Designing A User Experience (UX) for Encryption Key Loading

Malcolm Weir[†]
[†]Ampex Data Systems Corporation, Hayward, California USA,
mweir@ampex.com

Abstract:

Data-at-Rest Encryption is now generally mandated in some form or other for most new systems (as well as many established ones). While there are a few different approaches to implementing the encryption mechanism(s), there are vastly more issues involved in deciding how the keys get loaded into the system for use. Understandably, security authorities are often unwilling to share the specifics of a given approach for a given platform, but there are several common concepts that may apply when considering what might be appropriate for a new system. However, there is no universal key loading interface, nor is one likely to be appropriate. The latest NATO Advanced Data Storage Interface (NADSI / STANAG 4575) has had to address the lack of this universal method and is used as an example of how any user experience / user interface can be accommodated.

Key words: *Encryption, User Interface, System Usability, Interoperability, Standards Development*

Introduction

It is often said[1] that "encryption is easy, key handling is difficult!". While this aphorism is rather dismissive of the efforts expended by cryptologists and the relevant national certification bodies such as the Common Criteria certification and recognition bodies as well as the laboratories that support them, it is very true that the business of key handling involves the innate complexity of human beings and the purposes to which they put these systems.

This paper discusses some of the issues that the designers of key handling systems for encryption will generally have to consider. As it happens, the NATO Advanced Data Storage Interface has been being updated to include an "umbrella" key handling mechanism that is intended to support as many potential implementations as is possible.

Background

Prior to the year 2000, serious data-at-rest (DAR) encryption products were pretty much considered entirely the province of the government: export restrictions and a lack of transparency in how algorithms worked put those DAR products squarely in the category of things that, if you needed the product, you'd put up with the user experience they gave you.

So for US government applications, this meant using the High Assurance "Type 1" product that most closely fitted your needs (or having someone create a brand new one and getting it certified) and the "approved" key handling system, which usually meant "key squirters" like the venerable KYK-13 and the ubiquitous "DS-102" port.

Figure 1 KYK-13 in the National Cryptologic Museum, Maryland, USA.

The reality of this regime was that data-at-rest encryption solutions were both expensive to acquire and costly to operate, but that was generally accepted as just the "cost of doing business". And in general the easier alternative method for securing your data involved solid doors, impressive locks and posted guards!

But with the explosive growth of the public-facing Internet, serious cryptographic tools became

[1] At least, this author says it often.

necessary, leading first to the relaxation of controls on the technology, and second to the availability of efficient, and perhaps most importantly, economical encryption solutions.

And of course, as the financial cost of such solutions grew cheap enough, and implementation easy enough, basic cybersecurity hygiene and externally mandated policy instituted requirements for encryption, including for data-at-rest. And with those requirements, the operational cost rose to the forefront: if a particular solution is cumbersome and challenging to use, one might predict that users will seek ways to make it easier, which typically results in introducing vulnerabilities.

A simple (and a little simplistic) parallel may lie in efforts to improve the complexity of passwords: the more complicated the policy is (upper and lower case, numbers, special characters, etc.), and the more frequently changes must be made, the more likely frustrated users will just write the password on a Post-it® note stuck under the keyboard! Which is all to say that "*Les champions Rouge et Noir*" is probably a better password than "T0ulousa!n"[2]

So having a key handling system that is at least considerate of the user experience is desirable for both aesthetic reasons and security ones.

Key Handling Schemes

Key handling is, broadly, the majority of the infrastructure that supports the encryption system. There are, of course, other components (firmware updates and certificates, and so on) but these are outside the scope of this paper since they tend to be handled using the specific maintenance strategy for the device in question.

The various types of key handling mechanisms can be summarized as described below:

(A) Remote Management. From an implementation standpoint, this approach is the easiest, and indeed is the typical method used for data-in-transit systems: some other system, perhaps the mission computer or something similar, sends the keying material over an internal interface (often MIL-STD-1553 for the data-in-transit variety), often using the central "OFP" as the mechanism by which keys are transported.

Even though Remote Management is convenient (for the DAR solution), usually a secondary method is also needed for downloading data once the encrypted data is removed from the vehicle.

(B) "Console" Loading. By any measure, this is the simplest concept: the system prompts for a key on a console-type interface (e.g. a serial line), and the operator types it in (or, more likely, these days, cuts and pastes it).

