.</td></tr>
<tr><td>Actors:</td><td>System administrator, Committee Chairs</td></tr>
<tr><td>Related Use Cases:</td><td>updatePerson</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>The sysadmin must have privileges to modify users.</td></tr>
<tr><td>Postconditions:</td><td>The affected user’s profile must reflect their changed role.</td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table>
<tr><td>1. The sysadmin accesses the user management GUI.
2. The sysadmin enters a valid email for the user in question.
3. The user’s profile information is displayed.
4. The sysadmin chooses a role from the list for the user.
5. The sysdamin updates the role.
6. The role is changed as required.</td><td>2.1 The system processes the email and looks up the matching Person information for that email.
5.1 The system changes the user role in the database.</td></tr>
</table></td></tr>
<tr><td>Exception Conditions:</td><td>2. If the email is invalid, an error is thrown.</td></tr>
</table>

<table>
<tr><td>Use Case Name:</td><td>deletePersonRole</td></tr>
<tr><td>Scenario:</td><td>A person’s stored role is removed entirely</td></tr>
<tr><td>Triggering Event:</td><td>The sysadmin removes the role</td></tr>
<tr><td>Brief Description:</td><td>When a user is deleted by the sysadmin via the user management GUI, their associated role is also removed. Roles may also be removed as necessary through the GUI.</td></tr>
<tr><td>Actors:</td><td>System administrator, Committee Chairs</td></tr>
<tr><td>Related Use Cases:</td><td>deletePerson</td></tr>
<tr><td>Stakeholders:</td><td>All valid users</td></tr>
<tr><td>Preconditions:</td><td>The sysadmin must have privileges to modify users.</td></tr>
<tr><td>Postconditions:</td><td></td></tr>
<tr><td>Flow of Activities:</td><td>Actor System
<table>
<tr><td>1. The sysadmin accesses the user management GUI.
2. The sysadmin enters a valid email for the user in question.
3. The user’s profile information is displayed.
4. The sysadmin chooses to delete the user.</td><td>2.1 The system processes the email and looks up the matching Person information for that email.

4.1 When the delete command is given, the system removes the Person and their role from the database, or JUST the role, if required.</td></tr>
</table></td></tr>
<tr><td>Exception Conditions:</td><td>2. If the email is invalid, an error is thrown.</td></tr>
</table>

APPENDIX C – Example Code

The following code shows several functions that may run in the profile page.

Inserting System Values into Auth DB

```
Protected Sub systemValuesInsert(ByVal authSysID As Object)
       Try
            If conn.State = Data.ConnectionState.Closed Then conn.Open()
            cmd = New SqlCommand("validSystemsInsert", conn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@authpersonid", authSysID) 'authDB PK
            Select Case ViewState("sys")
                Case 1
                    cmd.Parameters.AddWithValue("@validsystemid", 1) 'CSB
                Case 2
                    cmd.Parameters.AddWithValue("@validsystemid", 2) 'Entropy
            End Select
            cmd.Parameters.AddWithValue("@systempersonid", authSysID)
            cmd.ExecuteNonQuery()
        Catch ex As Exception
            Me.uiMessage.Text = "Error on Sys values insert: " &
              Err.Description
        Finally
            conn.Close()
        End Try
    End Sub
```

