GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials: Network, Endpoint, and Cloud (Security 401)"
at http://www.giac.org/registration/gsec

Simple
Single Sign-On Solution for Web Applications

Peter Kohler
2004-02-18
GSEC Practical Version 1.4b, Option 1

© SANS Institute 2004, Author retains full rights.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Table Of Contents

© SANS Institute 2004, Author retains full rights.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

1. Abstract

Due to the variety of existing computer systems and services, Single Sign-On Systems tend to be highly complex and fragile. The more the scope of a Single Sign-On System is limited however, the easier and simpler it can be implemented.
This paper explores the "low-end" of Single Sign-On Systems by proposing a simple but secure way to implement a Single Sign-On System for Web Applications. Furthermore, it examines and compares available Open Source cryptography projects for their suitability for a possible implementation.

2. Requirements

The need for a Single Sign-On System arose from a real world situation: given an existing web-based banking application, a second web-application from an affiliated bank in a different country had to be integrated. The primary goal was to provide customer convenience, i.e. the customer should only need to authenticate once and then be authorized to use the local *and* the foreign application without having to repeat the login process. Further requirements were:

- Simplicity: for cost and security reasons, no additional external systems (proxies, ticket servers) should be required.
- Security: as financial and personal data is involved, the system has to be as secure as possible.
- No legacy systems: Both sides of the application were either developed in-house or available as source code. Because of this, the Single Sign-On functionality can be implemented directly into the applications themselves.

3. Single Sign-On Methods

During the history of Single Sign-On, three different concepts evolved (Lee): Ticketing, Reverse-Proxy and Password-Synchronization. In the following sections, these methods are briefly introduced and evaluated against our requirements.

3.1. Ticketing

In a ticket based Single Sign-On system, a single, central authentication server performs the authentication. Upon successful authentication, this server returns a "ticket" which then can be used to get access to all "ticket enabled" applications and resources. The tickets usually contain an encrypted copy of a temporary secret shared between the client and the application/resource. The most well known ticked based Single Sign-On system is Kerberos , from which also the term "ticket" originated (Smith, 343). Kerberos uses symmetric cryptography; each participant of the ticketing system has its own secret key. The central authentication server has a copy of all involved keys, which it needs to issue the tickets.

But how do ticketing methods fit our requirements? Actually, Kerberos *can* be used for web applications, as demonstrated in (Garman). If Kerberos is used however, the system gets considerably more complex. Not only have the applications and/or web servers to be

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

'kerberized', there also has to be a Kerberos key distribution server installed and maintained. Besides the cost factor of installing and operating an additional service, this also adds an additional attack vector to the system, reducing the overall security.

The ticketing idea per se is simple; if it could be implemented without the required infrastructure of Kerberos, it would be a valid option for our solution.

3.2. Reverse Proxy

Reverse proxies are mainly limited to web based applications. All client (web browser) requests have to pass through a single proxy server. This proxy server is responsible for authenticating the user. Once authenticated, the proxy server maintains a session with the browser and forwards all requests of the user to the required application, inserting the necessary credentials into the requests. Those credentials may differ from the ones used to authenticate on the proxy, and may also differ between the different applications (Posey).

The main advantage of the proxy method is that existing applications can be integrated without any changes. This is often the only solution, if legacy or proprietary applications are involved where source code is not available. Most commercially available Single Sign-On systems targeting web applications are of the proxy type (See (esecurityplanet) for a list of commercial Single Sign-On systems).

As simple as the concept of the reverse proxy may sound, in practice there may side effects that make the actual implementation difficult. As the web browser is not directly communicating with the application, it might be necessary for the proxy to change URLs in the HTML code on the fly. If those URLs are generated dynamically (e.g. by JavaScript), this can become a difficult if not impossible task. Furthermore, most web applications maintain a session with the web browser through the use of HTTP cookies. The proxy server has to ensure translation of these cookies between the application and proxy domain correctly, without collision with the proxy's own session handling.

Regarding our requirements: The main advantage of the proxy method - being able to use existing web application without change- does not apply in our case, as the source code of the applications is available. The disadvantages, as described above, would still be present however, increasing the effort needed to implement such a system. Adding a proxy server also adds costs by having to install and operate (and keep secure) a separate system. Furthermore, due to the additional system, additional attack vectors are introduced, weakening the security of the overall system. Due to these issues, the proxy method is only a second choice for our project.

