A Strong Authentication Mechanism for Consumer-Facing Online Transactions

Dario Anibal Marra
MIT - Department of EECS
Chisec Group
May 16, 2005

Abstract

Most consumer-facing online applications such as banking and e-commerce rely primarily on single-factor authentication schemes to authenticate users. Such schemes, while easy to use, suffer from easily exploitable security vulnerabilities that cost business billions of dollars per year. While alternative authentication mechanisms such as biometrics and two-factor schemes provide increased security, neither is currently viable for the consumer market. This project proposes an economically practical and user-friendly two-factor authentication mechanism that uses a familiar end-user device—the cell phone—as the second factor in authentication. The system implemented in this project includes a server component and a client component. The server piece is an enterprise-grade authentication server that features pluggable authenticators, configuration management, and identity management. The client piece consists of software that runs locally on a Java-enabled cell phone to generate passcodes for multiple services.

I. Motivation

User authentication is a ubiquitous process in the modern Internet era. From online banking to online credit management to e-commerce, authentication is necessary to establish the identity of a given user. Although authentication is crucial in such applications, the vast majority of online sites use authentication methods that provide inadequate security and are susceptible to various attacks. The standard authentication mechanism used by nearly all applications is the single-factor authentication scheme. In this scheme, the user has a static user id and a static password that he uses to authenticate to the application. Although it is simple for developers to implement and simple for users to use, the single-factor authentication scheme suffers from serious security vulnerabilities. The two major attacks against the single-factor authentication scheme are keyboard logging and phishing.

Keyboard logging is the process via which private user information is collected directly on a personal computer by recording all keystrokes inputted by the user. The prototypical keyboard logger is a malicious program that runs on a user's machine without the user's knowledge, collecting all keystrokes typed by that user. This collected information can then be periodically offloaded to an adversary. With such information, an adversary can gain access to, among a myriad of other things, the credentials necessary to access online sites.

Whereas keyboard logging usually requires the "infection" of a target host machine with the keyboard logger program, phishing requires no such installation of software. For this reason it is becoming an increasingly popular form of obtaining personal information from a user. In simple terms, phishing involves spoofing a legitimate website in an attempt to obtain user information. Typical phishing attacks send users a spoofed email that provides a link to the purported site and instructions to update his account. Unbeknownst to the user, though, the website linked to in the email is under a malicious adversary's control, and is simply made to look like "the real thing." The user clicks on the link, is presented with what he believes to be the real website, and enters his credentials and other personal information. The adversary now possesses the user's credentials and can use those credentials as he sees fit.

Keyboard logging and phishing are two of the most prevalent schemes for online identity theft. Once an adversary obtains a known set of credentials for a given user, he can use those credentials to authenticate himself as the user and perform any number of malicious acts: modify account information, transfer money from one account to another, open new credit card accounts, obtain credit card numbers, and gather personal information, to name a few. While such techniques may seem trivially simple to implement and detect, the damage they cause is far from trivial: a recent Gartner report puts the cost of online identity theft due to keyboard logging and phishing alone at over $12 *billion* over the past year, and the problem is growing at an alarming rate [1]. Clearly, an alternative to the single-factor authentication scheme needs to be employed.

II. Alternative Solutions

The growing problem with single-factor authentication schemes has led to new alternative solutions. One class of authentication schemes that is making its way into the online consumer is biometrics. These schemes attempt to authenticate users by reading unique identifiers such as fingerprints or iris patterns. While such techniques are prevalent in areas where high security is required, their success in the online consumer space seems doubtful. First and foremost, such techniques are expensive. A sampling of USB-based fingerprint reader solutions from large manufacturers such as Sony, Fujitsu, and Microsoft, for example, places the average value of such devices at $100 to $150. Additionally, such products tend to produce false rejections, thereby denying a legitimate user access [2]. More importantly, though, there are privacy considerations that must be taken into account. A typical user will likely be uneasy about distributing his fingerprints or iris patterns to online merchants. Some proposals have been made to address the issue of privacy [3], but implementation hurdles make the future of biometrics as the standard for online authentication uncertain.