Checking for User Existence in Auth DB

```
Protected Sub uiUserCheckBtn_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles uiUserCheckBtn.Click
        'First, run an SP to check if credentials exist in auth-db
        ViewState("id_check") = Me.uiUnivIDCheck.Text
        ViewState("email_check") = Me.uiUnivEmailCheck.Text
        If authDBCheck(ViewState("id_check"), ViewState("email_check")) Then
            pnlUserConfirm.Visible = True
            uiUnivIDCheck.Enabled = False
            uiUnivEmailCheck.Enabled = False
            uiUserCheckBtn.Enabled = False
        ElseIf sqlError = 1 Then
            Me.uiMessage.Text = "Error on Entropy update [SQL Error = 1]: " &
                     Err.Description
        Else
            pnlInsertUpdateForm.Visible = True
            pnlUserCheck.Visible = False
        End If
    End Sub

Function authDBCheck(ByVal univID As String, ByVal univEmail As String) As Boolean
        Dim exists As Boolean = False
        Try
            If conn.State = Data.ConnectionState.Closed Then conn.Open()
            cmd = New SqlCommand("uspPersonExistCheck", conn)
            cmd.CommandType = Data.CommandType.StoredProcedure
```

```
            cmd.Parameters.AddWithValue("@unividnum", univID)
            cmd.Parameters.AddWithValue("@emailuniv", univEmail)
            cmd.ExecuteNonQuery() 'Executing the query...

            dtrReader = cmd.ExecuteReader
            dtrReader.Read()
            If dtrReader.HasRows Then 'Did the query return any rows at all?
                exists = True 'Rows returned – this person exists
                lblUserCheckConfirmMsg.Text = "We have confirmed that " &
                     dtrReader("firstName") & " " & dtrReader("lastName") &
                     " is in our database.  Is this you?"
            Else
                exists = False
            End If
        Catch ex As Exception
            exists = False
            sqlError = 1
        Finally
            dtrReader.Close()
            conn.Close()
        End Try
        Return exists
End Function
```

Checking for User Existence in Primary DBs

```
Protected Sub btnUserYes_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles btnUserYes.Click
        Dim entExists = False
        Dim csbExists = False
        'The user has clicked "yes", to claim that they are who was found in the
        'auth-db.  Now we must find in which system they actually exist.
        Select Case ViewState("sys")
            Case 1
                If csbDBCheck(ViewState("id_check"), ViewState("email_check"))
                 Then 'If they exist in CSB...
                    uiErrorLoginMsg.Visible = True
                    uiErrorLoginLink.Visible = True
                    pnlUserConfirm.Visible = False
                    csbExists = True
                End If

                If Not csbExists Then 'Not in CSB? Check Entropy…
                    If entDBCheck(ViewState("id_check"), ViewState("email_check"))
                     Then 'If they exist in Entropy...
                        If createCSBAccount(ViewState("id_check"),
                            ViewState("email_check")) Then
                            Me.uiMessage.Text = "Your account has been created!
                                  Please log in."
                        Else
                            Me.uiMessage.Text = "Error on account duplication from
                                  Entropy: " & Err.Description
                        End If
                    Else
                        Exit Sub
                    End If
                End If
```

```
            Case 2
                If entDBCheck(ViewState("id_check"), ViewState("email_check"))
                 Then 'If they exist in Entropy...
                    uiErrorLoginMsg.Visible = True
                    uiErrorLoginLink.Visible = True
                    pnlUserConfirm.Visible = False
                    entExists = True
                End If

                If Not entExists Then 'Not in Entropy? Check CSB…
                    If csbDBCheck(ViewState("id_check"), ViewState("email_check"))
                     Then 'If they exist in CSB...
                        If createEntAccount(ViewState("id_check"),
                            ViewState("email_check")) Then
                            Me.uiMessage.Text = "Your account has been created!
                                  Please log in."
                        Else
                            Me.uiMessage.Text = "Error on account duplication from
                                  CSB: " & Err.Description
                        End If
                    Else
                        Exit Sub
                    End If
                End If
        End Select
End Sub

Function entDBCheck(ByVal univID As String, ByVal univEmail As String) As Boolean
        Dim exists As Boolean = False
        Try
            If eConn.State = Data.ConnectionState.Closed Then eConn.Open()
            cmd = New SqlCommand("uspPersonExistCheck", eConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@unividnum", univID)
            cmd.Parameters.AddWithValue("@emailuniv", univEmail)
            cmd.ExecuteNonQuery() 'Executing the query...

            dtrReader = cmd.ExecuteReader
            dtrReader.Read()
            If dtrReader.HasRows Then 'Did the query return any rows at all?
                exists = True 'Rows were returned!  This person must exist...
            Else
                exists = False
            End If
        Catch ex As Exception
            Me.uiMessage.Text = "Error on Entropy database check: " &
                    Err.Description
            exists = False
            sqlError = 1
        Finally
            dtrReader.Close()
            eConn.Close()
        End Try
        Return exists
End Function

Function csbDBCheck(ByVal univID As String, ByVal univEmail As String) As Boolean
```

```
        Dim exists As Boolean = False
        Try
            If cConn.State = Data.ConnectionState.Closed Then cConn.Open()
            cmd = New SqlCommand("uspPersonExistCheck", cConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@univIdnum", univID)
            cmd.Parameters.AddWithValue("@emailuniv", univEmail)
            cmd.ExecuteNonQuery() 'Executing the query...

            dtrReader = cmd.ExecuteReader
            dtrReader.Read()
            If dtrReader.HasRows Then 'Did the query return any rows at all?
                exists = True 'Rows were returned!  This person must exist...
            Else
                exists = False
            End If
        Catch ex As Exception
            Me.uiMessage.Text = "Error on CSB database check: " & Err.Description
            exists = False
            sqlError = 1
        Finally
            dtrReader.Close()
            cConn.Close()
        End Try
        Return exists
End Function
```