3.3. Password Synchronization

The third way of implementing Single Sign-On is through password synchronization. This means that a user's password is synchronized among all applications and resources that need authentication. Once authenticated to a system, an application can reuse the given password to access further systems and resources. Examples of this scheme include file system services, such as samba or AFS. If a user logs on to his workstation, the login process passes the user's password to the samba or AFS server to get authenticated access to the user's file system. Single-Sign on in this case works only if the password on all systems is the same. This is however also a weakness of this method, as having the same password for multiple systems increases the potential damage if the password gets into the wrong hands.

As for web applications, password synchronization alone provides only half of a Single-

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Sign on solution. It doesn't solve the problem of how user credentials will be passed from one application (where the user authenticated himself) to another (where the user expects not having to log in again). Somehow, user credentials will have to be passed through the user's web browser from one application to another. This makes the method quite similar to a ticketing system, where the user credentials are used as the ticket.

3.4. Conclusion

What will be our Single Sign-On method of choice when it comes to web applications? Technically, all three methods could be implemented. Proxying is often the only way, if the applications cannot be modified and made Single Sign-On aware. It leads however to high complexity and effort in the configuration of the proxy. Since in our case, modification of the applications is not a problem, the other methods - ticketing and password synchronization - seem more promising and easier to implement. Thus, the proxy method will no longer be evaluated.

As for ticketing, using kerberos is not an option as the costs and complexity of adding a kerberos infrastructure are too high. A simpler method would have to be found. An open question remains for ticketing and password synchronization: How are user credentials (ticket or username/password) passed from one application to another. The next chapter elaborates this question in more detail and tries to find a simple yet secure solution.

4. Passing Sessions between Web Applications

4.1. Common Authentication Schemes for Web Applications

Web applications differ considerably from normal applications running on a workstation, as they are request-response based. Most of the time, the browser is not connected to the web server. Only if the user performs certain actions, such as submitting a form or clicking on a hyperlink, is a connection to the server is established. The server processes the request and returns the result (usually as a HTML page) back to the browser. After that, it disconnects again.

But how does user authentication work when the connection between browser and server is terminated after each step? Of course, it would be very inconvenient for the user to have to log in again after each click in the application. There are several ways this problem can be solved:

- Client side caching of the credentials: Upon first login, the web browser caches the credentials locally. With each further request, it adds them to the HTTP headers automatically. For security reasons, the browser sends the credentials only when accessing URLs on the same server and within the same URL path. This method is also known as "Basic Authentication".
- Using dynamically generated URLs to cache the credentials: Upon successful login, the server inserts the credentials into all URLs that are pointing back to the application. The application is then able to extract the credentials from the next request (i.e. by parsing the query string).
- Using HTTP cookies: After the user has authenticated himself to the server successfully, the server returns an HTTP cookie containing the credentials. This cookie is stored within the web browser and is sent back to the server with each further request.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

- Using a random session id: Once the user is authenticated, a random session id is generated and stored on the server. The server inserts this session id either into all URLs that it generates, or into a HTTP cookie. When the browser issues the next request, the server verifies that a corresponding session id exists in its cache. If it does, the request is considered authenticated.

It has to be noted that embedding user credentials within URLs and/or cookies is a potential security risk, as they may end up on the user's hard disk in the browser cache or in the cookie database. A potential intruder (human or malware) could easily extract them once it gets access to the user's hard disk. The "Basic Authentication" method is more secure, as the credentials are only stored within the memory of the browser (and forgotten, once the browser is terminated). The best option, security wise, is the random session id method, as it allows also implementing a time limit. Even if URLs could be extracted from the browser cache, the session id would probably be invalid by then. Actually, it is very similar to a ticketing system, only that the ticket issuer is also the only one that accepts this ticket back for authentication.

4.2. Extending the Scope: Transferring a Session

Now, lets analyze how an authenticated session with a web application can be transferred to another (remote) web application, without having the user log in again.

The first question that arises is which of above authentication methods would be the best starting point. As already stated, embedding user credentials within URLs or cookies is generally a bad idea. Basic Authentication is more secure; however, the same credentials can only be used for the same host name and within the same URL path (and sub-paths). If the two applications for which Single Sign-On has to be enabled are located on different hosts (or even domains), Basic Authentication will not work. There is no way to tell a browser to use cached credentials for different hosts (and this is good, as it would open huge security holes).

