Encrypted Storage

(Cryptographic Storage)

Abstract

The *Encrypted Storage* pattern provides a second line of defense against the theft of data on system servers. Although server data is typically protected by a firewall and other server defenses, there are numerous publicized examples of hackers stealing databases containing sensitive user information. The *Encrypted Storage* pattern ensures that even if it is stolen, the most sensitive data will remain safe from prying eyes.

Problem

Web applications are often required to store a great deal of sensitive user information, such as credit card numbers, passwords, and social security numbers. Although every effort can be taken to defend the Web server, one can never be sure that some new vulnerability won't be discovered, leading to the compromise of the server. Hackers are known to specifically target this sort of information.

Historically, Web sites that have experienced the loss of sensitive customer data have found it very difficult to recover from the adverse publicity. While many sites have recovered from the shame of being defaced, the large-scale loss of credit card numbers is a catastrophic failure.

Ultimately, it is always preferable not to store sensitive data. However, sometimes it is not avoidable. For example, credit card transactions are often not a single event. If an item is back ordered or the user requires a refund, the site must be able to access the credit card number that was used. Similarly, many government and financial sites rely on the social security number as the primary identifier for American users. These sites need a better approach to protecting this data.

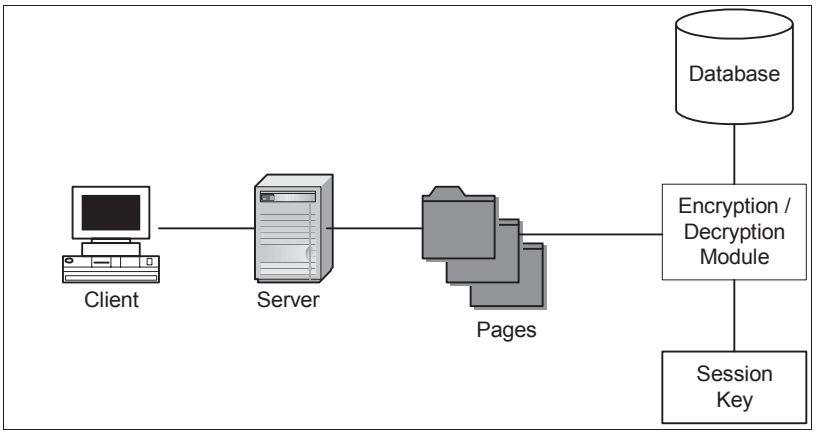
Solution

The *Encrypted Storage* pattern encrypts the most critical user data before it is ever committed to disk. Before it can be used, it is decrypted in memory. If the Web server is compromised, an attacker may be able to steal the data store, but will not be able to gain access to the sensitive data.

In the most straightforward approach, each user's data is protected using a single key. Under this solution, the application server maintains a single key that is used to encrypt and decrypt all critical user data. The key should be stored in a secure fashion, and especially not in the same data store as the protected data.

The key should be changed occasionally. This requires that the system be able to decrypt data using the old key and re-encrypt it using the new. Because of the complexity of encrypting and decrypting data on the fly, this should be performed with the database off-line during a period of

downtime. If downtime is not possible, a large key should be selected with the expectation that it will not be changed.



Server startup:

- The server loads the key into the encryption module
- The server takes protective measures to ensure that the key cannot be further accessed

Receipt of sensitive data:

- The client submits a transaction containing sensitive data
- The server submits the data to the encryption module
- The server overwrites the cleartext version of the sensitive data
- The sensitive data is stored in the database with other user data and an identifier for the sensitive information

Use of sensitive data:

- A transaction requiring the key is requested (usually from the client)
- The transaction processor retrieve the user data from the database
- The sensitive data is submitted to the encryption module for decryption
- The transaction is processed
- The cleartext sensitive data is overwritten
- The transaction is reported to the client without any sensitive data

Key refreshing:

- A utility program is started and loaded with both the old and the new key
- Each user record in the database is converted individually.

Issues

Never echo the sensitive data to the user. If you need to differentiate among several credit card numbers, display only the last four digits of the card. These should be stored in the database along with the encrypted card number. Both performance and security could suffer if the card numbers are decrypted every time the last four digits are required.

Do not rely on any Operating System-level encrypting file system. Encrypting file systems are adequate for defending against a lost hard drive. But if the system is compromised by a remote attacker, the attacker will gain some sort of toehold on the system. In that case, the operating system will dutifully decrypt all data as it is requested from the file system and deliver it to the attacker.

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.
- Care must be taken to insure that sensitive data is not written into virtual memory during processing.
- Use only symmetric encryption algorithms. Asymmetric (public/private) algorithms are too 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 may 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, use one or more of the *Server Sandbox* pattern techniques, even going so far as to chroot the server so 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. Minimize the code modules that have access to the key. 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.

Variation: One Key Per User

This alternative is similar to the *Password Propagation* pattern in that it requires that the individual user's password be available in order to gain access to that user's data. The server itself does not even have a key that will allow access to a user's data. It is not really applicable to the protection of credit card numbers, as those numbers must be available to the server even when the user is not connected.

In this approach, the user's password is used to encrypt the data that is sensitive to that user. To decrypt the data, the user must again provide their password, which is never stored in decrypted form. Because decryption of the data requires the user to provide his/her password, and because that password is not known outside of the context of an authenticated user transaction, the site administrator has no access to that data.

If the password itself is stored in the data, it should be stored in hashed form, using a different algorithm than the hash function used to encrypt the sensitive data. If the password is stored in plaintext or hashed using the same algorithm, the attacker will have the key needed to decrypt the data.

If the user changes his/her password, the data must be decrypted using the old password and re-encrypted using the new. If the user loses his/her password, encrypted data will be lost. Data protected in this way must be data that can be recovered through some other means, such as the user providing it again.

Possible Attacks

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

- Search of virtual memory – if sensitive data is paged out of RAM into a disk-based virtual memory file, it may be possible for an attacker to search the pagefile for obvious data

patterns (such as numeric strings that are recognized as credit card numbers).

- Key capture – an attacker will attempt to gain access to the key used to encrypt the data
- Dictionary attack – when encryption keys are generated from passwords, password-guessing attacks are generally much less difficult than exhaustive search of all possible keys.
- Race condition attack – an attacker may be able to capture the data before it has been encrypted.

Examples

The UNIX password file hashes each user's password and stores only the hashed form.

Several Web sites with which we are familiar use encryption to protect the most sensitive data that must be stored on the server. All use variations on this pattern.

Trade-Offs

Accountability	No effect.
Availability	Availability could be adversely affected if encryption keys are lost. If a global key must be reentered by a human operator each time the system restarts, availability can suffer in those circumstances as well.
Confidentiality	This pattern increases confidentiality by ensuring that the data cannot be decrypted, even if it has been captured.
Integrity	No direct effect.
Manageability	If system administrators must intervene to provide a global password on system restart, this will require around-the-clock administration. If individual passwords are used to encrypt data, administration will be complicated when users lose their passwords, or if other access to user data is required.
Usability	If individual keys are used, usability will suffer when a user loses his/her password and consequently loses all encrypted data.
Performance	This pattern will have a slight impact on performance due to the encryption algorithms and extra storage logic required.
Cost	Costs will be incurred from additional processing power required, additional management overhead, and the cost of adding encryption and decryption logic to the application.

Related Patterns

- *Client Input Filters* – a related pattern that verifies all data coming from the client.

References

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