CSB & Entropy Account Duplications

```
Function createEntAccount(ByVal univID As String, ByVal univEmail As String) As
Boolean
        Dim duplicated As Boolean = False
        Try
            'Opening a connection to CSB, getting the account info from
              uspPersonProfileDuplicate, and saving it into variables.
            If cConn.State = Data.ConnectionState.Closed Then cConn.Open()
            cmd = New SqlCommand("uspPersonProfileDuplicate", cConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@univIdnum", univID)
            cmd.Parameters.AddWithValue("@emailuniv", univEmail)
            cmd.ExecuteNonQuery()
            dtrReader = cmd.ExecuteReader
            dtrReader.Read()
            Dim dupID = dtrReader("personID")
            Dim dupUnivID = dtrReader("univIDNum")
            Dim dupFN = dtrReader("firstName")
            Dim dupLN = dtrReader("lastName")
            Dim dupEmailUniv = dtrReader("emailUniv")
            Dim dupEmailDes = dtrReader("emailDesired")
            dtrReader.Close()
            cConn.Close()

            'Opening a connection to Entropy; inserting the variables into the SP.
            If eConn.State = Data.ConnectionState.Closed Then eConn.Open()
            cmd = New SqlCommand("uspPersonUpdate", eConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@personID", dupID)
```

```
            cmd.Parameters.AddWithValue("@emailuniv", dupEmailUniv)
            cmd.Parameters.AddWithValue("@emailDesired", dupEmailDes)
            cmd.Parameters.AddWithValue("@firstname", dupFN)
            cmd.Parameters.AddWithValue("@lastname", dupLN)
            cmd.Parameters.AddWithValue("@univIDNum", dupUnivID)
            cmd.Parameters.AddWithValue("@clicker", "")
            cmd.ExecuteNonQuery() Student role is auto-inserted through the SP.
            duplicated = True
        Catch ex As Exception
            Me.uiMessage.Text = "Error on account duplication from CSB: " &
                    Err.Description
            duplicated = False
            sqlError = 1
        Finally
            cmd.Dispose()
            If eConn.State = Data.ConnectionState.Open Then eConn.Close()
        End Try
        Return duplicated
End Function

Function createCSBAccount(ByVal univID As String, ByVal univEmail As String) As
Boolean
        Dim duplicated As Boolean = False
        Try
            'Opening a connection to Entropy, getting the account info from
              uspPersonProfileDuplicate, and saving it into variables.
            If eConn.State = Data.ConnectionState.Closed Then eConn.Open()
            cmd = New SqlCommand("uspPersonProfileDuplicate", eConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@unividnum", univID)
            cmd.Parameters.AddWithValue("@emailuniv", univEmail)
            cmd.ExecuteNonQuery()
            dtrReader = cmd.ExecuteReader
            dtrReader.Read()
            Dim dupID = dtrReader("personID")
            Dim dupUnivID = dtrReader("univIDNum")
            Dim dupFN = dtrReader("firstName")
            Dim dupLN = dtrReader("lastName")
            Dim dupEmailUniv = dtrReader("emailUniv")
            Dim dupEmailDes = dtrReader("emailDesired")
            dtrReader.Close()
            eConn.Close()

            'Opening a connection to CSB, and inserting the variables into the SP.
            If cConn.State = Data.ConnectionState.Closed Then cConn.Open()
            cmd = New SqlCommand("uspPersonUpdate", cConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@personID", dupID)
            cmd.Parameters.AddWithValue("@emailuniv", dupEmailUniv)
            cmd.Parameters.AddWithValue("@emailDesired", dupEmailDes)
            cmd.Parameters.AddWithValue("@firstname", dupFN)
            cmd.Parameters.AddWithValue("@lastname", dupLN)
            cmd.Parameters.AddWithValue("@univIDNum", dupUnivID)
            cmd.ExecuteNonQuery() 'Student role is auto-inserted through the SP.
            duplicated = True
        Catch ex As Exception
            Me.uiMessage.Text = "Error on account duplication from Entropy: " &
```

```
                    Err.Description
            duplicated = False
            sqlError = 1
        Finally
            cmd.Dispose()
            If cConn.State = Data.ConnectionState.Open Then cConn.Close()
        End Try
        Return duplicated
End Function
```