A second type of authentication that is commonly used in business environments is the two-factor authentication scheme. In typical two-factor schemes, the user has a static user id and pin but a dynamic password that changes either at each login or after a set time period. Because the password is dynamic, two-factor authentication schemes are far more impervious to keyboard logging and phishing than single-factor schemes; Microsoft, for example, arrives at the same conclusion in [4]. Popular two-factor

authentication schemes include the SecurID token by RSA Security and similar offerings by VeriSign, SafeNet, CRYPTOCard, Rainbow Technologies, and others. These products typically consist of a backend software piece and hardware tokens that are distributed to users. These tokens generate pseudorandom number sequences called *passcodes* that are used to form the password; such sequences change after a short time period (30 seconds, for example), and thus a given password is only valid for that short time period. Two-factor authentication schemes have found their niche in large companies that need to authenticate employees into the company intranet via the Internet. Their adoption in the general consumer space, however, seems more doubtful. As with biometrics, such systems tend to be expensive, with single tokens costing over $50 a piece. These systems are often difficult to administer, since tokens must somehow be tied in the backend to a particular user. Finally, they suffer greatly in terms of usability: a user that has to keep track of `n` tokens for `n` different e-commerce sites will quickly become frustrated and overwhelmed. A recent Gartner survey, for example, shows that lack of usability hinders current two-factor schemes from being accepted within the consumer market [5].

A slightly different flavor of the two-factor authentication scheme is to use the cell phone as a delivery mechanism for the passcode. Such a system is described by Wu et al in [6], and is offered commercially in products such as RSA Security's RSA Mobile. In this system, when a user attempts to authenticate, the passcode is sent to that user as an SMS message on his cell phone. The user then authenticates using his user id, pin, and this passcode. While this scheme resolves to some degree the problems with traditional two-factor systems, it suffers from a number of issues that has prevented widespread adoption. If a user does not have reception, for example, he cannot authenticate. Even if the user does have reception, cell phone transmissions are far from perfect, and text messages can be delayed or lost.

While none of these schemes serve as the "silver bullet," the gravity of the identity theft problem is so great that online companies are nonetheless offering them to consumers. AOL, for instance, allows users to authenticate to their servers using a SecurID-style implementation, but at an increased cost to the user [7]. Banks and other financial institutions have been pushing for their own two-factor schemes, as well [1]. It is unlikely, though, that users will pay extra for a cumbersome system most do not think they truly need. After all, typical users expect security to "just work."

III. Overview of the SecureCell System

The SecureCell system, as the software developed for this project is called, is a two-factor authentication scheme that tackles many of the problems previously mentioned by using the ubiquitous cell phone as the medium for passcode generation. However, instead of passcodes being delivered to the cell phone via SMS, the cell phone itself generates the passcodes; that is, software running locally on the cell phone is responsible for generating passcodes. The user can then authenticate to the backend system using his user id, pin, and this passcode.

Abstractly, the SecureCell system is a prototypical client-server application. As Figure 1 shows, the system can be divided into three basic components: (1) the SecureCell client; (2) the SecureCell server; and (3) a middle-layer plugin component.

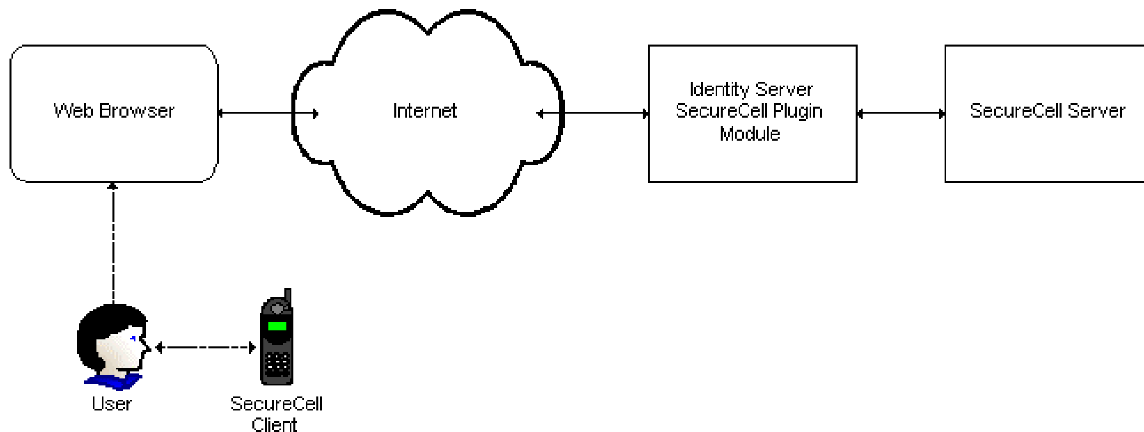


Figure 1 – Abstract view of the SecureCell system.