This leaves the session id method remaining; this is good, as it is already considered to be quite secure.

Implementing Single Sign-on means, an established and authenticated session has to be transferred from one web application (lets say 'A') to another one ('B'), without having the user authenticate again. This means, if the user changes from application 'A' to 'B', application 'B' has to automatically create a new session in which the user is also authenticated. For simplicity it is assumed that application 'B' uses the same session id that has been created by application 'A'.

This session transfer introduces two new questions:

1. How is the session id transferred from web application 'A' to web application 'B'?
2. How does web application 'B' verify if the session id that it receives really belongs to an authenticated session of application 'A'?

As for question 1, there are not many options; the session id of 'A' has to be sent to application 'B' by the browser. There's no other way application 'B' could assign an incoming request to a certain session id.

The HTTP protocol offers several possibilities to transfer such information. It could either be encoded in the URL (e.g. in the query string), the HTTP headers (e.g. in a cookie) or in the HTTP body (e.g. by posting an HTML form). The use of cookies is usually limited to the same domain; if e.g. one application is located at foo.com and the other one at bar.com, cookies cannot be used to transfer data from foo to bar. This leaves embedding the session id within URLs or HTML forms as the most generic solution.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Question number 2 is far more interesting; how does application 'B' make sure that the session id it receives from a user's browser belongs to an authenticated session on 'A'? Two fundamentally different methods to achieve this will be discussed in the following sections.

4.3. Server to Server Communication

The most straightforward method to let application 'B' know about existing sessions on 'A' is by a direct communication between the two application servers. The transfer of a session between 'A' and 'B' would be done as follows:

1. The user authenticates to application 'A'. Application 'A' creates a new session with a corresponding session id.
2. Application 'A' transmits this session id together with the user id to application 'B'. On 'B', a new session using the same id is created (Application 'A' logs in to application 'B' in behalf of the user).
3. Application 'A' can now embed hyperlinks to Application 'B', containing the common session id.
4. If the user clicks on such a URL, he finds himself already logged in on application 'B', as the corresponding session has already created.

Although the methods seems to be quite simple and straight forward, there are a few security considerations that have to be taken into account:

- The communication between the applications 'A' and 'B' must be secure. Otherwise, anyone could just create an arbitrary session on ‘B’, without having to authenticate. Possible measures to secure this connection could be
    - Checking of the source IP address (using TCP wrappers or the 'Allow from' directive of the apache http server). This is not very secure though, as the source address could be spoofed (Cole, 605)
    - Establishing an encrypted and authenticated SSL connection between 'A' and 'B', using SSL/X.509 client- and server certificates on the respective side.
    - Using cryptography to encrypt or sign the ids (symmetric or asymmetric encryption)
    - “Out-of-band” communication, e.g. using a dedicated leased line
- Session timeouts: Usually, a session is closed automatically after a certain time of user inactivity. On a single web application, this timeout can be chosen to be relatively short (usually a few minutes), as the user is constantly updating the session with every request he makes to the application. If there's a second application (application 'B' in our case), it could very well be that the user stays there for a longer time. During this time, the session on application 'A' is not being updated and might time out. When the user returns to 'A', his session there has expired and he has to log in again. This problem can either be avoided by increasing the timeout value (and thus decreasing security), or by some form of communication between 'A' and 'B' updating each other’s sessions with each user activity (difficult to implement).
- Server-to-Server communication: Depending on the environment and existing security policies, it may not be allowed for an application server to open connections outside its local network (DMZ).

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

- Reliability: The method requires a reliable channel between application server 'A' and 'B'. If this communication takes place over the Internet, there might be interruptions in the network connectivity. During this interruption, Single Sign-On would not work.

4.4. Signed or Encrypted Ticket

In the previous described method, the direct communication between the two application server is used mainly for two purposes: providing a means for 'B' to identify and trust the session ids received from a user's browser, and to synchronize the creation of the session between application 'A' and 'B'.

The first part of this functionality -trusting a received session id -could also be solved by using cryptographic methods: encryption and/or signing (RSA).

4.4.1. Encryption: Using symmetric cryptography

Lets assume 'A' and 'B' have a common secret key available that has been securely distributed and installed beforehand. Application 'A' would encrypt the current session id of a user with a symmetric crypto algorithm and place the encrypted result into the URLs pointing to 'B'. If 'B' receives such a URL, it can extract this encrypted id and decrypt it using its identical secret key. Assuming that the key is only available to 'A' and 'B', application 'B' can be sure that the session id, if it could be decrypted successfully, must have originated from 'A'.

