Client Data Storage

(a.k.a. Cryptographic Storage, Tamperproof Cookie)

Abstract

It is often desirable or even necessary for a Web application to rely on data stored on the client, using mechanisms such as cookies, hidden fields, or URL parameters. In all cases, the client cannot be trusted not to tamper with this data. The *Client Data Storage* pattern uses encryption to allow sensitive or otherwise security-critical data to be securely stored on the client.

Problem

Many Web application designs depend on the ability to store data on the client. From a security perspective, it is almost always preferable to store data on the server. But there are considerations other than security that may require client-side storage to be used:

- Load balancing across multiple application servers can be complicated by the need to share session data across servers. It is much easier to store the session data on the client and have the client provide that data with each request. Making the application servers stateless with respect to any particular client allows large sites, such as e-Bay, to be far more responsive.

- Many e-commerce sites find considerable value in tracking users' browsing habits across visits. But such tracking can represent a significant amount of data that must be stored on the server. Furthermore, the server will generally include large amounts of stale data that may no longer be associated with any browser. It is far more economical to store this data on the client.

- Some designs (including versions of the *Password Propagation* pattern) require that the user's password be stored on the client in order to effect single sign-on. Amazon.com's one-click shopping is an excellent example of this.

- Storing sensitive data, such as passwords and credit card numbers, on the client instead of the server would prevent a compromise of the Web site from compromising the sensitive data associated with every client.

- In order to access data stored on the server, the client needs to be able to identify that data using a session identifier. The session identifier is itself sensitive.

Whatever the reason for storing sensitive data on the client, the problem arises that the client cannot be trusted:

- If attackers manually inspect the contents of cookies and Web pages, they may be able to glean information about the operation of the Web site that could later be used to compromise the site.

- If a security-conscious individual notes that sensitive data is stored in the clear, the results can be a public relations disaster.

- If a user's machine is stolen or otherwise compromised, an attacker would be able to gain access to any sensitive client-resident data.

- Cross-site scripting attacks allow a remote Web site to gain access to cookies created by another site. Any sensitive data in the cookie could be compromised remotely. Similarly, sensitive data in a URL could be extracted from the browser's referring page.

- If an attacker is able to modify authentication data on the client, the attacker could assume the identity of another user and compromise that user's account.

- If an attacker is able to modify sensitive application data, they could manipulate the application into behaving improperly.

Solution

The *Client Data Storage* pattern uses encryption techniques to protect data that this stored on the client. Using encryption ensures that sensitive data will not be inadvertently revealed. Using message authentication codes ensures that the data cannot be tampered with on the client.

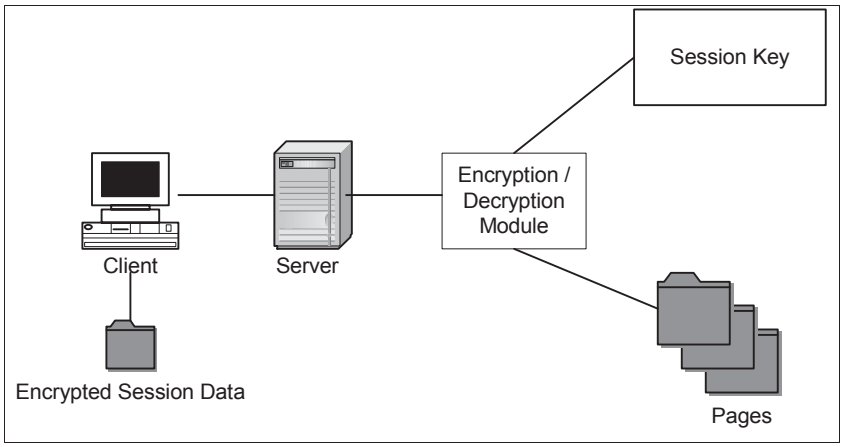
The *Client Data Storage* pattern employs these cryptographic techniques whenever sensitive data is to be delivered to the client. When this data is retrieved from the client it is decrypted and verified by a security checkpoint function. Any evidence of tampering is reported as a security-relevant event.

The *Client Data Storage* pattern uses a single symmetric key to encrypt and authenticate all data for all clients. When the technique is used to protect data within a single session, the key can be quite short, but should be rotated frequently. When the technique must protect long-lived client data across sessions, the key must be much stronger, as it will be very difficult to rotate keys without significant impact on either development cost or usability.

