Single Sign-On: Is It Really Possible?

Jonathan Chinitz

As the proliferation of applications accelerates, users find themselves losing the authentication war; every application that they are introduced to requires its own authentication or login. Some applications do this because they need to identify the user they are dealing with. Other applications do so because they simply do not trust anyone else to conduct authentication for them. The evolution of computing has introduced many ways to accomplish authentication, some stronger (in the security sense) than others. Single Sign-On (SSO) represents an attempt to address the “multiple login” issue, as well as other issues that are intrinsically tied to multiple logins.

WHY IS SINGLE SIGN-ON SO IMPORTANT?
The importance of SSO cannot be stressed enough. Many organizations are implementing SSO (despite the many technical issues involved, which are examined later) because of numerous significant benefits:

- **User productivity.** This is the most immediate benefit of implementing SSO. If I can accomplish in one login what today takes me ten logins to accomplish, one would have to agree that there is inherent efficiency in making this happen. I no longer have to remember ten accounts and ten passwords. I do not have to remember when I will be asked to change my password on system X, what policies govern this password versus the password on system Y, and who do I need to contact when I forget the password on system Z.

- **Increased security.** This is a less tangible but far more important (at least in the author’s eyes) benefit of Single Sign-On. The human brain can remember only so much detail, and eight, nine, or ten different accounts and passwords far exceeds that amount. I think we all know what users do when they are overloaded. They write their user IDs and passwords down and put them in “safe” places: in desk drawers, under keyboards, or in a file on their computer or PDA. The system needs to manage this for you and it needs to do this securely and efficiently.
Some claim that implementing Single Sign-On can actually weaken security. The argument goes something like this: All I have to do is break one password and I have broken them all. This argument, however, is patently false. If the password used to protect my SSO identity is inherently weak (because it is static or because it has a weak policy engine behind it), then it is probable that my other passwords are of equal quality. So breaking any one of them would probably mean that I have now discovered the password for more than half of the other systems at the same time. The time it takes to break a weak password is no time at all. Knowing that the vulnerability of any security system is measured by its weakest link, what is required is to strengthen that link, not throw away the system.

Cost savings. Finally, increased productivity and increased security add up to significant cost savings for the enterprise. The former is easy to measure, as shown in Figure 1. As for increased security, it can be said that this benefit can have an even greater impact on a company’s balance sheet. Just ask yourself who would you rather do business with, a company that knows how to secure its assets, or one that does not?

**DEFINING SINGLE SIGN-ON**

Single Sign-On has been around for quite some time. Early distributed systems, such as Kerberos and Lotus Notes, implemented SSO as a matter or practice; that was the way that designers built applications for those environments. The model was designed around a centralized repository that stored user information (i.e., the user's name and a shared secret — the user's password). Applications used an API that contacted the repository, performed the authentication event, and returned a network credential to the user. The authentication would succeed if the user provided the correct user name and password. Subsequent application invocations used the credential for proof of identity and all was well. This single network credential model is extremely pow-

<table>
<thead>
<tr>
<th>EXHIBIT 1</th>
<th>Example of Cost of Passwords to the Enterprise</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A: Manual Login</strong>&lt;br&gt;Average number of seconds for one login (5) + Average number of seconds spent per user waiting for login and application to launch (10) × Average number of applications (8) × Number of times each application is used per day (3) × Average work days per month (20) &lt;br&gt;Total: 7200 seconds per user per month</td>
<td><strong>B: Password Changes</strong>&lt;br&gt;Average number of seconds spent changing a password (30) × Average number of applications (8) × Number of times password is changed each month (1)</td>
</tr>
<tr>
<td><strong>Total: 240 seconds per user per month</strong></td>
<td><strong>Total: 240 seconds per user per month</strong></td>
</tr>
<tr>
<td>A + B + C = hours per user per month (2.13) × average hourly wage ($15) × 12 months = Annual cost to a company with 10,000 employees: $3.834 million</td>
<td></td>
</tr>
</tbody>
</table>
ful as long as all applications adhere to it. In Kerberos terms, all applications would be considered to be “Kerberized.”