Inserting into Entropy (after Auth DB Insert)

```
Protected Sub entropyInsert(ByVal entID As Object)
        Try
            If eConn.State = Data.ConnectionState.Closed Then eConn.Open()
            cmd = New SqlCommand("uspPersonUpdate", eConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@personID", entID)
            cmd.Parameters.AddWithValue("@emailuniv",
                    Me.uiUnivEmail.Text.ToLower)
            If Me.uiDesiredEmail.Text.Trim = "" Then Me.uiDesiredEmail.Text =
                    Me.uiUnivEmail.Text
            cmd.Parameters.AddWithValue("@emailDesired",
                    Me.uiDesiredEmail.Text.ToLower)
            cmd.Parameters.AddWithValue("@firstname", Me.uiFirstName.Text.Trim)
            cmd.Parameters.AddWithValue("@lastname", Me.uiLastName.Text.Trim)
            cmd.Parameters.AddWithValue("@univIDNum", Me.uiUnivID.Text)
            cmd.Parameters.AddWithValue("@clicker", Me.uiClicker.Text)
            cmd.ExecuteNonQuery()
        Catch ex As Exception
            Me.uiMessage.Text = "Error on Entropy insert: " & Err.Description
        Finally
            eConn.Close()
        End Try
    End Sub
```

Inserting into CSB (after Auth DB Insert)

```
Protected Sub csbInsert(ByVal csbID As Object)
        Try
            If cConn.State = Data.ConnectionState.Closed Then cConn.Open()
            cmd = New SqlCommand("uspPersonUpdate", cConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@personID", csbID)
            cmd.Parameters.AddWithValue("@emailuniv",
                    Me.uiUnivEmail.Text.ToLower)
            If Me.uiDesiredEmail.Text.Trim = "" Then Me.uiDesiredEmail.Text =
                    Me.uiUnivEmail.Text
            cmd.Parameters.AddWithValue("@emailDesired",
                    Me.uiDesiredEmail.Text.ToLower)
            cmd.Parameters.AddWithValue("@firstname", Me.uiFirstName.Text.Trim)
            cmd.Parameters.AddWithValue("@lastname", Me.uiLastName.Text.Trim)
            cmd.Parameters.AddWithValue("@univIDNum", Me.uiUnivID.Text)
            cmd.ExecuteNonQuery()
        Catch ex As Exception
            Me.uiMessage.Text = "Error on CSB insert: " & Err.Description
```

```
        Finally
            eConn.Close()
        End Try
End Sub
```

