VOCAL

# Whitepaper: Design and Implementation of Custom Cryptosystems

Document #W930_002

Aug 26, 2008

## Table of Contents Page

## 1 Introduction

This white paper examines some of our experience in the cryptography field, in particular the creation of custom cryptographic systems for custom applications.  We focus on easy-to-overlook issues and critical aspects of custom cryptosystem design, with an emphasis on constraints found in embedded systems.

We have found it convenient to separate cryptosystem design and implementation into two distinct sets of operations.  The first is the bulk data transport, which uses keys to secure underlying data.  The second is the keying mechanism, which provides keys to the bulk data transport. This separation of duties helps limit the scope of each, and fits well with the requirements of most cryptographically protected systems.

For consumer applications, the boundary may serve only as a well-defined interface point.  For high security and governmental applications, it may be mandatory for key generation and transfer to reside on an external device.  For military security systems, the key generation and transfer mechanisms must always be separated from the data layer.

For each part of the system, different design processes may be used.  This white paper will cover some of the more common issues and problems encountered as part of the design process.

## 2 Caveats and Warnings

Selection of proper primitives for your application is by far the most important decision to make, and selection should be done early on in the design process to allow the security implications to be discovered and understood. There are vast bodies of literature describing cryptographic primitives and algorithms; we recommend investigating widely deployed systems similar to yours to determine which primitives might be suitable for your application. Some of the attacks and security analysis for the other system may also impact your design decisions.

It goes without saying that you should not attempt to design or construct your own low level cryptography primitives. We say it anyway: use only analyzed, well-understood and widely recognized primitives. If you must modify a primitive, do so with full understanding of the risks involved and full peer review and discussion. Even seemingly trivial changes can compromise your system.

A less-well-recognized implication of this concerns random number generators. Much as you should not attempt to build your own cryptographic primitives, you should never attempt to build your own PRNG for use in a cryptographic system. Many cryptographic systems rely on hard, cryptographic grade PRNGs for proper operation. Use of a poor PRNG can compromise utterly or or greatly reduce the required search space for cryptographic keys.

For secured hardware, the data layer may need to be protected to prevent extraction of key material. Physical access controls are a critical part of most system designs, but are beyond the scope of this white paper.

## 3 Attack Vectors

This list of attack vectors is intended to describe only general classes of attacks that any cryptographic system must be resistant to:

- Replay attacks - can previously recorded data be played back into the system to cause damage? Note that even if the data in the recording is unknown, this can still cause damage, for example by replaying control messages or any other data that lacks timestamp checks.

- Modification attacks - change an unknown message into another unknown message. This can be particularly damaging if some aspect of the message can be deduced via other means; for example if a seldom used field in a known location is almost always zero, changing it to 1 to affect the receiving system is a trivial operation if the data is not integrity checked.

- Spoofing attacks - make it look like data is coming from the wrong location.

- Denial of service attacks - prevent valid data from reaching its destination, typically through a protocol flaw.

- Precomputation attacks - precompute and archive some portion of the algorithm to greatly reduce the search space or make a timing-based attacks possible.

- Direct/brute force attacks - is your keyspace large enough to make brute force attacks infeasible? Does your implementation preserve all of the keyspace? Even a 128 bit AES primitive can be brute forced if only half of the keyspace need be searched.

- Infrastructure attacks - if aspects of your system rely on other infrastructure to operate correctly, attacks against that infrastructure may compromise or expose that system. One example might be global timestamps; if the timestamp synchronization messages can be hijacked and manipulated, many attack vectors become possible. By controlling the timestamps, an attacker may open windows for replay attacks, may invalidate keys causing DOS, or even cause key material to be used outside its appropriate lifetime.

For each of these major classes of attacks, there may be many more specific attacks that are relevant to your system. All of these, and even attacks that are only marginally relevant to your current design, should be considered for any part of your cryptosystem. A detailed database or document that lists each attack vector and the results of attack analysis should be kept current at all times, with updates any time a change must be made to the design.

