1 Motivation

There are many types of data that require protection from unauthorized access or modification. These include financial and medical records, commercial proprietary and competitive information, and government or military security data. There are also many sources of threats to this data, including organized crime, governments, terrorists, competitors, the mass media, disgruntled or dishonest employees, and recreational hackers. In some cases laws such as the Health Insurance Portability and Accountability Act of 1996 (HIPAA), the Financial Modernization Act of 1999 (Gramm-Leach-Bliley), or the Sarbanes-Oxley Act of 2002 mandate specific protections for certain data. In other cases there is concern about the risk of civil liability on the grounds of negligence if sensitive data is compromised. In all cases ethics, due diligence, and simple common sense dictate that appropriate measures should be taken to protect sensitive data.

Protecting sensitive data is a complex task, requiring a layered approach to information security. Consider the following three venues in which data can be compromised.

1.1 Data in Use

This refers to data in memory, either process-private or shared, that is available for use by an application, either stand-alone or as a server. The host operating system provides one level of protection, preventing other local system users from accessing memory containing sensitive data, or in some cases preventing such users from accessing the host system at all. The application itself can also provide data protection, in most cases by authenticating the identity of users and enforcing access control (authorization) on data based on that authenticated identity. User access can occur in local mode, connection-oriented client/server mode, or stateless request/response mode (such as a Web service). Caché provides both Kerberos authentication [Kerberos] [RFC 4120] and simple password options for user authentication in all access modes, and a highly configurable authorization mechanism for controlling access to data (as well as for other important resources). (Note that the Kerberos protocol also provides protection for data in motion; see below.)

1.2 Data in Motion

This refers to data in transit on a communications network, between clients and servers or among cooperating servers. This data is vulnerable to both passive attacks (sniffing) and active attacks (modification or injection). There are several high quality, freely available tools [Ethereal] [dsniff] that make such attacks easy to mount. Protection requires guarantees of both source and data integrity, usually provided by an encrypted or keyed message authentication code, and data
privacy, usually provided by encryption. Caché provides Kerberos data integrity and confidentiality services [Kerberos] [RFC 4120] to satisfy these requirements (in conjunction with SSL [SSL] for Web services). (Note that the Kerberos protocol also provides user authentication to support protection of data in use; see above.)

1.3 Data at Rest
This refers to data stored on non-volatile media, such as disks or tape. The main concern is usually loss or theft of physical media, although attention must also be paid to the eventual fate of discarded media, and to the possibility of surreptitious copying of media. The usual solution is data encryption. The remainder of this paper describes Caché’s built-in block-level database encryption solution to the problem of protecting data at rest.

2 Design Goals
In order for a data encryption solution to be both effective and useful, certain critical requirements must be met. First, it should provide security that is strong by the standards of modern cryptography. Cryptographic algorithms selected should, whenever possible, have been reviewed by the cryptographic community and judged resistant to all known cryptanalytic techniques, now and for decades to come, despite anticipated advances in cryptanalysis and computing power. Keys should be long enough to provide immunity from brute-force attacks. Chained cipher modes should be chosen to minimize information leakage, and any initialization vectors used should be generated in a way that does not compromise the advantages of the chosen cipher mode. Finally, key management techniques must provide protection for encryption keys at least as strong as that provided for the underlying encrypted data.

Equally important, the solution cannot have a disproportionate impact on database performance. Although it must be recognized that a sophisticated data encryption capability cannot operate without using some resources, that resource consumption must be relatively small compared to the other requirements of the application. The impact needs to be evaluated both in terms of data throughput and read/write latency.

Encryption of sensitive data can be applied at various levels of granularity. The following table shows how those levels are reflected in the SQL and Objects programming models.

<table>
<thead>
<tr>
<th>SQL</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>database</td>
</tr>
<tr>
<td>table</td>
<td>class</td>
</tr>
<tr>
<td>column</td>
<td>property</td>
</tr>
<tr>
<td>row</td>
<td>instance</td>
</tr>
</tbody>
</table>

A flexible solution supports all these levels of granularity, as well as others appropriate to available proprietary programming models, such as Caché sparse multi-dimensional arrays. The complete solution should be transparent to application code. While no data encryption technology can prevent modification of the encrypted data, it can provide assurance that data modification will not go undetected. Finally, the solution should be platform independent, not just in terms of availability, but also in terms of the portability of encrypted data. (Note that Caché is available for AIX, HP-UX, Solaris, Tru64, Linux, Windows, MacOS, and OpenVMS.)
3 Architecture

A database application can be viewed as a set of layered components, each with different functions and responsibilities.