Unfortunately, the situation currently is that there are many disparate computing environments, each with its own security infrastructure. Most of them are password-based, but there are some that use other means of authentication, such as one-time passwords (OTPs) and public key certificates. What an SSO system is forced to do in this scenario is to provide a “meta-authentication facility,” in which the user first authenticates to the SSO system. When the user attempts to log in to a subsequent application, the SSO system retrieves the user’s name and password from the SSO system and performs the login to the target system.

I call this model “authenticate once, log in everywhere,” because what is really happening behind the scenes is multiple logins resulting possibly in multiple network credentials, one for each target. It is important to state that the Single Sign-On system itself is most likely based on the single network credential model. You do not want to have to authenticate to the SSO system every time you want to retrieve a user name and password.

Essentially, Single Sign-On could be fairly benign and actually quite trivial to implement. A robust SSO system, however, must also address the following requirements:

- **Ease of use.** The SSO system needs to be easy to use and as transparent as possible to the end user.
- **Security.** The SSO system should employ strong security if it is to be the keeper of passwords.
- **Multiple authentication methods.** The SSO system should support existing as well as newer forms of authentication, such as tokens, certificates, cookies, and smart cards.

- **User (account) management.** How do I add users to the system? What do I need to do to make this work for all my targets?
- **Password synchronization.** Does the SSO system require that passwords be synchronized? How is this accomplished?
- **Authorization and session management.** Can the SSO system provide access control to itself and to the targets that if supports? Can the SSO system tell me who is logged in and possibly even force them off?
- **Infrastructure.** How can operational stability be ensured, given that the SSO system cannot be a single point of failure? We will also examine sample SSO architectures based on different approaches: scripting; API toolkit; DLL replacement; application replacement; and protocol interception, inspection and manipulation.
- **Bootstrapping.** What are the challenges of deploying the SSO system? How do I get all my users into the SSO system?

The foregoing requirements are no discussed in greater detail.

**Ease of Use**

One might contend that if we asked vendors to actually implement what they are selling (i.e., a Single Sign-On system) things would be vastly simplified in and of themselves. Why should we not expect our users to log in once and only once? Surprisingly, the “single” aspect of Single Sign-On is one of the more difficult requirements to satisfy. It requires that the SSO system be integrated into the operating system login mechanism.

Unfortunately, like many applications, the platform login mechanism is different for different operating systems. Windows NT uses GINA (graphical identification and network authentication), whereas Windows 9X uses a network control panel device. Some UNIX systems use...
PAM (pluggable authentication module); others use a plain login program. Mainframes use RACF, ACF/2, or Top Secret. A good SSO system will have thought about these and addressed them with add-on modules that are OS specific. Beware of such terms as “Reduced Sign-On” or “Single Sign-On for the Web.” You will get exactly that.

Ease of use is also measured in the system’s user interface. Most SSO products offer some form of graphical front end that aggregates the sign-on activity, allowing for point-and-click activation and monitoring of the login sessions. Intuitively, that seems acceptable; there are, however, certain environments in which this change of style might represent a departure from the norm. Some users may have created their own automated environment, whether through startup shortcuts or scripts, which will now no longer work because these are invoked immediately after platform login and not from the SSO tool. Graphical interfaces are desirable, but they should not be mandatory.

**Secure Single Sign-On**

Purchasing a Single Sign-On tool does not necessarily mean that you have bought a security solution. There are certainly products on the market today that sell ease of use under the banner of security. Never confuse the two. A false sense of security is the worst kind of security. SSO products need to utilize encryption, as well as other accepted security practices. Most SSO products will utilize encryption for their own login, or what I call the management interface. This is fairly straightforward for vendors, because they control all aspects of the application (the client GUI and the SSO server).

This is not the case, however, for the application interface, which is where SSO products differ the most. Many products that fall into the ease-of-use category simply pass the user name and password into the application channel, and if the application itself does not use encryption, then the login continues to be insecure. Some Single Sign-On products address this by replacing components of the application, whether they are DLLs or entire executable images. This raises another set of issues regarding the feasibility of the implementation itself, which is discussed later.