(C) "Token" Loading. This approach uses a physical token or "dongle". Figure 2 shows a number of token styles: a front panel token (akin to a USB thumbdrive, but with different contacts), a latching "plug" type, a "bar" type with the contacts on the large area – the reverse side of this image, and a "key" style.

Figure 2 Various Token Styles

At their simplest, tokens just provide some memory in a convenient form-factor, but they can be extended to include cryptographic features, such as:

- Attestation: the token came from a trusted source and has not been tampered with.
- Authentication: the token won't release its keying data except to a trusted recipient.
- Auditing: the token tracks where it has been been.
- Acceleration: the token can include embedded logic to perform cryptographic operations for an otherwise low-power device; this also provides a level of certainty that the operations are not (as) vulnerable to being tampered with.

Tokens can also be Smart Cards, such as the US Department of Defense Common Access Card (CAC), such as the one shown in Figure 3; these are probably the most commonly used style of token.

[2] Unless you're from Dublin, obviously.

Tokens almost always are used in conjunction with a PIN code.

Figure 3 US DOD CAC Token

(D) Key Loader Devices. As the KYK-13 shown in Figure 1 demonstrates, a very typical scheme is to use a key loader device that attaches to a dedicated interface. Sometimes called a "key squirter", this approach often combines the features of the cryptographic tokens with features like an on-board power supply – a battery – that can provide enough energy to activate enough of the host system to be able to transfer the keys.

(E) Biometric Key Loading. This scheme is not often applicable to most test instrumentation applications, due to the severe environments typically involved: do you want to remove your gloves in -40°C weather for a fingerprint reader? Place your eye against an sub-zero iris scanner? But biometrics are mentioned here for completeness.

Key Use

Broadly, there are two types of entities generically referred to as "keys": credentials, which are used to unlock the real keys that are used for encryption, and the actual bits that become an input into the encryption function.

The two varieties are sometimes referred to as "key encrypting keys" (KEKs) and "data encrypting keys" (DEKs). See Figure 4.

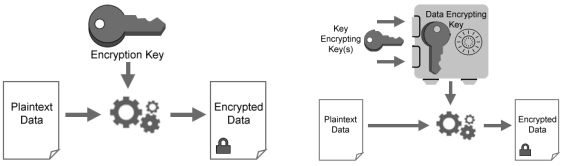

Figure 4 Two Types of Key Data

In practice, the second variety (the "actual bits") are often perturbed by factoring in additional data (such as the serial number of a storage device) so as to conform with cybersecurity "best practices" by never reusing the same DEK for two different tasks (so ensuring that the same plaintext data, perhaps a disk partition table, is never encrypted with the same key in two different settings). The primary difference between the two is that a credential or KEK has absolutely no connection with the algorithm's key except that it can be used to "unlock" the latter, while the DEK is the critical input to the encryption/decryption process. Indeed, the credentials/KEK can be entirely different in form and length from the binary DEK, perhaps having the former be a lengthy text passphrase while the latter is a 256-bit binary value.

It follows that it is entirely possible to change a credential / KEK without re-encrypting previously encrypted data, but changing a DEK requires rewriting everything using the new keying value.

Key Generation

There are broadly two types of key: those generated by a central key issuance body, and those generated "organically" by the specific encryption solution.

Centralized issuance has the advantage of a well-known key quality; the entropy is likely as close to truly random as they can get, so that the probability of selecting any one key value in a 256-bit key is as close as possible to the "perfect":

$$P_{VALUE} = 2^{-256}$$

The challenge with central key issuing structure is that it brings a requirement to securely distribute the keys generated to the point of use. While hard, this may be beneficial in that it makes simultaneous wide-area distribution possible, which is of course the goal with things like communications keys: you need to get the same key to both the transmitter and receiver for communication to occur.

By contrast, organic (or "local") key generation makes for solutions that are easier to scrutinize for security and integrity, and for certification as appropriate. Since the whole system includes an assessment of the quality of the key generation, an evaluation of the system will reflect the strength of the key as well as the integrity of the key loading process and encryption algorithms.

One of the reasons that organic key generation has gained in acceptance is that the availability and quality of the random number generators has improved dramatically with modern components and standards guidance such as the US National Institute for Science and Technology's (NIST's) Special Publication (SP) 800-90 series, including SP800-90A, SP800-90B and the (still in draft form) SP800-90C.