The SecureCell client is the piece that runs locally on the cell phone and generates the passcodes needed for authentication; the user interacts directly with the client on the cell phone in order to authenticate. The SecureCell server processes authentication attempts for users. Finally, the Identity Server plugin module is an arbitrary middle-layer component that exposes the login page via which the user authenticates. Although not strictly necessary, it provides several useful services such as URL rewriting and cookie management that are useful in creating a realistic test environment for the system. The following sections will discuss the server and the client pieces in more detail.

IV. The SecureCell Server

The SecureCell server is responsible for processing user authentication requests. It is a full-featured, enterprise-grade server written in the Java programming language. The basic goals behind the design of the server are flexibility, performance, and the ability to integrate into existing application environments. That is, the server should be flexible in terms of the authentication algorithms it supports; it should be capable of handling a large number of concurrent authentication requests; and it should offer the necessary hooks and translation pieces to tie in with existing applications that use other authentication mechanisms. To address these goals, the server implementation follows a modular design, as shown in Figure 2.

Server Front-End

The server itself can be divided into two basic sections: (1) the server front-end, and (2) the server back-end. The server front-end exposes the protocol that incoming authentication requests and outgoing authentication responses should use to authenticate to the back-end. This distinction of a server front-end and a server back-end effectively

decouples the actual authentication process from the protocol translation process. Perhaps more importantly, though, it allows one to leverage existing protocols that can be chosen based on the needs of the overall application. In the current system, an HTTP front-end was chosen due to ease of implementation. Specifically, a Java Servlet running on an Apache HTTP server instance receives incoming requests for authentication (that is, it receives the user id, pin, and passcode with which to authenticate in a POST request, for example). This HTTP front-end then delegates processing to the server back-end to yield the result of the authentication attempt, and transmits the result back to the requesting client as an HTTP response. The back-end server, thus, is protocol-independent. If a secure connection is needed during authentication, an HTTPS front-end could be used. If the high performance is required, a UDP front-end can be employed. If the server needs to be a direct replacement for the SecurID ACE Server, a front-end can be deployed that mimics the SecurID protocol. The protocol chosen is irrelevant to the operation of the back-end server.