When evaluating Single Sign-On solutions, I strongly recommend that you automatically place another “S” in front of SSO. That “S” stands for “Secure,” and what you get is SSSO: Secure Single Sign-On. Automating sign-on is simply not sufficient in today’s world; SSO products need to address the encryption and other security aspects of the sign-on as well. Otherwise, you may simply be creating a new set of problems to contend with.

**Multiple Authentication Methods**

The authentication technologies that are available to us today are far more advanced than the password-based systems that represent the bulk of our legacy infrastructure. The challenge that IT is faced with today is how to integrate stronger authentication methods into applications without creating another security infrastructure to maintain. One of the ways to introduce these newer authentication techniques into an organization is through Single Sign-On. If the SSO system were to provide support for the new infrastructure while at the same time making it easy to integrate the legacy password system, the resulting benefit is clear. We would be strengthening the overall security of the environment while making it easier to use.

The technical challenge for Single Sign-On and authentication boils down to an exercise in “mapping.” This is especially true when dealing with certificate-based or token-based systems. The SSO system must be capable of performing certificate login, requiring it to interface with a certificate directory (most likely through the use of LDAP), in order to retrieve both the user’s certificate and the certificate revocation list (CRL) associated with it.
Associated with it. Once the authentication phase is complete, the SSO system will need to map the user (a simple way would be to use the CN attribute from the certificate) to an internal profile, possibly containing names and passwords of other target systems. The same is true of token-based systems. In general, SSO systems need to be good at interfacing with other authentication systems. In a sense, they need to act as “authentication gateways” (see Exhibit 2).

User (Account) Management
Every SSO system supports a management interface to its server. Administrators use this interface to add and delete user accounts, set or reset passwords, and enforce password and account policies that apply to its users. That is fine for the SSO system itself, but what about the target systems that it is signing on to? How does the SSO system update these targets? Should that be its responsibility? Is this to be done in a batch mode or does it have to be updated in real time? Different products take different approaches to solving these issues.

Some Single Sign-On solutions do not attempt to deal with this issue and instead defer to the larger systems management products (e.g., CA’s TNG Unicenter, IBM’s Tivoli, or BMC’s Control-SA) to handle account and password synchronization. The sole purpose for these SSO products is to automate the sign-on to the target systems. In these cases, the products will have interfaces that allow them to be updated by the larger system when an account is created or a password is changed. The SSO product is treated as another target in the management system.

Still other SSO solutions focus on password synchronization with minimal account management. The premise behind this approach is that Single Sign-On can be accomplished only with a synchronized password on all targets. This way, the SSO product can be deployed entirely as a server-side solution with minimal interference on the client side. Since all the passwords are the same,
there is nothing to store and nothing to remember other than the single user name and password (see Exhibit 3).

A third alternative for user and account management with SSO is a system that can operate both with synchronized as well as nonsynchronized passwords (see Exhibit 4). Needless to say, the nonsynchronized environment is far more difficult to manage. The Single Sign-On system needs to maintain an SSO map (I like to call them “e-sticky notes”) where the user’s profile defines the name and password for a specific application on a specific host. There is also an implied default hierarchy in case there is no specific entry for application or host.

Another advantage that nonsynchronized SSO systems have is that they can employ random passwords for any of the targets that they support. A random password is inherently stronger than a typed one. What is more important than its strength, however, is that a random password is never known by the end user. Therefore sign-on to the target system must be done through the SSO system.
guard, in the event that the SSO system is not operating properly, the administrator can always assign a temporary static password to allow the user to log in.

In all of the foregoing cases there is one important aspect that I have not mentioned. What happens when an administrator on target A changes your password or disables your account? How does the SSO system get notified? The answer is different for each of the three implementations already described. In the case of a systems management tool this should never happen, because the premise is that the management tool is the center of the universe and the targets are not to be administered independently. In the case of synchronized passwords, changing one necessitates a change in all targets, which requires synchronizing agents on all targets. Depending on the number of target systems, this could be an expensive operation, especially if performed in real time. Finally, in the non-synchronized model, only the SSO server need be updated with the change of the target — nothing else.