Analyzing the impact of all design changes for relevant attacks is time consuming, which makes it all the more critical that the design be completed early in the project lifetime.

# 4 General Observations

Practical security can almost never be perfect; while certain aspects of the system may be provably secure, there is almost invariably some aspect of the system which is vulnerable to attack. The goal should not be to provide provable perfect security for the entire system; the goal should instead be to make all known attacks sufficiently difficult as to be infeasible for the lifetime of the system.

With the speed of advancement of modern technology, this can require difficult engineering tradeoffs to be made. What is the estimated lifetime of the system? How long should secured data remain secured after lifetime end? Will future products require compatibility, extending the effective lifetime of the cryptosystem?

Extrapolating Moore's Law and the impact of cryptographic research, how secure will the cryptosystem be in ten years, or twenty years, or thirty years? How will the system fare against an attacker with a billion times the processing power and storage available today? These questions can never be answered with certainty, but so long as the security of the primitives remains mostly intact they can be addressed. The results should be a part of the overall security analysis, as they will provide a minimum upper bound on the required key lengths.

You can't just focus on one particular aspect of the system and declare the system secure; the security profile of the entire system must be taken into account. Any weak link can compromise the system. Even the interaction of the system with users should be well understood; with modern cryptographic systems, social engineering and attacks against the physical hardware are often far more effective than algorithmic attacks. Your design should be aware of such attacks, even if the best you can do is to only slightly reduce the attack window.

The following sections contain some general rules of thumb and things to consider when designing and implementing a specific system.

## *4.1 Always Authenticate*

Authentication and integrity check of data should be considered mandatory for any cryptosystem and should be applied to data at all levels. A large number of attack vectors exist that exploit weaknesses in unauthenticated cryptosystems, including spoofing, modification, and replay attacks.

An integrity check serves only to guarantee that the data sent by the encrypting party was decrypted properly without error or modification in transit. Depending on which headers are part of the integrity check, this may also verify some aspect of the sender. A good integrity check can validate both encrypted data and unencrypted data (such as packet headers) without risk of data or key exposure.

Authentication serves to validate that the message was encrypted by a known specific party. The party identifier may be as trivial as a well-known static IP address, or may be a proper cryptographic token such as an RSA public key.

In most systems, both integrity check and authentication are performed simultaneously by the same algorithm. A good example of this would be the CCM mode of cipher operation used in the 802.11 specification; the MAC address of the ethernet interface serves as the unique party identifier, and is included in the MIC calculation which validates message integrity.

Note that both integrity check and authentication rely on the security of key material at some level; should the key material be compromised, an attacker may modify, create, or spoof any desired message.

### *4.2 Be Aware of Factory Bootstrap Constraints*

If your cryptosystem is stateless, then all key material and public tokens must be provided to it on startup; however virtually no cryptosystem is truly stateless. It is desirable in nearly all scenarios for even a stateless system to have a unique ID by which to identify and authenticate itself. That unique ID may need to be provided during factory production, or it may be created during first boot of the system. it may even be generated by the system if a suitable hard random number generator is available.

Sometimes hardware unique IDs such as ethernet mac address can be used as a unique ID, but one must be careful not to use exported material as a random number seed for key generation.

### *4.3 Be Aware of Shutdown Constraints*

While startup conditions are almost always well understood, shutdown conditions are much less so. All devices are subject to spontaneous abnormal power-down, and the design of your cryptosystem should be such that abnormal power-down does not compromise the integrity of the cryptosystem. While an obvious statement, this can often have subtle consequences based on the selection of cryptographic primitives.

One simple example would be a data layer using counter-mode encryption that is expected to retain its keys across boots or until rekeyed. The use of counter mode requires that no counter be used twice for a given key. One way to guarantee this is to keep an up-to-date record of the last used counter in persistent storage.