<table>
<thead>
<tr>
<th>Application</th>
<th>Database</th>
<th>Filesystem</th>
<th>Disk Driver</th>
<th>Disk Hardware</th>
</tr>
</thead>
</table>

The Application layer implements what is often called the “business logic”, where knowledge of the semantics, or meaning, of the data resides. The Database layer is responsible for indexing, searching, sorting, and modifying data, and organizing it for efficient storage on disk. It can provide a variety of programming interfaces to the Application layer, such as SQL statements (SELECT, INSERT, UPDATE, DELETE, etc.), object properties and methods, or Caché sparse multi-dimensional arrays. The Filesystem layer maintains a storage hierarchy (directories), file metadata (ownership, permissions, dates, etc.), and is responsible for allocating physical storage. It generally provides a programming interface to the Database layer using system functions (open, seek, read, write, close, etc.). The Disk Driver layer provides an interface to the data storage hardware, and may also interact with performance optimizations such as a buffer cache. It provides a programming interface to the Filesystem layer using function calls similar to those provided by the Filesystem layer. The Disk Hardware layer includes the disk controller and actual physical non-volatile storage media.

Data encryption can be applied at any of the interfaces in this hierarchy, and the resulting functionality depends substantially on what choice is made. Encryption at the Application/Database interface can be implemented explicitly as part of the Application code, built into the Database layer, or added to an existing Database interface using triggers or some similar mechanism. There are a number of commercial products available that use this approach [NWC 1] [NWC 2]. There are several technical problems introduced by the fact that encryption at this level must be able to deal with variable-sized (and often small) data elements. Encryption must either use a cipher mode that does not require initialization vectors (which is bad cryptographic practice), such as Electronic Code Book (ECB) [NIST 800-38A]; or use fixed initialization vectors (which is almost as bad). Otherwise, storage must be allocated for the initialization vector for each encrypted data element, which alters the size and structure of the database. The computational overhead of setting up encryption (and possibly generating initialization vectors) for small data elements can have a substantial impact on performance. In addition, encrypted data elements generally cannot be indexed.

Another option is to encrypt data at the Filesystem/Disk Driver interface. This can be integrated into the operating system, but that option is available for only a limited subset of platforms, e.g. Windows [EFS 1] [EFS 2], Mac OS X [FileVault], or Linux [CryptoLoop] [dm-crypt] [Loop-AES]. The resulting encrypted data is completely non-portable across platforms. Encryption can also be performed at the Disk Driver/Disk Hardware interface, either by a hardware appliance in a storage-area network [NWC 3] or by the disk controller [Seagate]. (The latter option currently has very limited availability.) In either case this option provides complete application transparency, but is generally usable only at the highest (database) level of granularity.

A final option is to encrypt at the Database/Filesystem interface. This must be implemented by the Database vendor, since it requires knowledge of the underlying data storage strategy, and
must be integrated into the core of the database engine. Since data encryption and decryption are performed only when data is written to or read from the Filesystem, initialization vectors can be computed from existing metadata (preserving database structure and size), and encryption setup can be amortized over large block reads (minimizing performance impact). This option also provides complete application transparency, and knowledge of the data storage strategy can be used to provide encryption at any level of granularity. Since the encryption mechanism is part of the Database layer, encrypted data is completely portable across platforms. This is the architectural choice used in Caché block-level database encryption.
4 Implementation

Caché block-level database encryption was designed to meet the design goals outlined in Section 2. See Appendix A for a complete table of implementation specifications and applicable standards.

4.1 Database Encryption

Caché stores data in multiple files (or UNIX raw disk partitions) called databases. All database reads and writes occur in 8192-byte blocks. This includes application data as well as metadata such as indices, bitmaps, pointers, allocation maps, and incremental backup maps. All blocks (with the exception of a single initial label block) are always encrypted on disk. Encryption is enabled on a per-database basis, allowing for a flexible mix of encrypted and unencrypted data. Encryption at any level of application granularity can be achieved using multiple databases with Caché namespace global mapping and SQL storage retrieval code.