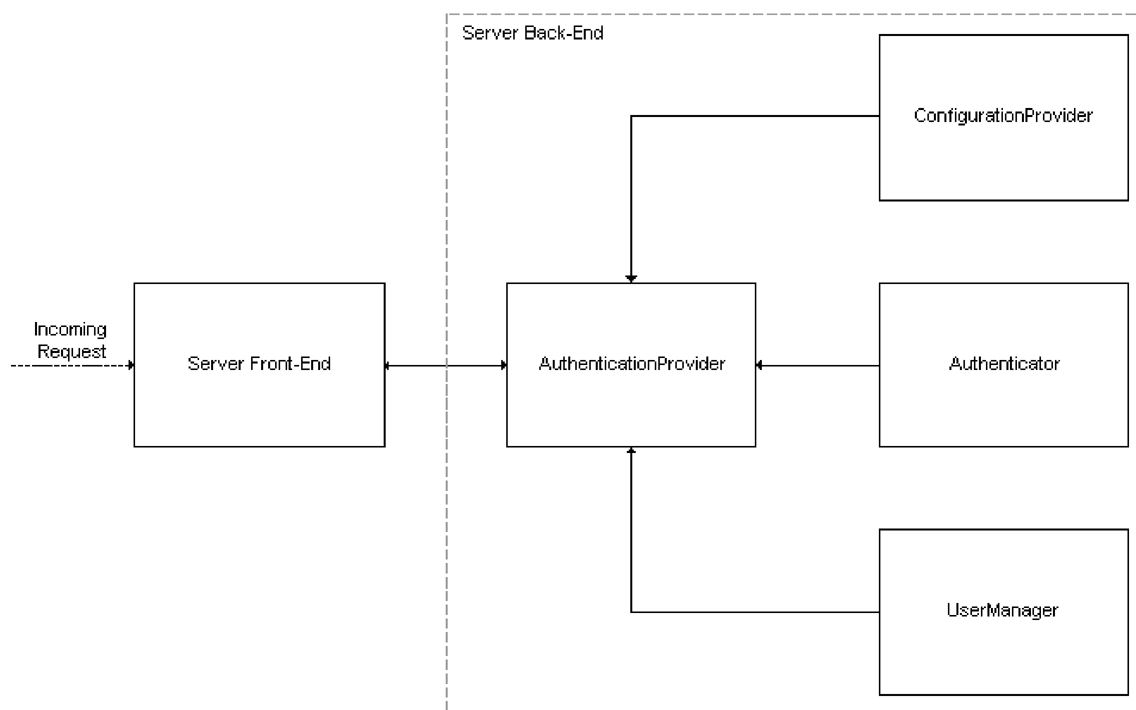


Figure 2 – Server architecture showing the server as a decomposition of logical modules.

The decoupling of the front-end and the back-end also has the advantage that one need not re-implement complex and advanced features. For example, load-balancing clustering and fail-over clustering are powerful features that an enterprise-grade server should support, yet are not trivial to implement. However, by dividing the server into a front-end and a back-end piece, such features are essentially "inherited" by the choice of an appropriate front-end. One could, for example, set up a cluster of authentication servers using appropriate HTTP web servers as the front-end and introduce a load-balancer between the clients and the servers to yield a load-balanced authentication source.

Server Back-End

The back-end server follows a similar process of modularization to ensure the maximum flexibility in implementation. The back-end is composed of three primary modules that operate within the context of an `AuthenticationProvider` to handle incoming authentication requests from the front-end. First, there is a configuration module from which server configuration parameters are obtained. The configuration module is completely back-end independent, which means that the actual configuration data may reside in any backend source (LDAP, SQL, XML, etc.). Secondly, there is the identity management module that interfaces with a backend source to provide access to user configuration parameters. This user data includes things such as the user id, the user pin, and other parameters necessary to authenticate a user. As with the configuration module, the identity management module is back-end independent. Both the configuration module and the identity management module use XML as the backend source in the current implementation. Finally, there are the authenticator modules that handle the actual authentication process for an incoming request. Each authenticator essentially encapsulates some authentication algorithm. One authenticator implementation may, for example, use a counter-based two-factor authentication algorithm while another uses a time-based two-factor authentication algorithm.

In addition to being highly modular, the back-end server also includes common features found in most well designed authentication servers such as locking out of user accounts after a certain number of failed login attempts and optionally locking a user account for a certain period of time following a successful authentication attempt to prevent replay attacks (pertinent for time-based algorithms).

V. The HOTP Algorithm

On both the client and the server, the choice of algorithm for passcode generation is essentially arbitrary, so long as it provides adequate security and can be used in a user-friendly manner (this is particularly relevant for the client). Most two-factor schemes use time-based or challenge/response algorithms. In this implementation, challenge/response algorithms were considered too cumbersome for a typical user to use, and time-based algorithms were omitted simply because they are largely proprietary and unavailable to the public. Instead, this implementation chose a counter-based algorithm called HOTP that is relatively easy to implement and met the necessary usability requirements. The algorithm is described in detail in [8].

As with most two-factor authentication algorithms, the HOTP algorithm requires a strong shared secret between the client and the server. Each client has a unique shared secret, typically 128 bits or 160 bits in length. The shared secret is combined with a monotonically increasing counter, also shared between the client and the server, to generate the current passcode. The actual HOTP algorithm is relatively simple to understand. First, a SHA-1 HMAC generator is initialized using the shared secret. Then the HMAC of the current counter, or *moving factor*, is computed. Next, through a process called *dynamic truncation*, certain bytes are extracted from the HMAC. Finally, these

bytes are taken modulo 10^n, where n is the number of digits desired in the passcode, to produce the current passcode. Custom, lightweight implementations of both the SHA-1 and HMAC algorithms were implemented in accordance to RFCs 3174 and 2104, described in [9] and [10] respectively.