4.4.2. Signing: Using asymmetric cryptography

The same can be achieved using asymmetric or public key cryptography: A key-pair is generated beforehand, consisting of a public and a private key: K_{pu} and K_{pr}. The private key is then installed on 'A', the public key on 'B'. Again, 'A' encrypts its session id using its private key $K_{pr.}$ This process, encrypting a message with one's own private key, is called signing. As the corresponding public key K_{pu} is public, everyone is able to decrypt such a message. So this does not provide any confidentiality. However, being able to decrypt it with a certain public key means that the message could only have been created by someone who owns the corresponding private key. This is exactly what is needed to solve the problem of authenticity.

As in the previous example using symmetric cryptography, 'B' decrypts a received session id using the locally available public key K_{pu}. If the decryption succeeds, 'B' is sure that the id originated from 'A', because only 'A' has access to the required private key K_{pr} that is necessary to create the encrypted id in the first place.

4.4.3. Comparison

Although both methods, using symmetric or asymmetric cryptography, seem quite similar, there are a few subtle differences:

- Confidentiality of the keys: In the symmetric key system, there is a single key that has to be protected by all means. Once the key is compromised, the system is broken. In the asymmetric key system however, only the private key has to be secured. If the public key is compromised, it could only be used to verify signatures, but not to create them. The system would still be secure.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

- Performance: Asymmetric algorithms usually need much more computing power than symmetric algorithms. As for our application, this is not relevant however, as the messages that have to be signed (session ids) are very short.
- Key distribution: At first sight, it seems that the distribution of the keys to application 'A' and 'B' is easier for the asymmetric system, as the public key does not have to be confidential and could be transmitted from over an insecure channel. This is unfortunately not true. While it's correct that the public key is not confidential, the operator of 'B' must be sure about its authenticity and integrity.

4.4.4. Session Timeout

In the 'server-to-server' communication method described in chapter 4.3, the creation of the session between 'A' and 'B' was synchronized. When there's no direct communication between 'A' and 'B' however, the session cannot be created synchronously. In fact, the session on 'B' is only created upon the first request of the user to 'B'. This behavior can be a problem, especially if the user does not access 'B' at all. Since there is no time information within the session id, 'B' does not know at which time the URL containing the session id has been created. It could have been created weeks ago, but it would still be accepted by 'B' as a correctly signed id. This is a security risk, as such URLs could be retrieved from the user's browser cache.

This problem can be avoided by adding an expiration date or a timestamp to the session id, and sign or encrypt both. This guarantees that such a ticket, consisting of a session id and a timestamp, is only valid for a limited time. If extracted from the browser cache after the expiration time, it will be worthless.

As in the previous 'server-to-server' method, the session timeout must be chosen to be long enough to allow the user to switch between the applications without either side exceeding the timeout. If this is not acceptable, a possible solution could be to embed URLs to the currently not active application into the HTML (e.g. by referencing images) in order to keep the session there alive.

5. Implementation

Implementing secure cryptographic algorithms is a difficult task. Fortunately, there are several open source implementations available that have a good reputation and that are supported by large communities. Due to the openness of the implementation the risk of flaws in their implementation is small. Should there be any found nevertheless, they are usually eliminated quickly.

In the following implementation examples, the focus lies on the "signed ticket" method, using tickets containing a random session id and a time stamp. Two well-known open source projects are evaluated: GNU Privacy Guard (GnuPG) and OpenSSL. Both provide the necessary symmetric and asymmetric crypto algorithms that are needed to create and verify the authentication tickets.

5.1. GNU Privacy Guard

GnuPG (GnuPG) is a tool for secure communication and data storage. It is able to encrypt and sign data, based on license free symmetric and asymmetric crypto algorithms. Its main usage is usually encrypting and signing email messages, but it can actually be used for any arbitrary cryptographic needs. All functionality is bundled within a single command line binary (gpg), making the handling and integration into a web application simple. Lets go through the necessary steps to generate the keys, signing and verify a ticket:

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

5.1.1. Creating the keys

In GnuPG, keys are stored in so called 'key rings'; they are simple files, containing one or several keys. Public and private keys are stored in different files. To create a key pair, the "`—genkey`" option is used. By default, the key rings are stored in the user's home directory in `~/.gnupg/pubring.gpg` and `~/.gnupg/secring.gpg`. In order to create the key ring files separately, the options "`--no-default-keyring`", "`—keyring`" and "`--secret-keyring`" have to be used.