Updating Entropy (after Auth DB Update)

```
Protected Sub entropyUpdate(ByVal entID As Object)
        Try
            If eConn.State = Data.ConnectionState.Closed Then eConn.Open()
            cmd = New SqlCommand("uspPersonUpdate", eConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@personID", entID)
            cmd.Parameters.AddWithValue("@emailuniv",
                    Me.uiUnivEmail.Text.ToLower)
            If Me.uiDesiredEmail.Text.Trim = "" Then Me.uiDesiredEmail.Text =
                    Me.uiUnivEmail.Text
            cmd.Parameters.AddWithValue("@emailDesired",
                    Me.uiDesiredEmail.Text.ToLower)
            cmd.Parameters.AddWithValue("@firstname", Me.uiFirstName.Text.Trim)
            cmd.Parameters.AddWithValue("@lastname", Me.uiLastName.Text.Trim)
            cmd.Parameters.AddWithValue("@clicker", Me.uiClicker.Text)
            cmd.ExecuteNonQuery()
        Catch ex As Exception
            Me.uiMessage.Text = "Error on Entropy update: " & Err.Description
        Finally
            eConn.Close()
        End Try
End Sub
```

Updating CSB (after Auth DB Update)

```
Protected Sub csbUpdate(ByVal csbID As Object)
        Try
            If cConn.State = Data.ConnectionState.Closed Then cConn.Open()
            cmd = New SqlCommand("uspPersonUpdate", cConn)
            cmd.CommandType = Data.CommandType.StoredProcedure
            cmd.Parameters.AddWithValue("@personID", csbID)
            cmd.Parameters.AddWithValue("@emailuniv",
                    Me.uiUnivEmail.Text.ToLower)
            If Me.uiDesiredEmail.Text.Trim = "" Then Me.uiDesiredEmail.Text =
                    Me.uiUnivEmail.Text
            cmd.Parameters.AddWithValue("@emailDesired",
                    Me.uiDesiredEmail.Text.ToLower)
            cmd.Parameters.AddWithValue("@firstname", Me.uiFirstName.Text.Trim)
            cmd.Parameters.AddWithValue("@lastname", Me.uiLastName.Text.Trim)
            cmd.ExecuteNonQuery()
        Catch ex As Exception
            Me.uiMessage.Text = "Error on CSB update: " & Err.Description
        Finally
            cConn.Close()
        End Try
End Sub
```

APPENDIX D – Survey Materials

Authentication Survey 1

Your participation is voluntary. You may stop at any time or refuse to answer any question without being treated any differently by the researcher. Data will be kept secure once it is in the PI's possession; however the PI cannot guarantee security during transmission of data due to key logging and other spyware technology that may exist on any computer used by the subject.

1) I am:
[] A student
[] A faculty member
[] A staff member

2) On average, how often do you sign into Entropy?
[] Daily
[] Several times a week
[] Several times a month
[] Only a few times a semester
[] Almost never

3) On average, how often do you log into myCSB (the CSB Portal)?
[] Daily
[] Several times a week
[] Several times a month
[] Only a few times a semester
[] Almost never

4) OpenID allows users to log in to applications (i.e., Entropy, myCSB) using the username + password they set up with another Web provider. Some of these Web providers are listed below. You might already have an OpenID account and not be aware that you do. Do you already have an existing account with any of the following providers? (Mark all that apply)
[] Google
[] Facebook
[] Yahoo!
[] flickr
[] Myspace
[] Livejournal

5) How would you prefer to sign into myCSB and Entropy services?
[] I would prefer to just keep using the same username and password (i.e. UNCW id and customized password)
[] I would prefer to log in with my username + password from a Web provider, like Facebook or Google
[] I would like to have both of the above options available