Depending on the type of persistent storage, this may not be possible. Flash memory has a limited number of write cycles, and may simply take too much time to use for every counter value. One possible solution is to pre-allocate blocks of counters from the overall counter range, such that no two blocks may be reused. On reboot of the system, always allocate a new block to guarantee that reuse will not occur.

### *4.4 Have the Right Hardware*

Three primitive pieces of hardware which are generally useful for cryptographic applications are:

1) persistent storage
2) strong random number generation
3) battery backed system clock

All too often, embedded crypto devices are designed without software in mind, leaving the keying layer with clumsy flash-based persistent storage and no strong entropy source onboard. Battery backed time-of-day clocks are also often ignored, as they add extra expense.

Note that the data layer typically only needs item 1, persistent storage. This may be needed to track counters across boots or even remember loaded keys. The data layer may also require system clock operation to expire keys with a limited lifetime, though this is less common in consumer applications.

Strong random number generation is always useful, but much more critical for key generation applications.

### 4.5 The Rule of Least Exposure

With today's software, operating system compromise may be more likely than cryptosystem compromise. While no design can withstand dedicated scrutiny by an attacker with complete access to a device, any time bought resisting system analysis improves the odds of intrusion detection and blacklisting of the compromised system.

To limit exposure in such a scenario, use and copying of cryptographic material should be limited to a small, local area. Unused memory and disk storage should be actively cleared, not just orphaned; wrapped keys should be stored when possible, and only unwrapped when used. Paging to disk should be understood if applicable, as well as system debug facilities that may be used against the system should the operating system be compromised.

### 4.6 System Self Test

System self-test at startup and at periodic intervals may be required as per specification, or may be desirable for other reasons. Detection of system hardware and software anomalies is the most obvious use of such testing; implemented properly it may also serve to detect tampering and other types of attacks.

An analysis of the benefits of a system self test should be included in the design documentation.

# 5 Data Layer Design

This layer is a consumer of key material. Its purpose is to encrypt and decrypt data as that data passes through the system. A fundamental part of this layer should also be to authenticate decrypted data and verify message integrity; in our opinion, no cryptosystem is complete without message integrity checks and tamper detection.

The data layer typically relies on the security of the keying mechanism to perform its function. So long as there are no fundamental design flaws (including unanticipated attack vectors) and the data layer is properly implemented, the security of the layer should be as strong as the keys it uses.

Once you have a thorough understanding of the responsibilities of the data layer, you can begin designing this part of the system. This section does not attempt to detail all parts of the data layer design, but rather focuses on two portions of the system we have traditionally found to be problematic.

## *5.1 Select Primitives*

Where possible, use an existing hard crypto specification. Generally such specifications list cryptographic weak and strong points, as well as the various types of attack resistance. Searches for attacks against a well known crypto system will also provide a larger list of attack vectors than you might find for a less-well used cryptosystem.

Often, some of the low level primitives will be mandated by the application. For example, the use of AES-256 may be required for US government applications, but the manner in which AES-256 must be used may be unspecified.

In one of our recent applications, AES-256 was specified as the only acceptable block cipher primitive, and we chose the CCM cipher mode of operation to produce authenticated and encrypted packet data. The usage of CCM mode was derived from the 802.11 CCM encryption mode, using many of the same fields and CCM construction constants. This helped minimize the amount of analysis required of the final cryptosystem, and provided a large body of known attacks to consider in that analysis.

## *5.2 Be Aware of Startup Constraints*

As much as you may wish the data layer to be a stateless, unintelligent piece of software, that may not be possible. The algorithms and primitives selected may require a complex amount of state to be preserved across startup events, or even a substantial amount of persistent storage.

One obvious scenario in which this can occur is when using counter-mode primitives. With these types of primitives, the security of the system is based on the secret key material, but also on a publicly available counter value. The counter must be unique for every block encrypted with the key; if the same counter is used twice with any key, the block data becomes almost trivially recoverable.

One way to handle scenarios like this is to force the keying layer to provide counters and other data along with key material. If this is not possible, the data layer must perform this function.