In order for a client to authenticate to a server, both must generate the same passcode. Specifically, assuming that the server has already distributed the shared secret to the client, the client counter and the server counter must be synchronized. When the counters are not synchronized, a process called *resynchronization* must occur. The HOTP algorithm has two basic mechanisms to resynchronize the server with the client. The most straightforward method is for the client to simply send the counter value over to the server. The server would merely need to verify that the new counter is greater than the current counter. The second method is for the server to maintain a *look-ahead window* of future passcodes. If the client provides a passcode that lies within this window, the server will ask the user to generate the next passcode and send it to the server. If two consecutive passcodes match, then the server will resynchronize. While the first of these two resynchronization methods is easier to implement, this project chose to use a look-ahead window to follow industry convention.

Assuming that the underlying SHA-1 algorithm is "secure" (i.e., it adheres to certain well-known tenets of a strong hashing function), the security of the HOTP algorithm is dependent exclusively on the shared secret being kept secret. That is, even with a large number of successful and unsuccessful passcodes and counter values, the best an adversary can hope to do is a brute-force attack [8]. The probability of success in such a brute force attack is approximately

$$\Pr\{success\} = \frac{wa}{10^n}$$

where w is the size of the look-ahead resynchronization window, a is the number of authentication attempts before an account is locked, and n is the number of digits in a passcode. In the current implementation, $w=10$, $a=3$, and $n=6$, which gives a probability of success of 3×10^{-5}.

VI. The SecureCell Client

The SecureCell client runs locally on Java-enabled cell phones to generate passcodes that the user can use to authenticate to one or more SecureCell servers. Although perhaps the simplest of the modules in terms of basic functionality, the client is the most difficult to implement due to fact that users must interact directly with it. As such, one runs into the perennial problem of security software: how to hide the complexity of the underlying security algorithms within a user-friendly design. The client design, thus, adheres to two main goals. First and most importantly, it cannot suffer with regards to usability. The user interface must strive to be friendly enough for the novice user and powerful enough for the seasoned user; these requirements define a *good* user-interface. Secondly, the client must be flexible enough to accommodate multiple *services*

(i.e., a particular online application instance), each of which may use different algorithms for passcode generation. After several iterations, both of these goals were largely met in the current implementation.

J2ME Platform and Cell Phone Limitations

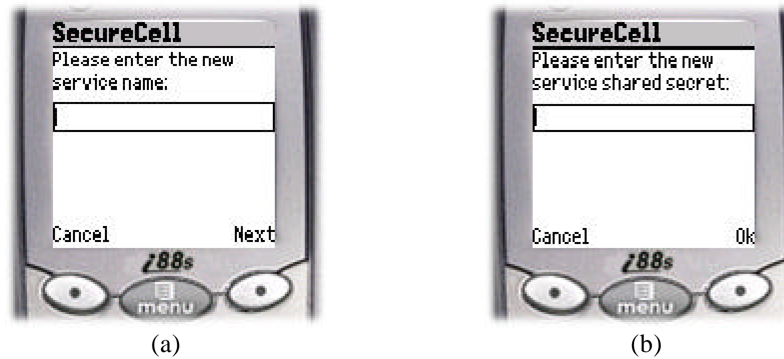
Modern cell phones typically run a “standard” operating system or application environment such as Windows CE, Java, or Symbian, or they use a vendor-specific application environment. The cell phone used for development and testing in this project, the Motorola i88s, is a Java-enabled cell phone that runs applications written in Java. The J2ME Mobile Information Device Profile (MIDP) 1.0 defines exactly what is available to the cell phone Java application developer [11]. Overall, the MIDP1.0 is very restrictive in the components it provides for building a user interface; only the most basic widgets are available. The event handling operations available for these components are even more restrictive. The major challenge in designing a good UI, then, was how to transform the two or three basic widgets available into a powerful and user-friendly implementation.