**Authorization and Session Management**

I would classify both of these requirements as advanced features of SSO. They are not required per se to implement a successful SSO environment, but they are extremely useful when they are available.

First, authorization cannot be accomplished without authentication (whereas the inverse is not true). Once users are authenticated to the SSO system, it would be helpful to be able to control their access to applications from one location. By placing access control features into the SSO server and managing them in a central place, a good administrator can control the use of applications throughout an enterprise. Common authorization requirements would be to enable or disable access by user, group, role, location, and time of day.

Authorization is not necessarily limited to the login itself. If the SSO system is well integrated into the target application environment, either through server-side agents or through higher-level management products, then it is possible to control access to application resources from the SSO system itself.

Occasionally, there may be overlap between the authorization implemented in the SSO server and the target application, while other times they actually might conflict. It is possible that the Single Sign-On system will allow a user to log in to a database, but the DBA will have disabled the account temporarily or even permanently. This is one of the reasons why some SSO products shy away from authorization and focus simply on automating the login.

The SSO system can also keep statistics on number of users logged in as well as the times that they have logged on and off. This information is classified under the broad banner of session management. A good example of this is the RADIUS protocol. What is of more interest is the ability to “bounce” a user off the network. In a centralized environment — one in which all the terminals are connected to a single source — this is not difficult to do. In a distributed environment, however, this becomes much more interesting. How does one get logged off automatically if there is no current network activity? A user could have achieved sign-on through the system, but is currently not signed on to any particular application. Supporting this type of session management requires a more intricate design, one in which the SSO system actually contacts the client desktop to “destroy” the SSO credentials.
**Infrastructure**

It should be obvious by now that an SSO system is itself a large, multiplatform, distributed application. Therefore it requires all the common design principles that any system of this type should exhibit. The SSO server needs to be replicated, both for failover as well as load-balancing reasons. The SSO server can never be a single point of failure. In turn the SSO clients must be capable of locating the server with minimal effort. Users tend to get impatient when it takes too long to log in. It should be understood that to be effective for today’s organization, SSO needs to be implemented across all architectures, from mainframe and client/server to the Internet. It is in meeting this need in the marketplace that many Web security and SSO products come up short.

The SSO server must also serve as a security/policy engine. A good SSO system will know how to enforce strong passwords by performing things like dictionary checking, password construction policies, account and password expiration policies, as well as support a hierarchy of these to make administration easier.

Finally, SSO systems today are being asked to scale to tens and hundreds of thousands of users. The storage aspects of such systems demand that SSO products make use of advanced database technology or an enterprise directory, preferably one with an LDAP interface to it.

Now that we have explored a variety of requirements for SSO systems, it is time to consider some of the more popular Single Sign-On architectures. I would classify them in the following categories:

- [ ] Scripting
- [ ] API toolkit
- [ ] DLL replacement
- [ ] Application replacement
- [ ] Protocol interception, inspection, and manipulation

A closer examination of each of these architectures will show their strengths and weaknesses.

**Scripting.** This is, by far, the simplest approach of any. It is fairly easy to implement, because it is noninvasive to either client or server applications. Its primary goal is to automate the login procedure. At the same time, it is not concerned with securing any of the existing applications. Also, scripts are not necessarily intuitive to write and certainly present a scalability concern if they need to be managed on each and every workstation (see Exhibit 5).

**API Toolkit.** This is probably the most common method for enabling security, not just SSO, in applications today. It is also by far the most invasive. Organizations must gather all the applications that they want to secure and reimplement them using a brand new set of APIs. This approach certainly has the advantage that security is embedded inside the application, attaining a high degree of optimization and efficiency. This method, however, does not work at all for commercial off-the-shelf (COTS) products, in which the customer does not own the source code to the application (see Exhibit 6).