With these tools, and test mechanisms to assess the quality of keys, using a microcontroller

costing less than $10 or $15 one can obtain a random number entropy as good as:

$$P_{VALUE} = 2^{-252}$$

The benefit of having organic key creation is that the user interface can be designed using terminology relevant to the user's taks; for example, instead of something very non-specific like "Request Association Key Pair" on a menu, the user can be presented with "Add Recorder to Existing Squadron Group": the results of choosing that option might be the same, but the latter uses the language that's more relevant to the user.

Key Validity Duration

A critical aspect of any key handling system is the duration that the key will be valid. This validity period might be applied to the key creation / generation time, and / or the time after the key is loaded into the encrypting system.

For example, it may be that the design calls for a key to be usable for only 7 days after creation, and 8 hours after being loaded into the encryptor; the former perhaps to minimize risks of inadvertent key disclosure ("spillage": even if an adversary gets hold of the key, they only have a week during which it's valid) and the latter to act as a sort of "auto-logout after a period of inactivity" mechanism.

However, the most common form of validity enforcement comes from a requirement to be able to load keys into a device well before an operation (e.g. a test flight) so that the system can be tested and verified as operational before the operation actually begins (e.g. so that faulty subsystems can be replaced without delaying the main tasks). So, for example, the keys can be loaded the night before a test operation.

Frequently, this goal results in having the key storage be, in some way, non-volatile so that they can "survive" the appropriate duration without external system power.

Having some kind of key storage that's non-volatile (albeit even for a relatively short period) creates new challenges; the easiest approach is simply to keep power to enough of the system to preserve to data, but low power and high performance are typically orthogonally aligned requirements and providing the energy storage needed to accomplish that is usually disfavored for size, weight or safety reasons, or all three.

This typically leads to one of three strategies:

i. A non-real-time approach where the stored keys are periodically checked for validity, and removed "after the event" if not.

ii. A dedicated key store module with a real-time clock and local processing power that can autonomously destroy the key material after the specified interval.

iii. Using non-sensitive ("public") keying material with an encryption scheme like Ampex's Turnstile method. This approach may only be suitable for certain applications but provides an "easy" solution where it's practical.

All of those strategies provoke a need for some level of anti-tamper protection. For example, writing a sensitive key to an internal regular Solid State Device (SSD) compromises the overall solution in the SSD can simply be removed and the key extracted; similarly, if the key store module can simply be disconnected and interrogated by an "imposter" host to deliver up its keys, the system is vulnerable. And while the public key used by Turnstile doesn't need to be protected *from disclosure* it is necessary to protect it against unauthorized modification (forgery).

Most of these vulnerabilities can be addressed with suitable cryptographic tools: protecting an internal SSD vulnerability can be addressed by using disk encryption (akin to BitLocker on Microsoft Windows) that ties the SSD to that specific host e.g. using a Trusted Platform Module. An even better refinement would be to super-encrypt the file on the encrypted SSD so that even if you compromised the SSD encryption somehow, another layer protects the keying data.

The most attractive (and also most secure) strategy is to use a dedicated key store device. This could be attached to the host system using a convenient interface like USB or another lightweight choice like I^2C or SPI, have its own power source, and contains a clock or counter that can trigger or wake-up the controller after the appropriate interval. The "deluxe" variant of this contains cryptographic authentication to the "other end" of the interface, so that it would refuses requests to store or retrieve the key (or set the duration value, etc.) unless the requesting host has the proper credentials, and would contain anti-tamper provisions so that the whole subsystem can be encapsulated to defeat "probing" or other forensic efforts to extract the data.

Typically, the power source would be a battery; keeping a modern microcontroller "alive" (but mostly asleep) for 24 months on a standard-sized single cell is practical, so recommending replacement every 12 months makes for a reasonably maintainable solution, especially

since the microcontrollers that might be selected for this task typically have analog inputs that can monitor the battery health.

However, an interesting variant is to use a supercapacitor as the power source, which would naturally discharge itself over the sort of durations involved – often a day or so is the maximum expected. As the voltage from the supercapacitor drops, it can eventually trigger a tamper event, which will purge an internal private key (not accessible outside the microcontroller's environment), which will make the stored keying data permanently irretrievable.

Use Case Example 1

This example covers an operational (non-test) application, where the end user wanted to secure and authenticate logging data collected over a period of months. The data would be accessed only following an anomaly or as part of an *ad hoc* survey (e.g. to collect statistics or verify that the data was being collected as expected), and periodically be purged during scheduled major maintenance on the vehicle in such a way that the old data was not retrievable.