Further complicating the design of a solid user interface were the inherent limitations in the cell phone itself. Unlike modern PC displays, the cell phone screen real estate is very limited. Thus, the user interface must be very succinct and concise. Likewise, a cell phone does not have the rich input/output capabilities of modern PCs. Something that takes a few seconds to type on a keyboard might be quite challenging and cumbersome on a cell phone key pad. Finally, instead of rich menus and multi-windowed environments, cell phones can typically display a single *screen* at a time, and navigation among screens using the two or three *command buttons* is difficult. All these limitations had dramatic effects on what sort of user interface could be implemented effectively.

Basic Client Operations

One of the advantages that the SecureCell client offers over traditional two-factor authentication schemes is that the same end-user device—the cell phone—can handle multiple services. That is, if Bank A and Bank B both use the SecureCell system, the same cell phone can support both services. In a system such as SecurID, on the other hand, one token is required for every service. The client, then, must provide three basic operations: (1) adding a new service, (2) deleting an existing service, and (3) generating passcodes for a particular service.

Adding a new service is likely to be the most complicated step for a new user, as it currently involves a fair amount of typing. The current UI features a “wizard-like” interface that first asks the user to enter the unique name for the service and then shared secret, as shown in Figures 3a and 3b.



(a) (b)

Figure 3 – Specifying the service name (a) and shared secret (b) when creating a new service. Note how the command button labels guide the user from one step to the next.

Previous iterations in this process had a central window from which the user would first navigate to a screen to enter the service name, navigate back to the central screen, navigate to a screen to enter the service shared secret, navigate back to the central screen, and finally create the new service. This process, however, proved difficult to use, and was thus abandoned in favor of the current wizard-style interface.

The most daunting task for any user is undoubtedly entering the shared secret. While a later section will propose different mechanisms to accomplish this, the current system requires the user to type in the shared secret directly. With a key length of 128 or 160 bits, this may require as many as 40 different keystrokes if using numeric secrets and slightly more (on average) if using alphanumeric secrets. Errors are likely to occur, given that it is tedious to type on the keypad and sometimes difficult to read the screen. Several approaches such as chunking and checksums were investigated to mitigate possible errors, but each had their drawbacks. Chunking the input into groups of 5 or so characters, for example, was complicated to use given the cell phone UI limitations, while checksums are likely to be a foreign concept to most users and will thus be ignored. Instead of requiring the user to verify the shared secret directly, the current system implements a simple credit card algorithm that can detect any single bit error and any single transposition. If this internal checksum fails, the system asks that the user verify the shared secret that was just entered.

Service deletion is a relatively straightforward process. First, a list of available services is presented to the user, as shown in Figure 4. A user selects the service to delete, confirms the selection, and then the service is deleted.

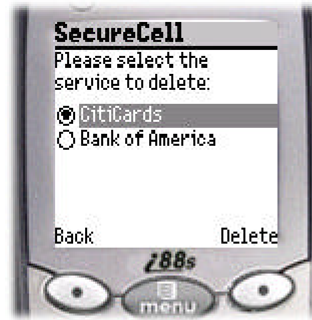


Figure 4 – List of services available to be deleted.

The most complicated aspect of deleting a service is realizing that the service list does not implicitly select an item when it is highlighted. Thus, actually picking a service to delete is three-step process: first, the desired list item is highlighted; second, the highlighted item is selected using a non-standard button; and finally, the service can be deleted by pressing the "Delete" command button. This three-step selection process is perhaps one of the most serious drawbacks of the MIDP1.0 specification, and is caused by the lack of UI components and deficiencies in the event handling capabilities. More perplexing, though, is the fact that MIDP2.0, which claims to have "greatly improved UI features" is subject to the same flaw [12]. It is possible, though, that the issue might only affect the Motorola i88s; other cell phones were not used during testing and may behave differently.

It should be noted that service deletion is risky when using a counter-based algorithm. Although resynchronization at the server is relatively simple, it is quite a bit more difficult to resynchronize a client from a server. Thus, the deletion confirmation message must be very forthcoming about the dangers of deleting an existing service. For time-based algorithms, this would not be an issue.