**DLL Replacement.** Replacing a DLL to achieve Single Sign-On is similar to the API approach, except here the SSO system is making an assumption that most of the applications use one or more of the same system DLLs, and therefore they replace it with one that implements the authentication and credentials management. It most often also augments the network connection with encryption facilities. The issue with DLL replacement is that it is impossible to cover all the applications by replacing DLLs. A good example is the WINSOCK DLL. Some implementations replace this DLL because it is a popular one that is used for socket-based applications. However, Microsoft Office applications do not use sockets; rather, they use a different network paradigm entirely known as SMB. Replacing system DLLs can also be prob-
1. SSO client waits for event to happen (application to run, specific window to open—e.g., mail)

2. When application X opens, SSO client opens encrypted session with SSO server to get User ID and password for the application

3. SSO client gets (User ID for app1, and password for app1) from stored "profile" for USER

4. SSO client inserts User ID and password for the application into the necessary location on the screen (either through mouse location choices, or something like "select window, tab, tab, tab, userid-app1, tab, password-app1, tab, tab, return")

5. User ID and password then sent through normal means to the application (target); encryption is application-specific

Legend

- Secure
- Cleartext
SSO Architecture Utilizing an API Toolkit Approach

1. Application is started (e.g., mail) and works normally—when login option is used new vendor-created APIs are called instead of the client's original login APIs (this requires a developer to recode and compile the source).

2. Replaced API presents USER identity to target server/service through normal system DLLs (e.g., winsock).

3. Server gets login request and USER identity; the new API then calls (again recompiled for the server application) and checks the validity of the USER identity with the SSO server.

4. SSO Server responds with yea or nay.

5. Application Server accepts the authentication and continues to communicate with the client.

Legend
- Original
- SSO System
lematic when a new platform distribution is loaded on the workstation, as it will often replace the Single Sign-On DLL.

Application Replacement. Replacing commonly used applications with ones that are SSO-enabled is another way of implementing SSO, one that yields the tightest integration between the application and the security infrastructure. On the other hand, it is rather impractical to ask a vendor to build hundreds of SSO-enabled applications — especially when that vendor does not own them in the first place. Furthermore, this puts a significant burden on customers to deploy this type of solution. They quickly realize that instead of being in the business they were in yesterday, they have entered the software distribution business in a big way.

Protocol Interception, Inspection and Manipulation. This approach to achieving SSO is a twist on the DLL and API toolkit approach (see Exhibit 7). Instead of replacing system DLLs or introducing modified applications, the protocol interception, inspection, and manipulation method tackles Single Sign-On — as well as the security issues surrounding it — at two different layers: the network layer and the application layer. Network interception is often accomplished through the dynamic loading of a device driver. The driver is responsible for low-level connection management and encryption. The application layer is responsible for adding security content to the application dynamically, at runtime, so there is no modification to existing applications.

Because it is transparent to existing applications, protocol interception, inspection and manipulation is not dependent on the existing application architecture: it relies only on the network connection. It is also relatively quick and easy to deploy, since organizations do not need to replace applications, modify DLLs, or utilize API toolkits, regardless of the application. And by adding authorization parameters to the system, this method enables security enforcement. One drawback to this approach, however, is that the Single Sign-On system must keep up with changes as new operating systems are introduced. Exhibit 8 lists advantages and disadvantages of some SSO architectures.

BOOTSTRAPPING A SINGLE SIGN-ON SYSTEM

Suppose that you have decided to deploy an SSO system; now, there are many more things that you need to be concerned with. I do not mean to downplay the importance of analyzing the features and benefits of the various products, procuring the hardware necessary for deployment, lining up management support, acquiring the proper personnel to implement the system, and implementing overall management of a project of this scope. All these are critical. Rather, what I would like to focus on are two issues that are equally critical to the success or failure of an SSO implementation, but often overlooked in the process. The first is software distribution, and the second is database population.

Software Distribution

It should be apparent that software distribution is really not an issue for Web applications. Every desktop comes equipped with a browser. The Web protocols of today know how to do basic authentication and certificate-based authentication. Other Web-based applications utilize their own private login pages, which SSO

It is inconceivable to ask customers to completely tear down the existing security infrastructures, rename accounts, and reset passwords and policies in the name of a new SSO system.
EXHIBIT 7  An SSO Architecture Drawing on Protocol Interception, Inspection, and Manipulation

2. Interception Driver in network stack looks at network traffic that is generated by application (i.e., looks for specific TCP port usage). When traffic is generated on a specified port/host, the Driver redirects all traffic through its tunneling mechanism, possibly encrypting it as well.