This example can be solved by a combination of the Turnstile encryption scheme and a secure key store. The "heavy lifting" of encrypting the data is performed by Turnstile using a public key loaded from the key store. The key store ensures that the public key has not been compromised, as using a fraudulent public key would encrypt the data but without any mechanism to decrypt it. The key store uses a pair of batteries to provide at least two years of tamper-resistant key storage and is set to trigger a tamper-event when experiencing crash loads, so that an adversary cannot use the stored keying data, for example, to forge the logging data after a crash.

The user ecosystem generates the keys needed for Turnstile (in this case, a 4096-bit RSA key pair) using a GUI on the ground download station. This then writes the public key to a FOD-aware locking memory token (dongle), while sending the private key to secure storage within the data processing facility.

As and when the public key needs to be transferred to the vehicle (either because the key is new, or because the previously loaded copy was purged — either accidentally or by design), the dongle is attached to a loader device that contains a battery and a real-time clock. When attached to the vehicle (via a USB Type C connector without the USB data signals connected!), the battery powers (just) the key load device and the key is transferred.

When required, the host recording platform obtains the keying data over by using a USB interface that sends cryptographically signed requests to the key store microcontroller. And when a crash event is detected (via an accelerometer attached to the microcontroller), the host platform is immediately halted by cutting power to its CPU, which ensures that any copies of the key in the platform's memory is removed.

Use Case Example 2

This example is a "IRIG 06 Chapter 10" flight test acquisition system that must encrypt acquired data using a self-encrypting SSD.

Keys are generated using a standalone application running on the download station, a checksum is then calculated from the newly generates key, and then the pair simply copied (cut-and-pasted) to a file on an (security personnel approved) USB stick.

This stick is then attached to a Ground Support Equipment (GSE) laptop, which has a command line interface (IRIG 106 Chapter 6) open to the Chapter 10 recorder. The key and its checksum are then copied and pasted into the command line windows as part of a ".KEY" command.

When the new key is received, assuming it matches the checksum (and if it doesn't, the ".KEY" command just returns an error code), the recorder creates a new file on its internal logging SSD volume, encrypting this file with a private key generated by and stored in the recorder's TPM. Together with the key, the file contains a timestamp derived from the recorder's current real time from its battery-backed clock and a configurable "persistence" value / duration, which defaults to 12 hours.

As an early part of its start-up procedures, the recorder then access this file (using the private key from the TPM) and checks to see if the current time (per the real-time clock) is within the window of the duration from when the key was loaded, and if not the TPM key is overwritten and the encrypted file deleted. The same checking / erasure process occurs any time a memory module must be mounted e.g. the module is removed and replaced.

Obviously, this scheme can be defeated by maliciously adjusting the real-time clock, setting it backwards in time so that the clock always appears to be within the specified window in time. But in order to do that, the physical security of the system would have to be seriously compromised, and counter-measures can be developed to try to detect clock/time anomalies, e.g. by examining artifacts such as the system log to verify that the current time is after the time of the last message, and if not, purge the key file immediately!

NATO STANAG 4575 & NADSI

NATO's STANAG 4575 and the related engineering document ("AEDP-6": Allied Engineering Document) defines a download interface for data collected onboard Intelligence, Surveillance, and Reconnaissance aircraft. The interface defined by the STANAG is termed the NATO Advanced Storage Interface ("NADSI").

A simplistic explanation of the rationale behind this STANAG is that the coalition wanted to be able to have aircraft from one member nation land at a base operated by another nation and have the data collected by the aircraft accessible by the analysts on the "foreign" base. A perhaps more mundane scenario is simply that the imagery intelligence analysts at a shared base can use a single piece of hardware to ingest data from multiple aircraft platforms, rather than having to have a roomful of devices, one dedicated for each potential data recorder type.

It is the second, mundane, scenario that was mirrored by the flight test community: a test range might be working with half a dozen or more test recorders, and for each type of recorder to need a dedicated download station was neither cost-effective nor efficient, as each device needed a a slightly different skillset and so on.

So when the Chapter 10 standard was being developed, it was clear that a common download interface would have significant value to the test range users (who, after all, were paying for the equipment), and so the Chapter 10 development committee looked around for a possible solution, and adopted the NADSI, incorporating the text of the STANAG into the IRIG standard, rather than simply referencing the one from the other.

The IRIG standard and the STANAG evolved in parallel, although not in lockstep. But by the year 2020 both had independently reached a technical dead-end: rather than providing access to a higher-level abstraction of data in terms of files, they provided an interface to read and optionally write low-level storage blocks, and overlaid onto that interface was an unsophisticated filesystem structure that allowed for the identification of groups of contiguous blocks as a file, but no directories, non-contiguous files, etc.