# 6 Keying Layer Design

The keying layer is often more complex and easier to attack than the data layer. Special care must be taken to understand all aspects of the keying layer, including:

- Key generation using a hard RNG or PRNG
- Authentication and validation of endpoints
- Protected transfer of key material to the data layer
- Protection of configuration and operational changes to the data layer

The following sections provide general design rules and may point out specific problem areas we have encountered in the past. We will not attempt to detail all portions of the design of the keying layer; this must be done on a per-application basis.

## *6.1 Identify Isolated Components and Interfaces*

Ideally, the keying layer can be broken up into well-defined, somewhat isolated components with well understood interfaces and contracts. More commonly than in other software systems, portions of the keying layer may require specific hardware access or be spread out across multiple devices, each with its own security boundaries and interfaces. Care must be taken to identify each component, its interface layer, and its startup/shutdown constraints. Authentication of a component may be required before it can be used by another component or the system in general; this is often the case in JTRS compliant systems.

## *6.2 Select Primitives*

As the type of key material is often specified by the data layer, selection of primitives often involves selection of suitable RNG, key generation, signing and authentication algorithms. The RNG is the single most important primitive, and should be implemented using hardware entropy support if at all possible. Poor random number generation can compromise utterly (or merely greatly reduce) attack resistance.

Key generation algorithms vary based on the key and algorithm types. RSA keys are 'found' somewhere in a very large keyspace and then verified by various algorithms; AES keys may simply be a string of purely random digits. Other algorithms may have very specific constraints on key generation and organization.

Signing and authentication algorithms should provide a level of security suitable for the expected lifetime of any message authenticated by those algorithms. One rule of thumb is to define the lifetime of any message or data session to be ten years beyond the anticipated lifetime of the cryptosystem. Whenever determining the level of security suitable for such a lifetime, we recommend using an exponential curve such as Moore's law to estimate increases in computational power and attack improvements.

## *6.3 Select a Key Transfer Mechanism*

Key material is useless if it cannot be transferred to the data layer. In some scenarios, such as the 802.11 CCM layer, the keying layer and data layer may be trivially merged and this becomes a non-issue. Military applications reside at the other end of the spectrum, where at no point are the keying layer and data layer allowed to be in direct contact with each other; only through intermediate 'key fill' devices may key material be transferred between the layers.

Various types of key transfer mechanisms may be used, including:

- public key encryption to set up an authenticated session for key exchange
- public key posting, where keys are public key encrypted and posted publicly for retrieval
- black-key symmetric key wrapping, such as used in US military key fill devices
- IKE or other standardized Internet key exchange mechanisms
- physically secured key transfer

Typically at least some of these transfer mechanisms will be excluded based on either the problem definition or other design constraints.

## 6.4 Protect Configuration Data

Often, the configuration of both the keying and data layers is performed by the keying layer, or via keyiing layer services. Attacks against the configuration (or misconfiguration ) of the cryptosystem must not be overlooked; many previous cryptosystems have been laid bare by configuration or other side channel attacks which completely bypass well designed and well implemented security systems.

A good rule of thumb is to treat all configuration data as though it were only slightly less important than key material. All configuration changes and updates should be validated and authenticated as coming from a proper source; if possible, bad configurations should be locked out or not permitted.

## 7 Conclusion

No short whitepaper can detail all the intricacies of cryptosystem implementation. We have attempted to describe some of the techniques and problems we have encountered, but this is by no means a complete or full list. The viewpoint in this paper is also biased toward certain types of designs, and other systems may have vastly different requirements or constraints.

In the end, most of the points of this paper come down to having a detailed understanding of the system and each of the critical cryptographic components - in short, doing a proper engineering design and analysis. All too often this is ignored for traditional software development.

As a final note, it is critically important to find experienced people for this work. Few have a solid background in cryptography; even fewer have experience with system design and implementation. Allowing an inexperienced engineer to design a cryptosystem is risky at best.