6) If you had to sign into myCSB and Entropy using a Web provider, which would you most prefer?
[] Google
[] Facebook
[] Yahoo!
[] flickr
[] Myspace
[] Livejournal

Survey 1 Results

1. I am:

	Response Total	Response Percent	Points	Avg
A student	17	68%	n/a	n/a
A faculty member	8	32%	n/a	n/a
A staff member	0	0%	n/a	n/a
Total Respondents	25	100%		

2. On average, how often do you sign into Entropy?

	Response Total	Response Percent	Points	Avg
Daily	9	36%	n/a	n/a
Several times a week	9	36%	n/a	n/a
Several times a month	1	4%	n/a	n/a
Only a few times a semester	4	16%	n/a	n/a
Almost never	2	8%	n/a	n/a
Total Respondents	25	100%		

3. On average, how often do you log into myCSB (the CSB Portal)?

	Response Total	Response Percent	Points	Avg
Daily	0	0%	n/a	n/a
Several times a week	4	16%	n/a	n/a
Several times a month	7	28%	n/a	n/a
Only a few times a semester	11	44%	n/a	n/a
Almost never	3	12%	n/a	n/a
Total Respondents	25	100%		

4. OpenID allows users to log in to applications (i.e., Entropy, myCSB) using the username + password they set up with another web provider. Some of these web providers are listed below. You might already have an OpenID account and not be aware that you do. Do you already have an existing account with any of the following providers? (Mark all that apply)

	Response Total	Response Percent	Points	Avg
Google	17	68%	n/a	n/a
Facebook	19	76%	n/a	n/a
Yahoo!	6	24%	n/a	n/a
flickr	0	0%	n/a	n/a
Myspace	2	8%	n/a	n/a
Livejournal	0	0%	n/a	n/a
Total Respondents	25			

5. How would you prefer to sign into myCSB and Entropy services?

	Response Total	Response Percent	Points	Avg
I would prefer to just keep using the same username and password (i.e. UNCW id and customized password)	13	52%	n/a	n/a
I would prefer to log in with my username + password from a web provider, like Facebook or Google	2	8%	n/a	n/a
I would like to have both of the above options available	10	40%	n/a	n/a
Total Respondents	25	100%		

6. If you had to sign into myCSB and Entropy using a web provider, which would you most prefer?

	Response Total	Response Percent	Points	Avg
Google	16	70%	n/a	n/a
Facebook	7	30%	n/a	n/a
Yahoo!	0	0%	n/a	n/a
flickr	0	0%	n/a	n/a
Myspace	0	0%	n/a	n/a
Livejournal	0	0%	n/a	n/a
Total Respondents	23	100%		
(skipped this question)	2			

Authentication Survey 2

Your participation is voluntary. You may stop at any time or refuse to answer any question without being treated any differently by the researcher. Data will be kept secure once it is in the PI's possession; however the PI cannot guarantee security during transmission of data due to key logging and other spyware technology that may exist on any computer used by the subject.

1) I am affiliated with:
[] RAP
[] CEN
[] Advisory Board

2) OpenID allows users to log in to applications (i.e., CSB Portal) using the username + password they set up with another web provider. Some of these web providers are listed below. You might already have an OpenID account and not be aware that you do. Do you already have an existing account with any of the following providers? (Mark all that apply)
[] Google
[] Facebook
[] Linkedin
[] Yahoo!

3) How would you prefer to sign into the CSB Portal?
[] I would prefer to use the username and password that is assigned to me by the University
[] I would prefer to log in with my username + password from a web provider, like Linkedin or Facebook
[] I would like to have both of the above options available

4) If you had to sign into the CSB Portal using a web provider, which would you most prefer?
[] Google
[] Facebook
[] Linkedin
[] Yahoo!

*** Note: Survey 2 results are in chapter 7.1.**