The actual encryption and decryption is done at a low level in the Caché system code, and uses the United States Government Advanced Encryption Standard (AES) [FIPS 197] in Cipher Block Chaining (CBC) mode [NIST 800-38A], with a 128-, 192-, or 256-bit database encryption key. The initialization vector for each block is generated by encrypting a big-endian representation of the logical block number using AES in Electronic Codebook (ECB) mode [NIST 800-38A], using the database encryption key. This provides a unique initialization vector for each block, prevents information leakage about identical blocks, and removes any predictable relationship among the initialization vectors for consecutive blocks, preventing known watermarking attacks [Saarinen].

![Diagram 1](image1.png)

Figure 1. Writing to an encrypted database.  

![Diagram 2](image2.png)

Figure 2. Reading from an encrypted database.

4.2 Scope of Protection

Data from encrypted databases is also encrypted whenever it is stored on disk by Caché for auxiliary purposes. This includes the write image journal file, preimage journal file (OpenVMS
only), and, optionally, regular journal files, the Caché temporary working database, and the auditing database. Backups of encrypted databases are done at the file level, preserving the existing encryption. Note that Caché cannot control the copying of memory containing sensitive data to an operating system swap file.

4.3 Key Management

The key management mechanism is designed to provide multi-factor protection for encrypted databases. Multiple encrypted copies of the database encryption key are stored in a key file, each for use by a different database encryption administrator, and each encrypted with a unique key encryption key derived from a passphrase used by that administrator. The key encryption is done using the AES Key Wrap algorithm [AESKW] [RFC 3394] approved by the United States Government’s Security Requirements for Cryptographic Modules, Annex D, Approved Key Establishment Techniques [FIPS 140-2D], and the key encryption keys are derived from the administrator passphrases using Password-Based Key Derivation Function 2 (PBKDF2) [PKCS #5v2] [RFC 2898] with HMAC-SHA-1 as the pseudorandom function [FIPS 180-2] [FIPS 198] [RFC 2104] [RFC 3174]. It therefore requires possession of both the encrypted database file and the key file, plus knowledge of a valid administrator passphrase, to access encrypted data. Entries for database encryption administrators can be added to or removed from the key file as needed. Encrypted databases and their associated key file should always be stored on physically distinct devices. Note that the key file is needed only when activating the database encryption key for a Caché configuration. The device containing the key file can be securely stored while Caché is running. A removable storage device such as a USB flash drive is well suited for this purpose.

It has been estimated that there are approximately 50 million easily guessed English-language passphrases, including proper names, spelling variations, character substitutions, etc. [Perrine and Kowatch] [Perrine] Both salting and stretching are employed to make attacks against poorly chosen passphrases more difficult [Ferguson and Schneier]. Stretching is done by repeating the basic PBKDF2 primitive operation many times for each key generation operation, in order to make each attempt more time-consuming. Salt is a unique random number that is processed along with a passphrase to generate a key encryption key. This technique prevents an offline pre-computation and storage dictionary attack, since each easily guessed passphrase maps to a large number of key encryption keys.
Figure 3. Storing an encrypted database encryption key in a database encryption key file.

Figure 4. Recovering the database encryption key from a database encryption key file.
4.4 Random Numbers

A cryptographically strong source of random numbers is needed in order to generate database encryption keys and PBKDF2 salt, since the security of the system depends on the unpredictability of these values. A cryptographic pseudorandom number generator has the property that knowledge of any of its output yields no information about its internal state or any other part of its output. (This is in contrast to the more familiar linear congruential generators [Knuth].) Such a generator needs to have its initial internal state seeded from a source of true physical randomness, or *entropy*, which is then processed by a *mixing function* to produce usable random bytes. Different mechanisms for gathering entropy are available depending on the underlying operation system. (See Appendix A.) The mixing function used is approved by the United States Government’s *Security Requirements for Cryptographic Modules, Annex C, Approved Random Number Generators* [FIPS 140-2C], and is fully described in *Digital Signature Standard (DSS)*, Appendices 3.1 and 3.3 [FIPS 186-2] and *Secure Hash Standard* [FIPS 180-2] [RFC 3174].

5 Operational Policy Recommendations

Since the database encryption key file is the only source from which the database encryption key can be recovered, loss of all copies of that file would render an encrypted database permanently unreadable. Since the database encryption key is stored in encrypted form only, and the key encryption keys are not stored anywhere and can only be derived from database encryption key administrators’ passphrases, loss of all passphrases would be similarly catastrophic. In order to avoid permanent data loss there should always be more than one administrator record in the key file, and there should always be more than one copy of the key file, on physically distinct media. This can be achieved by always adding an emergency recovery administrator entry to the key file when it is first created, copying the key file to a removable storage device such as a USB flash drive or CD-ROM, and storing that backup copy along with a written record of the emergency recovery administrator passphrase in a physically secure and distant location, such as a bank vault.