3. SSO slave server (which is colocated with the target application server) receives the traffic, possibly decrypting it and redirects the data stream into the Protocol Support Module (PSM). The PSM determines what information it needs to provide or augment the data stream with, e.g., USER identity, plus info on where to put app-userid and app-password.

7. Application receives userid and password, and continues processing, unaware of the manipulation.

1. Application is started.

6. SSO slave server inserts app-userid and app-password into the original data stream removing any data that might have been there--this is then passed to application.

5. SSO server supplies information if the identity is valid.

4. SSO slave server contacts SSO server to validate the User ID—if valid, then requests the app-userid and app-password

Legend

Original
SSO System

SSO Server

Passwords
products need to know how to deal with. There is nothing special that needs to be installed on the desktop to secure a browser. No wonder most of the Web security vendors advertise support for Single Sign-On.

The same is not true of client/server and legacy applications. Here, we ultimately need to put something on the desktop. The issue now becomes how easy is it to deploy the client-side component of SSO? As a customer you cannot afford to make a mistake here. (Take a moment to calculate what it would cost to rectify the mistake across 10,000 desktops.) Other things to consider include how big is the desktop component; how easy is it to install and uninstall; can this be done by an end user (which would simplify the deployment tremendously)? Once installed, is the desktop component easy to configure? How much information about the SSO system is stored locally on the desktop rather than on the SSO server? The answers to these questions are vital in choosing an appropriate SSO system that you can use.

### Database Population

The second issue that many organizations often overlook (until it is too late) when implementing Single Sign-On is populating the SSO database. Imagine having the following conversation with your SSO product vendor:

**Vendor:** Ok, we’re about to bring up the new SSO system.

**Customer:** Great, we can’t wait.
Vendor:  Oh, by the way, we are going to need all your accounts and passwords to initialize the new system.
Customer:  Great, they’re all on the mainframe in XXX.
Vendor:  Oh, we can’t use those (sigh).
Customer:  Why not?
Vendor:  Well, what is actually stored for the password is a hash, so it is useless to us. Furthermore, the XXX IDs are not accepted as proper account names in the new system.
Customer:  So what do you propose?
Vendor:  How about we reset everything and start from scratch?
Scene:  Mental image of vendor getting knocked on the head with a frying pan…

The foregoing scenario is symptomatic of what I call the “Genesis Event.” It is inconceivable to ask customers to completely tear down the existing security infrastructures, rename accounts, and reset passwords and policies in the name of a new SSO system. No company can afford to be “born again” just to implement SSO. That is why mature SSO systems must be capable of a smooth, almost seamless integration of existing security environments into their own.

There are two techniques for solving the Genesis Event and implementing SSO. The first is user self-registration. Once the Single Sign-On system is in place, users are put through a quick and easy registration process. The classic problem of assigning a user a new account on a system is that of distributing the password to them. What I propose here is that for the first registration event, the user utilizes an account and password that can be validated against an existing security database. This serves a number of purposes. First, it validates the user before loading his or her information into the SSO system. Second, it allows the SSO system to “activate” the user, possibly asking the user for additional pieces of information. Finally, it allows the SSO system to synch itself with the other target systems. Part of this final phase of registration could be to create accounts on other target systems or to contact the company’s PKI and register for a certificate.

Another technique for quick bootstrapping of SSO is automatic password learning. Since most password databases are not useable in an import operation, there is no way for an administrator to insert your password into the SSO system for that particular target. Instead, the Single Sign-On system should simply monitor your login to that application the first time, record your password and store it in the SSO system. The next time you attempt to log in, the SSO system will do the rest.

CONCLUSION
I think we can all agree on one thing: SSO is not easy. It has grown from a simple automation tool to one that encompasses a variety of management and security disciplines. My suggestion to the reader is to take it slowly. Do your research and find the Single Sign-On product that best matches your requirements and your user population. There will always be tradeoffs between ease of use, complexity of administration, and security. No one product does it all. If you don’t have the expertise in-house to implement SSO, seek the assistance from your vendor or a third party. At the end of the day you will be glad you did.