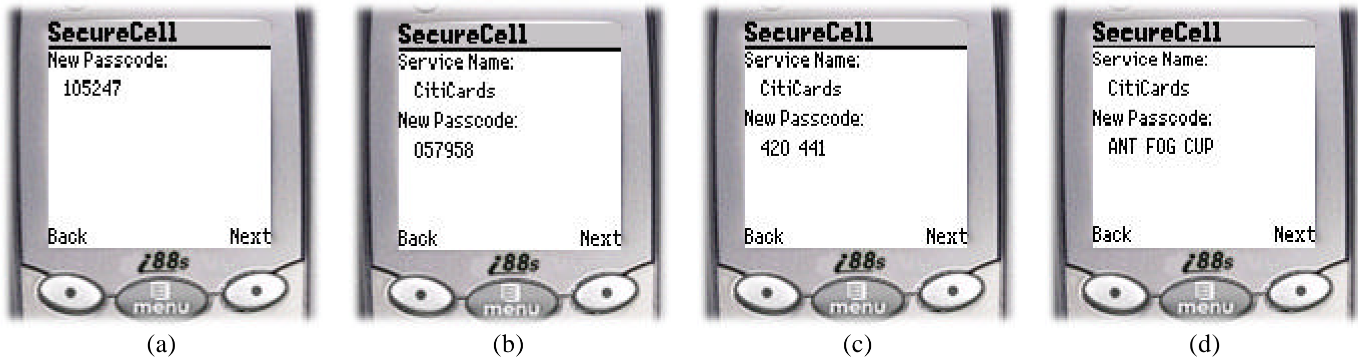
Passcode Generation

The primary function of the client is to generate passcodes for a given service. The first step in generating a passcode is selecting the desired service. This process is shown in Figure 5.

Figure 5 – The first step in generating a passcode: selecting a service.

As with the delete service screen, the service selection screen requires the awkward highlight-and-select process to actually select a desired service.

Once a service is selected, the new passcode for that service is presented. Presenting the passcode to the user in the simplest, friendliest manner possible was the primary goal, and several iterations were done on this one screen alone, as shown in Figure 6.



(a) (b) (c) (d)

Figure 6 – Various iterations in the passcode screen, with (a) being the first and (d) the last.

The third iteration introduced the concept of chunking, since small groups of numbers tend to be easier to remember than long, random sequences. A similar chunking was then presented in the actual login form; that is, the login form to authenticate to the server divided the passcode form element into two text fields. The final iteration took the concept of chunking one step further and introduced an SKey-style mapping from passcode numbers into simple words. This technique was again driven by the goal of creating the friendliest user interface possible: simple words are easier to remember than random number sequences. The vocabulary used was a subset of the SKey vocabulary, as described in [13].

One modification that would further improve the current passcode display would be to modify the font of the actual passcode string to make it easier to read. For example, a larger, bolder font would allow the passcode to be clearly read from a distance. Unfortunately, the MIDP framework does not provide such functionality in its default UI components, though one could conceivably implement a custom component for this task. This issue notwithstanding, though, user testing through the various iterations showed that the current passcode screen was quite simple to understand and to use, and was certainly preferred over SecurID-style implementations.

VII. Remaining Issues

Though the client and server both meet all the design goals set forth, a number of issues do remain. These include the software distribution problem, the key distribution problem, and security considerations that apply exclusively to cell phones.

While a thorough description of the client itself has been presented, no consideration has been given to how a user would actually install the client software on his cell phone. During development and testing, software was uploaded to the cell phone through the use of a USB cable. It is doubtful, however, that a prototypical user would have such a cable, let alone know how to use it correctly. Two alternative solutions would be to download the software through an HTTP connection on the cell phone, or to have the cell phone provider itself provide direct access to the software. The first of these has the problem that not all cell phones support HTTP connections, while the second would likely require a partnership with the various major cell phone providers.

The second issue is the perennial key distribution problem. Because the security of the system hinges on the secrecy of the shared secret, the shared secret should be transmitted to the user *out of band*. For example, the simple mechanism of sending the shared secret via email and then uploading it to the cell phone via a USB cable is inadequate, since an adversary can intercept the secret at various points without the user's knowledge. Instead, the shared secret should be transmitted through a medium such as paper mail, and then entered manually into the cell phone. Of course, this need for security clashes with the usability of the system, since it is a cumbersome task to type in the shared secret on a cell phone keypad. An alternative solution might be to download the shared secret directly onto the cell phone via an HTTPS connection or through the cell phone provider. Sniffing a cell phone connection is likely far more challenging than sniffing a wired PC, especially when an HTTPS connection is used.