The database encryption key can be activated manually at any time, or Caché can be configured to activate the database encryption key at startup. In the latter case, Caché’s temporary database (CacheTemp) can be encrypted, since it is re-created each time Caché starts.

5.1 Datacenter Operation with Continuous Operator Coverage

This deployment scenario provides the maximum level of security. The key file used for regular operation should be stored on a removable storage device such as a USB flash drive or CD-ROM, which is in turn stored in a secure location such as a safe. That location should be under the control of personnel different from the database encryption administrators, such as building security. The database encryption administrators then require the cooperation of those other personnel in order to access the device, eliminating reliance on the dependability of single individuals. Since the key file is only needed when activating the database encryption key for a Caché configuration it can be safely stored during normal operation.

5.2 Unattended Datacenter Operation

Some installations require the use of database encryption but do not have continuous operator coverage. In these cases Caché can be configured to activate the database encryption key using a key encryption key derived from a special long, random, binary (unprintable) passphrase. When using this configuration it is especially critical to ensure that the database encryption key file is not stored on the same medium as any encrypted databases, and that it does not contain any
entries for human administrators. Ideally, the key file should be on a medium that can be physically locked in place, such as a lockable CD-ROM drive in a rack, and the datacenter facility should be locked and monitored.

5.3 Personal Workstations and Laptops

In most cases individual users will serve as their own database encryption administrators. USB flash drives are a convenient, portable storage medium for the database encryption key file. Care must be taken never to leave the computer and key file together and unattended.

6 Performance Impact

There are two ways database encryption can impact performance: CPU consumption and disk read/write latency. The CPU processing requirement for Caché block-level database encryption in software (using 128-bit keys) has been measured at approximately 12 CPU cycles per byte, on popular modern processors. Note that only physical disk writes and reads require encryption or decryption. Given knowledge of an application’s disk read rate (obtained using the ‘GLOSTAT utility), the average length of the Write Daemon queue at the beginning of each pass (obtained using the ‘mgstat utility), the fraction of the data that is encrypted, and the clock speed of the processors it runs on, it is straightforward to estimate the processing requirement for encryption and to compare that to either the other measured processing requirements of the application or the total processing capacity of the system.

Caché application processes access data from global buffers stored in shared memory. Data accesses that require an encrypted database block to be read from disk to a global buffer experience decryption latency. Since Caché uses write daemons to move data from global buffers to disk, application processes experience no encryption latency on writes.

An example will illustrate how to estimate these effects. Consider an application running on a server with 3 GHz processors, and writing on average 200,000 encrypted database blocks at each Write Daemon pass. The processing time for a single processor encrypting the WIJ would be

\[
\frac{2 \times 10^5 \text{ blocks} \times 8192 \text{ bytes block}}{3 \times 10^9 \text{ cycles second}} \times 12 \frac{\text{cycles}}{\text{byte}} = 6.5 \text{ seconds}
\]

Data accesses that require an encrypted database block to be read from disk to a global buffer experience per-block decryption latency of

\[
\frac{8192 \text{ bytes block}}{3 \times 10^9 \text{ cycles second}} \times 12 \frac{\text{cycles}}{\text{byte}} = 0.03 \text{ milliseconds}
\]

This should be compared to the 2 millisecond average rotational latency of a high-performance 15,000 RPM disk drive.
7 Vulnerability Analysis

7.1 Captured Database
If an attacker acquires a copy of an encrypted database without an associated key file, the only known attack is a brute-force trial of all possible database encryption keys. (Caché databases contain a sufficient amount of predictable internal structure that incorrect keys can be identified after a very small number of block decrypts.) To assess this risk, consider the Electronic Freedom Foundation’s “DES Cracker”, built in 1998 at a cost of $250,000, and capable of testing 100 billion keys per second. Assume that a similar machine can be built to attack AES; that Moore’s Law continues to hold, so that by 2028 such a machine will be a million times as fast; and that an attacker is willing to spend $250 million, for another thousand-fold speed increase. Using this expensive machine from the future it would take 10 billion years to test 10% of all possible 3x10^{38} 128-bit keys. (The current age of the universe is approximately 14 billion years.)