NADSI with File-Level Access

So for the next revision to the download interface, known as STANAG "Edition 5" although it's properly known as AEDP-6 Edition C Version 1 (as the STANAG itself only says "do as the AEDP instructs you to do"), it was very clear that a file-based access method using networks would be required: a networked file system. Of the two dominant flavors in use today, neither had an advantage in terms of availability of differing operating environments, licensing or performance, but SMB, the one originating with IBM and moving through Microsoft Windows offered advantages over the NFS, the one that started with Unix, Solaris and Linux.

The core advantage that SMB offers is that it allows a user to specify a host to validate a logon attempt, while NFS centralizes authentication (SMB allows usernames to be specified as "HOST\USER"; of course, if the target host is not part of a larger domain, that only gets you authenticated for that specific host, which is reasonable for this use case).

Good cybersecurity hygiene requires authentication even if the username and password used is insecure (think username "user" and password "password"!), and since it's unlikely that a network administrator would look favorably on joining an itinerant memory module to their network domain simply to perform a one-time download, SMB was selected.

Since a robust network file system requires processing power commensurate with the desired data transfer performance, the NADSI model has shifted from having a detachable memory module into having the whole recorder system being treated as the entity from which the data is extracted. With download performance requirements for some programs now featuring multiple 100Gb/s of bandwidth, the adjustment from thinking about "just" plugging in a memory module to having to support the whole data system is necessary, if regrettable.

NADSI Edition 5 Key Encryption & Handling

Contemporaneously with the efforts to define how Edition 5 would function were increasing expectations for data encryption, culminating in US and other government mandates that data *shall* be encrypted.

Despite fond hopes from some participants in the process, it is obvious that mandating a single specific technical encryption solution is not a viable approach, not least because while NATO operations might encourage data sharing, NATO members also operate non-NATO missions (either purely sovereign operations or operations as part of other alliances), and the coalition cannot override a country's own priorities and interests. So while e.g. Germany, the UK, and the USA may all agree on the need for encryption, they are very unlikely to agree on the implementation, although ironically they all will likely use a Belgian algorithm ("Rijndael", better known as the Advanced Encryption Standard, AES).

Perhaps not surprisingly, along with the impossibility of imposing a single encryption solution on a group of disparate users each with their own preferences comes the impossibility of having a single key handling strategy.

Instead, imperfect though it is, a concept of providing tools and guidance is being implemented.

For NADSI, an encrypted storage system will send to the client that would like to download data a link to an HTML web page (hosted on the storage system) that guides the user on how to unlock the encryption: graphics to show how the token/dongle, if any, is inserted and prompts to enter the various text strings representing keys and checksums and provide simple normalization and validation (e.g. that when hexadecimal digits are required, only valid characters are entered, etc). This web page, written as usual in HTML, would be served by the storage system using the encrypted HTTPS protocol, providing security against eavesdropping on the key management operations (despite the improbability of that scenario being a signfiicant issue in the real world).

Note that for this download interface, it will typically be necessary to provide entries for keys that are “normally” delivered by a Remote Management system, because it’s unlikely (but not impossible) that the Remote system will still be connected to the data storage system, and if it is connected, that it’s powered on!

The contrast to this approach is the low-level “.KEY” command being implemented for Chapter 10 devices. While text-based interfaces are valuable, it’s undeniable that they are less “user friendly” than GUIs, and the framework around those commands requires more external knowledge of the encryption system; the expectation is that the user will have convenient access to documentation and will be willing and able to go find and then use it. To be sure, this is a not unreasonable notion for flight test instrumentation staff at major ranges or manufacturers, there is always room for usability improvement.

Summary

The hard truth is that when it comes to encryption and particularly key handling for encryption, one size does not fit all. Trying to shoehorn a specific approach into a wide cross section of applications leads to the expected outcome of having multiple imperfect solutions, which is unsatisfactory in general and worse when talking about physical and cyber security.

So rather than looking for the mythical “silver bullet” that will solve all problems, it’s necessary to step back a little and look for frameworks and toolsets that establish a common perspective from which useful and usable user interfaces can be assembled.

As usual, though, security only provides nebulous benefits: in an ideal world, it’s a complete waste of effort and is never needed or used. But our world is far from perfect, and accidental or malicious damage that would have been prevented by better security is commonplace.