This is the complete command used to create a key pair and store it in the files "publickey.pgp" and "secretkey.pgp":

```
gpg --no-default-keyring --keyring ./publickey.pgp \
    --secret-keyring ./secretkey.pgp --gen-key
```

During the key generation process, several parameters are needed, such as the crypto algorithm, key length, expiration time and a user id. Following parameters have been selected for this example:

- Crypto algorithm: DSA and ElGamal
- Key length: 1024 bit
- Expiration: never
- User id: Application_A <a@foo.bar>

Once the key pair has been generated, it can be installed on application server 'A', the public key on application server 'B'.

5.1.2. Signing a ticket

For the following example, it is assumed that the session id is a hexadecimal 128bit random number (e.g. as used by the Apache Tomcat application server). To make the ticket complete, a time stamp in ISO format of the form CCyymmddThhmmss is added. As separator, a colon is used.

Example ticket: `36BC0E6E7FCDBC85BB65F08C28C73A46:20040217T160531`

The following gpg command is used to sign the ticket:

```
gpg --no-default-keyring --keyring ./publickey.pgp \
    --secret-keyring ./secretkey.pgp --default-key a@bar.foo  \
    --sign
```

New options are '`—default-key`' to select the private key to be used for signing and the actual '`—sign`' command.

The ticket is sent to the command on standard input, the signet ticket is returned on standard output. As the output of gpg is binary, it has to be converted to ASCII in order to be able to use it in a URL. To achieve this, a base64 conversion is added, either provided by a library in the web application itself, or by an external converter (e.g. mimencode or openssl)[1].

[1] Actually, gpg is already able to generate ASCII output by using the –armor option. However, this ASCII output contains additional header and footer lines that are not suitable to be put into an URL.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

The resulting signed ticket that can be embedded into URLs pointing to application 'B' looks as follows:

```
owGbwMvMwCRYuFTxzbyIy8qMp82SGByMLASNzZycDVzNXM3dnF2cnC1MnZzMTN0MLJyNLJzNjR1NzKyM
DAxMDIwMzUMMzQxMjQ0 77JlZwRphJgkyCXxgmB+mfcFU5MoZV/+2g6tPzdv2wGXJRV+GeWppDz+y68zh
/zHxc4nQkyMcJtdfSQIA
```

5.1.3. Verifying a ticket

Once the ticket arrives at application 'B', it has to be verified that it has been correctly signed by 'A'. The following command is used to extract the clear text ticket from the signed ticket and to verify the signature using the public key:

```
gpg --no-default-keyring --keyring ./publickey.pgp \
    --default-key a@bar.foo --always-trust –decrypt
```

Again, the signed ticket is provided on the standard input, the clear text ticket is sent to the standard output. The gpg output containing the verification result is delivered on the standard error channel. Additionally, the return code can be used to check the successful verification of the signature (0 means the signature is ok). Sample output:

Standard output:

```
36BC0E6E7FCDBC85BB65F08C28C73A46:20040217T160531
```

Standard error:

```
gpg: Signature made Tue Feb 17 17:02:53 2004 MET using DSA key ID 9E58D323
gpg: Good signature from "Application_A <a@bar.foo> "
```

Actually, it would not necessary to add a timestamp to the ticket, as GnuPG adds a timestamp automatically upon signing. It is however more difficult to parse, as gpg uses a primarily human readable date format. So, adding a machine readable date to the ticket probably simplifies the implementation, on the expense of a slightly longer ticket.

5.2. OpenSSL

The OpenSSL (OpenSSL) Project is an Open Source implementation of the Secure Socket Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocol. Although its primary target is to provide encrypted and authenticated network connections, it contains also a general-purpose cryptography library, which can be used for arbitrary applications.

5.2.1. Creating the keys

As with gpg, a private/public key pair has to be generated first. In OpenSSL, the 'genrsa' utility can be used for this purpose:

```
openssl genrsa 1024 > privatekey.pem
```

This generates a 1024 bit RSA keypair. The public key has to be extracted from the private

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

key using the ‘rsa’ utility:

```
openssl rsa –in privatekey.pem –pubout –out publickey.pem
```

As in the gpg example, the private key is installed on ‘A’, the public key on ‘B’.