The most straightforward approach adds two functions on the application server: one to store sensitive data in the session object, and one to validate that data before it is used. When the user logs in, the authenticated username should be hashed using a session key stored on the server. The hash should be stored in the session data along with the username. When a protected page is requested, the server should recompute the hash of the username stored in the session data, and compare it to the hash stored in the session data. If the two match, the username can be trusted.

In order to protect against guessing attacks, the session key should be changed periodically. In general, changing the key once a day should be sufficient. In order to validate username hashes that span a change of keys, any failed hash comparison should be recomputed using the previous key as well. If the old key successfully validates the hash, the username should be accepted and a new hash created using the current key. In order to protect the key against brute force guessing attacks, repeated incorrect hashes submitted from a single network address should result in that

address being added to the network address blacklist. If the *Network Address Blacklist* pattern is not used, these events should be closely monitored by system administrators.



Issues

Many of the issues are identical to those of the *Encrypted Storage* pattern.

The following general principles should be followed:

- Never attempt to invent an encryption algorithm. Use a tested algorithm from *Applied Cryptography*.
- If possible, use a freely available library rather than coding one from scratch.
- After sensitive data is used, the memory variables containing it should be overwritten.
- Ensure that sensitive data is not written into virtual memory during processing.
- Use only symmetric encryption algorithms. Asymmetric (public/private) algorithms are computationally expensive and could easily result in processor resources being exhausted during normal usage.

Protection of the Key

If at all possible, the key should not be stored on the file system. There are COTS devices available that provide the system with a physical encryption card. These devices offer performance benefits and also guarantee that the key cannot be read off the device. The key is manually loaded into the card, the encryption takes place on the card, and the key cannot be extracted from the card. The only downside to this approach is the cost – both the cost of purchasing the hardware and the development and maintenance costs of programming around it.

A cheaper alternative to loading the key is to require that an administrator load the key at system start, either from removable media or using a strong passphrase. This reduces the risk of having the key on-line but does expose the key to a fair number of different people. This approach

might also sacrifice some availability because an operator must manually intervene whenever the system restarts.

If neither of these approaches is feasible, the Web server can read the key value from a file at server startup. This approach ensures that the server can restart unattended but puts the key at risk if the system is compromised. To reduce this risk, one or more of the *Server Sandbox* pattern techniques should be used, even going so far as to chroot the server so that it cannot see the key file once it has completed its initialization process.

Unless a hardware encryption device is used, the server will have to maintain a copy of the encryption key in RAM in order to decrypt data as it is needed. The code modules that have access to the key should be minimized. And if the operating system supports it, mark the decryption module so that it will never be swapped to disk. Also be aware that a coredump file might contain the key – these should be disabled on a production server.

In addition to protecting the key from attackers, the key must also be protected from conventional loss. The loss of the key would be catastrophic, since all user data would become inaccessible to the server. Maintain multiple backups of the key at various off-premises locations. Recognize that multiple keys increase the risk that one could be stolen, and take care to protect them all.

Possible Attacks

There are a number of possible attacks that could be perpetrated against this pattern:

- If the data is not sufficiently protected, an attacker may be able to decrypt the data via a brute force attack.
- An attacker that compromises the server may be able to gain access to the global encryption key.
- Cross-site scripting can be used to force the browser to reveal cookies that are intended for a different site.

Examples

Many application servers use cryptographic techniques to generate suitably random session identifiers that are stored on the client

Many large Web sites store encrypted cookies on the client's browser. They include Amazon.com, Buy.com, and e-Bay. Because they are encrypted, it is impossible to know precisely how they are being used.

Trade-Offs

Accountability	No effect.

Availability	Availability could be adversely affected if encryption keys are lost. Also, if a global key must be reentered by a human operator each time the system restarts, availability will suffer if an operator is not available to restart the server.
Confidentiality	This pattern increases confidentiality by ensuring that the data is secure, even if it has been captured in an encrypted form.
Integrity	This pattern helps to ensure the integrity of client data and processing that depends on that data. Any data tampering on the client can be detected.
Manageability	If system administrators must intervene to provide a global password on system restart, this will require around-the-clock administration.
Usability	No direct effect.
Performance	This pattern will have a negative impact on performance because of the processing burden of encrypting and decrypting every message.
Cost	Costs will be incurred from additional processing required to compensate for performance loss and the cost of adding encryption / decryption logic to the application.

Related Patterns

- *Client Input Filters* – a related pattern that verifies all data coming from the client.
- *Encrypted Storage* – a related pattern utilizing encryption to protect confidential data.

References

[1] Landrum, D. "Web Application and Databases Security". http://rr.sans.org/securitybasics/web_app.php, April 2001.