7.2 Captured Database and Key File
If an attacker captures both a copy of an encrypted database and an associated key file, the security of the data depends entirely on the strength of the passphrases used to generate the key encryption keys. If the only entry in the key file is one used for unattended startup then no advantage is gained, since that uses a random passphrase equal in length to the database encryption key itself. If, however, the key file contains entries for administrators using easily guessed passphrases the vulnerability can be considerable. Using the default values for the PBKDF2 parameters, it takes approximately 0.2 seconds to generate a key encryption key from a guessed passphrase with known salt on a 3 GHz processor. This would allow 50 million easily guessed passphrases to be tried in only 4 months. Note that the situation is considerably better if administrators use eight character passphrases containing a mix of upper and lower case letters, numbers, and punctuation (and not derived from actual words). This increases the number of possible passphrases to 6x10^{15}, and the attack time to 40 million years. Note, however, that these passphrases would still be vulnerable to attack by a hypothetical cracking machine like that described above, which underscores the importance of keeping key files separate from encrypted databases.

7.3 Captured Database and Key File with Pre-computed Keys
If salt were not used with PBKDF2, each passphrase would always uniquely map to a specific key encryption key. A list of all the 128-bit, 192-bit, and 256-bit key encryption keys derived from 50 million easily guessed passphrases could be pre-computed (along with their round key schedules) and stored in only 32 GB. These could then be rapidly tested against the encrypted database encryption keys in a captured key file. Use of salt equal to or greater in length than the key itself makes this attack less effective than simply brute forcing the database encryption key.
## Appendix A: Specifications

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Component</th>
<th>Value</th>
<th>Applicable standards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database encryption</td>
<td>algorithm</td>
<td>AES</td>
<td>[FIPS 197]</td>
</tr>
<tr>
<td></td>
<td>key length</td>
<td>128, 192, or 256 bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cipher mode</td>
<td>CBC</td>
<td>[NIST 800-38A]</td>
</tr>
<tr>
<td></td>
<td>initialization vector</td>
<td>encrypted logical block number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plaintext padding</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Key encryption</td>
<td>algorithm</td>
<td>AES Key Wrap</td>
<td>[AESKW] [RFC 3394]</td>
</tr>
<tr>
<td></td>
<td>key length</td>
<td>128, 192, or 256 bits</td>
<td></td>
</tr>
<tr>
<td>Key encryption key derivation</td>
<td>algorithm</td>
<td>PBKDF2</td>
<td>[PKCS #5v2] [RFC 2898]</td>
</tr>
<tr>
<td></td>
<td>pseudorandom function</td>
<td>HMAC-SHA-1</td>
<td>[FIPS 180-2] [FIPS 198] [RFC 2104] [RFC 3174]</td>
</tr>
<tr>
<td></td>
<td>iterations</td>
<td>65,536</td>
<td></td>
</tr>
<tr>
<td></td>
<td>salt length</td>
<td>512 bits</td>
<td></td>
</tr>
<tr>
<td>Cryptographic PRNG</td>
<td>entropy gathering</td>
<td>/dev/urandom (UNIX), Microsoft Cryptographic Service Providers (Windows), process statistics (OpenVMS)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mixing function</td>
<td>DSS w/ SHA-1</td>
<td>[FIPS 140-2C] [FIPS 180-2] [FIPS 186-2] [RFC 3174]</td>
</tr>
</tbody>
</table>
References

Note:
Standards are available online at the following URLs:

[Ferguson and Schneier]  Ferguson and Schneier, Practical Cryptography, 2003
Annex C, Approved Random Number Generators, March 2003
Annex D, Approved Key Establishment Techniques, February 2004
Change Notice 1, Appendix 3.1, Algorithm for Computing m Values of x, October 2001
Appendix 3.3, Constructing the Function G from the SHA-1, January 2000
[NIST 800-38A]  Recommendation for Block Cipher Modes of Operation, National Institute of Standards and Technology Special Publication 800-38A, December 2001
[Perrine] Perrine, “the end of crypt() passwords...please?”, ;login:, December 2003

[PKCS #5v2] Password-Based Cryptography Standard, Public-Key Cryptography Standard #5
v2.0, March 1999


[RFC 3174] Eastlake and Jones, US Secure Hash Algorithm 1 (SHA1), September 2001

[RFC 3394] Schaad and Housley, Advanced Encryption Standard (AES) Key Wrap Algorithm, September 2002


[Saarinen] Saarinen, Linux for the Information Smuggler
(http://mareichelt.de/pub/notmine/diskenc.pdf)


(http://wp.netscape.com/eng/ssl3)