5.2.2. Signing a ticket

To sign a ticket, the ‘rsautl’ utility of OpenSSL is used:

```
openssl rsautl -inkey privatekey.pem -sign
```

This command reads the ticket from standard in and returns the signed ticket on standard output. As OpenSSL provides a base64 codec, above command can be directly piped to the following command, resulting in an ASCII encoded signed ticket:

```
openssl base64 –e
```

The output for the example signed ticket looks as follows:

```
t2676vNAaXJnMSGcriGKcp05Cqopgb/4lrE5aiNhpRvimZ8gYSlRe7ruZtvKpUTvExbxWEQNchriVGki
bOYUx6ulYN/p4o6fj2mqcJmCZ1AOkchA1k67TuxpvODqf+K0fswNNgoIUU9shZTqnU4fARhK6ICFcBMQ
fvRMexyA10Y=
```

5.2.3. Verifying a ticket

To verify the signed ticket, also the ‘rsautl’ utility is used, this time together with the public key:

```
openssl rsautl -verify -inkey publickey.pem -pubin
```

The signed ticket is expected on standard input, the clear text ticket returned on standard output. The return code indicates the successful verification (0 = ok). Again, the base64 decoder can be used to convert the ASCII input from the URL to the binary format required by ‘rsautl’:

```
openssl base64 –d | openssl rsautl –verify ...
```

5.3. Comparison

Both tools, GnuPG as well as OpenSSL, are well suited to implement the ‘signed ticket’ Single Sign-On system. GnuPG has some quirks when it comes to automation, as it’s been designed as an end-user tool. Most of the dependencies can however be overridden by corresponding command line options and environment settings. A drawback is that its ASCII output is not easily usable for embedding into URLs, as it contains additional headers and footers.

OpenSSL is very slick; it has virtually no dependencies on user and environment settings, uses few and well understandable command line options, and seems also to be faster. It generates slightly shorter output than GnuPG, probably because it does not add a timestamp to signed messages. For use in an automated environment, such as the

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

described Single Sign-On system, it is the preferred choice.

6. Conclusions

The goal of this paper was to develop a concept for a simple Single Sign-On solution for web applications. Starting from known authentication schemes for single applications, several methods of passing an authenticated session between multiple applications have been examined and elaborated upon. In order to facilitate implementation, existing Open Source tools have been evaluated and their suitability demonstrated. This paper can now serve as a basis for further review of the concept and, eventually, for a real-world implementation.

© SANS Institute 2004, Author retains full rights.

A. References

Cole Eric, Fossen Jason, Northcutt Stephen, Pomeranz Hal, “SANS Security Essentials with CISSP CBK”, Version 2.1, SANS Press, April 2003, Page 605

esecurityplanet, “Authorization and Single Sign-on Products”, Online Security Magazine, June 11 2003.
URL: http://www.esecurityplanet.com/resources/article.php/964361
(Feb. 18 2004).

Garman, Jason. “Single Sign-on for Your Web Applications with Apache and Kerberos”, Sept. 11, 2003,
URL: http://www.onlamp.com/pub/a/onlamp/2003/09/11/kerberos.html
(Feb. 18 2004).

GnuPG, “The GNU Privacy Guard”, Project Homepage, URL: http://www.gnupg.org/
(Feb. 18 2004)

Lee, Anne Hart, “Single Sign-On: Holy Grail of Holy Crap!”, SANS GSEC practical, Sept. 21 2003,
URL: http://www.giac.org/practical/GSEC/Lee_AnneHart_GSEC.pdf
(Feb. 18 2004)

OpenSSL, “The OpenSSL Project”, Project Homepage, URL: http://www.openssl.org
(Feb. 18 2004)

Posey, Brien, “Legacy Single Sign-On, A Competitive Analysis”, Intranet Journal, Oct. 14 2002, URL:
http://www.intranetjournal.com/articles/200210/pij_10_14_02a.html
(Feb. 18 2004)

RSA Laboratories, “Frequently Asked Questions About Today’s Cryptography”, Version 4.1, Chapter 2.1.1,
URL: http://www.rsasecurity.com/rsalabs/faq/2-1-1.html
(Feb. 18 2004)

Smith, Richard E. “From Passwords To Public Keys”, Pearson Professional Education, Oct. 2001

© SANS Institute 2004, Author retains full rights.