Finally, although the SecureCell system has been billed as a possible replacement for token-based two-factor authentication schemes, it does have an important difference in terms of security: whereas tokens are *closed systems*, cell phones are *open systems.* In other words, a cell phone allows software to be uploaded to and downloaded from the cell phone, while a token is *sealed*. Thus, the only way for an adversary to impersonate a given user using a token-based system is to actually have possession of the token itself. With a cell phone system, though, an adversary merely needs access to the cell phone for a short period of time to obtain the shared secrets for all the services on the cell phone; a user would be completely oblivious to the fact that he has been compromised. In short, the same openness that makes a cell phone an attractive medium for a mass-consumer authentication system also makes it vulnerable to such an attack. While there are ways to mitigate this problem (i.e., periodically change the shared secret), this is an issue that is inherent to any cell phone system.

VIII. Conclusions

The SecureCell system presented in this project demonstrates how to use a familiar end-user device—the cell phone—to build a fully functional two-factor authentication system. The central goal of any such two-factor scheme is to provide a stronger authentication mechanism than traditional single-factor schemes, addressing specifically the costly and growing threats of keyboard logging and phishing. Although alternative solutions such as biometrics and other two-factor authentication schemes exist, none is currently viable for the consumer market because of high cost and difficulty

to use. The SecureCell system addresses both of these issues by deploying carefully designed software onto existing user cell phones. The system itself consists of two basic components: the cell phone client and the authentication server. The back-end piece is an enterprise-grade server that features pluggable authenticators, configuration management, identity management, and front-ends. The client piece runs locally on a Java-enabled cell phone, and generates passcodes for multiple services. Many iterations of the user interface on the client were done to achieve a friendly and powerful final product. Lastly, though issues do remain, the overall system shows a viable and powerful alternative to more costly and cumbersome systems.

IX. Acknowledgements

Credit must first be given to the project advisor, Professor Rob Miller, who provided invaluable input during the development of the system. His comments and suggestions were particularly helpful in creating a solid user interface for the client piece. Next, the Chisec Group must be acknowledged for their weekly input into various issues during development of the system. Finally, Jorge Marra and Santiago Marra offered help in defining the original scope of the project, as well as aiding with technical issues that arose during development.

X. References

[1] Internal report by Gartner Group provided by employer, 2004.

[2] Rainbow Technologies, Inc., "Two-Factor Authentication—Making Sense of all the Options," ITSecurity.com, February 2002, Rpt. in http://www.itsecurity.com/papers/rainbow2.htm.

[3] Cline, Jay, "How to Build Privacy into Customer Authentication," Computerworld, April 2004, Rpt in http://www.computerworld.com/managementtopics/ebusiness/story/0,10801,92511,00.html.

[4] Ilet, Dan, "Microsoft: Two-factor authentication would thwart phishers." ZDNet UK, November 2004, Rpt in http://news.zdnet.co.uk/software/applications/0,39020384,39174106,00.htm.

[5] Roberts, Paul, "Gartner: Consumers dissatisfied with online security," Computerworld, December 2004, Rpt. in http://www.computerworld.com/securitytopics/security/story/0,10801,98083,00.html?from=story%5Fkc.

[6] Wu, M. and Garfinkel, S. and Miller, R., "Secure Web Authentication with Mobile Phones," 2004, Rpt. in http://sow.csail.mit.edu/2003/proceedings/Wu.pdf.

[7] Tuohey, Jason, "New AOL software gives added security," Computerworld, November 2004, Rpt. in http://www.computerworld.com/securitytopics/security/story/0,10801,97599,00.html.

[8] M'Raihi, D., Bellare, M., et al, "HOTP: An HMAC-based One Time Password Algorithm", Internet Draft, October 2004.

[9] Eastlake, D., "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001.

[10] Krawczyk, H., Bellare, M., and Canetti, R., "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[11] JSR-000037 Mobile Information Device Profile 1.0 (Final Release), Available at http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html.

[12] JSR-000118 Mobile Information Device Profile 2.0 (Final Release), Available at http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html.

[13] Haller, N., Metz, C., et al, "A One-Time Password System", RFC 2289, February 1998.